25th High Performance Computing Symposium (HPC 2017)

2017 Spring Simulation Multi-Conference (SpringSim'17)

Simulation Series Volume 49 Number 3

Virginia Beach, VA, USA
23 - 26 April 2017

Editors:

Lukas Polok	**William Thacker**
Masha Sosonkina	**Josef Weinbub**

ISBN: 978-1-5108-3822-2

Welcome Letter

Welcome from the SpringSim'17 Conference Chairs

On behalf of the Organizing Committee, it is our pleasure to welcome you to the 2017 Spring Simulation Multi-conference in Virginia Beach, Virginia. Virginia Beach is in the Hampton Roads area, which is home to NASA Langley, the Virginia Modeling Analysis and Simulation Center (VMASC) and over 20 military installations from the Army Navy and Coast Guard. Hampton Roads is also the historical hub of the Colonial era of America filled with ancient cities, wineries and civil war sites. The conference is organized by the Society for Modeling and Simulation International (SCS), the World's oldest international M&S society, which, from its inception in 1952, has effectively engaged our community and continues to play a significant role in advancing research and its contribution to practice. SpringSim'17 covers state-of-the-art developments in M&S methodology, technology and application in disciplines as diverse as applied computing, communications and networking, medicine, adaptive and autonomous systems. This year, SpringSim is co-located with MODSIM World Conference, chaired by Eric Weisel. SpringSim'17 and MODSIM together provide an excellent opportunity to learn the state-of-the-art in modeling and simulation.

We have an excellent program to offer our attendees this year. This includes presentation of peer-reviewed original research papers, posters, work in progress, PhD student colloquium, keynote speeches, featured speeches, and tutorials delivered by experts. This year's conference consists of the following eight symposia: Agent-Directed Simulation Symposium (Chaired by Yu Zhang and Gregory Madey), Annual Simulation Symposium (Shafagh Jafer and Jose J. Padilla), Communications and Networking Symposium (Abdolreza Abhari and Hala ElAarag), High Performance Computing Symposium (Lucas Polok and Masha Sosonkina), Symposium on Modeling and Simulation in Medicine (Jerzy Rozenblit and Johannes Sametinger), Symposium on Theory of Modeling and Simulation (Fernando Barros and Xiaolin Hu), Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems (Saurabh Mittal and Jose L. Risco Martin) and a new symposium for SpringSim'17 - Chaired by Andrea D'Ambrogio and Umut Durak - Model-driven Approaches for Simulation Engineering. We would like to thank the organisers of the symposia and their respective technical program committees and reviewers for their effort in putting together the exciting program. As a Multi-conference, our success depends on their contribution.

We have an exciting line-up of distinguished keynote speakers; we would like to express our gratitude to Benoit Montreuil and Pieter Mosterman for accepting our invitation to deliver keynote speeches.

This year we are launching two new initiatives: Featured Speakers and Student M&S Demo Session. The Featured Speakers series brings spotlight to the authors of invited papers in selected symposia. This emphasizes the state-of-the-art contributions the Featured Speaker is making in the chosen field, as considered by the Chairs in the particular symposium. This year we have Bernard P. Zeigler as the Honorary Featured Speaker, along with Andreas Tolk, Neal Wagner, Eric Nielsen, Wes Bethel, Theodore A. Bapty, Umut Durak, Navonil Mustafee and Janet Roveda. We thank our Featured Speakers in defining the bleeding-edge. The Demo Session replaces the earlier Mobile App Competition. It is led by Salim Chemlal and Mohammad Moallemi. It encourages students to showcase their running simulations that they have authored in the contributed papers. We plan to grow the M&S Demo towards an online archive so that each simulation article has an accompanying simulation "execution" to inform the reader in a better way.

We would like to thank our sponsors who have donated money, software licences and books and which has made it possible for us to recognise best papers in the conference, support student travel, and provide an enhanced conference experience for our delegates. We sincerely thank VMASC, Old Dominion University, MOSIMTEC, Institute for Simulation and Training, University of Central Florida, and VMASC Industry Association.

Our sincere gratitude goes to our Organization Committee. We would like to thank Deniz Cetinkaya, Marina Zapater and Marc Banghart (Proceedings Co-Chairs), Saikou Diallo (Sponsorship Chair), Navonil Mustafee (Awards Chair), Andrew Collins (Publicity Chair), Umut Durak (Tutorial Chair), Murat Gunal (WIP Chair), Caroline C. Krejci (Poster Session and Student Colloquium Co-Chair) and Salim Chemlal and Mohammad Moallemi (Student M&S Demo Session Co-Chairs). We would also like to thank SCS Executive Director, Oletha Darensburg and Carmen Ramirez for their conference coordination activities and Mike Chinni for his help with the proceedings and digital libraries.

Thank you for making SprimSim'17 a success through your participation. We look forward to your continued participation in SpringSim'18.

Saurabh Mittal
General Chair
The MITRE Corporation
USA

Gregory Zacharewicz
Vice-General Chair
University of Bordeaux
France

Andrea D'Ambrogio
Program Chair
University of Rome Tor Vergata
Italy

HPC’17 CHAIRS’ MESSAGE

Welcome to the 2017 High Performance Computing Symposium!

This is the 25th special symposium devoted to the impact of high performance computing and communications on computer simulations. The symposium encompasses a wide variety of topics with a focus on tools and applications for the simulation of physical and engineering systems.

Advances in multicore and many-core architectures, networking, high end computers, large data stores, and middleware capabilities are ushering in a new era of high performance parallel and distributed simulations. Along with these new capabilities come new challenges in computing and system modeling. The goal of HPC 2017 is to encourage innovation in high performance computing and communication technologies and to promote synergistic advances in modeling methodologies and simulation. It will promote the exchange of ideas and information between universities, industry, supercomputing centers, and national laboratories about new developments in system modeling, high performance computing and communication, scientific computing as well as simulation.

Sincerely, **Lukas Polok** General Chair
William I. Thacker Program Chair

Masha Sosonkina General Vice-Chair
Josef Weinbub Program Vice-Chair
Karl Rupp Publicity Chair

MATRIX-FREE FINITE-ELEMENT COMPUTATIONS ON GRAPHICS PROCESSORS WITH ADAPTIVELY REFINED UNSTRUCTURED MESHES

Karl Ljungkvist

Department of Information Technology
Division of Scientific Computing
Uppsala University
Box 337
SE-751 05 Uppsala, Sweden
karl.ljungkvist@it.uu.se

ABSTRACT

This paper concerns efficient matrix-free finite-element algorithms on modern manycore processors such as graphics cards (GPUs) as an alternative to sparse matrix-vector products. In matrix-free finite element algorithms, the assembly and solution phases are merged, yielding a significantly lower memory bandwidth footprint, with a corresponding increase in efficiency on bandwidth limited processors. Additionally, no system matrix must be assembled or stored in memory.

We present a GPU parallelization of the matrix-free method including a novel algorithm for resolving hanging-node constraints on the GPU, capable of simulation on adaptively refined grids. For second-order elements and higher in 3D, our GPU implementation of the adaptive algorithm is between 1.8 and 2.3 times faster than an existing optimized CPU version, on comparable hardware. Compared to a matrix-based implementation using CUSPARSE, we get a speedup of 8 and can solve problems 8 times larger in 3D.

Keywords: finite element methods, GPU, matrix free, adaptive refinement, hanging nodes.

1 INTRODUCTION

The finite element method is a popular choice for numerical simulation due to its capability to easily handle complicated geometries and incorporate adaptive mesh refinement. The conventional procedure for finite-element computations is to first assemble a system of equations, and solve this using an iterative method. However, this two-step approach performs poorly when executed on modern multicore and manycore processors.

The computational core of a finite-element solver is a product between the large and sparse system matrix and a vector, which is performed a large number of times inside the iterative solver. This operation, the sparse-matrix vector product (*SpMV*), needs to fetch an 8-byte double for only every 2 floating point operations (flops) performed, whereas the memory system of most modern CPUs and GPUs can only deliver one double for every 32-64 flops. This means that the performance of the SpMV operation will be solely limited by the available memory bandwidth, and most of the computational hardware will be wasted (see, e.g., Gropp, Kaushik, Keyes, and Smith 1999). In addition to this, the system matrix must be stored in memory which can be a quite severe limit on how large problems can be solved, in particular on GPUs which typically have an

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

order of magnitude smaller memory. Also, the matrix assembly itself can amount to a substantial portion of the total simulation time, especially in applications where frequent reassembly is necessary.

Motivated by these shortcomings a matrix-free approach has been suggested, for the first time already in 1986 for computer systems with limited memory (Carey and Jiang 1986). This idea builds on the important observation that, within the iterative linear solver, the matrix entries are not needed explicitly, but only a recipe for computing the product of the matrix times a vector. The matrix-free multiplication algorithm is especially interesting for finite elements with tensor-product basis functions, such as quadrilateral and hexahedral elements, since for such elements the element-local numerical integration can be performed very efficiently using a sum-factorization approach. As shown by Cantwell, Sherwin, Kirby, and Kelly (2011), the matrix-free approach accesses less data than when using SpMV for elements of order two and higher, whereas the converse is true for first-order elements. Consequently, for order two and higher, the matrix-free algorithm can be expected to perform better on modern bandwidth-limited processors. In Brown (2010), a matrix-free approach is used for a high-order Jacobian combined with an assembled lower-order preconditioner. In Kormann and Kronbichler (2011), the authors present a generic framework for matrix-free computations as part of the open-source finite-element library `deal.II` (Bangerth et al. 2016). The framework uses vectorization, task-based parallelization, and message passing to achieve good performance in a quantum mechanics computation on distributed and shared-memory systems (Kronbichler and Kormann 2012).

Early work on finite-element computations on GPUs focused either on the SpMV operation in the solve phase (Göddeke, Strzodka, and Turek 2005, Dehnavi, Fernandez, and Giannacopoulos 2010), or on the matrix assembly (Cecka, Lew, and Darve 2011, Markall et al. 2013). The matrix-free approach was first used on GPUs in high-order discontinuous Galerkin (DG) or spectral element methods for explicit time stepping of hyperbolic problems. In Klöckner, Warburton, Bridge, and Hesthaven (2009), high-order DG elements are used in a simulation of Maxwell's equations on conservative form. In Komatitsch, Erlebacher, Göddeke, and Michéa (2010), the authors simulate seismic wave propagation in 3D using fourth-order spectral elements on several GPUs. We are not aware of any previous effort conducting finite-element computations with hanging nodes on GPUs.

In a previous paper, we have studied the performance of a matrix-free operator application for a Cartesian mesh where a constant local matrix can be used (Ljungkvist 2014). The present article extends that work to unstructured meshes, and to adaptively refined meshes with hanging nodes. We have implemented the method as a general framework with an underlying GPU parallelization based on CUDA (NVIDIA Corporation 2016), which is in the process of being made officially available as part of `deal.II`. With a unified interface for CPU and GPU backends, which will be the focus of an upcoming publication, we can run the same application code efficiently on both systems, and at the same time offer great flexibility for the application programmer.

The remainder of this paper is structured as follows. In Section 2, we describe the matrix-free algorithm in detail, as well as how we parallelize it for GPUs. In Section 3, our algorithm for resolving hanging-node constraints on the GPU is described. In Section 4, we elaborate on the data structures used. In Section 5, we present benchmark experiments evaluating the performance of our method. Section 6 concludes this work.

2 MATRIX-FREE MULTIPLICATION

The linear system originating from a finite-element discretization of a PDE has a system matrix A which is assembled as a sum of local matrices a_k on all elements k in the mesh. For a thorough introduction to finite-element methods, see, e.g., Brenner and Scott (2002). Inside the linear solver, we want to compute the product of A with a vector u of unknowns, or *degrees of freedom* (DoFs). Now, instead of using a precomputed A, the matrix-free multiplication algorithm is formed by merging the matrix assembly into the multiplication resulting in the following three steps for each element k,

1. Read the local unknowns u_k from the global input vector u
2. Evaluate the local matrix multiplication, $v_k = a_k u_k$
3. Assemble the local contribution v_k into the global result vector v

2.1 Local Matrix Multiplication

In Ljungkvist (2014), we showed that for a Cartesian mesh where all the a_k are equal, using a single precomputed local matrix a gives a competitive algorithm as long as a is small enough to fit in the GPU cache. However, for a general mesh, the distinct a_k would amount to more memory than the corresponding assembled sparse matrix, and thus cannot be favorable neither in terms of storage space nor in terms of reduced bandwidth usage. In this case, a more advanced approach is necessary.

To simplify the discussion, we will now assume that the original PDE is a Poisson equation with variable coefficients. While this is a simple model problem, it serves the purpose of illustrating the method, and can readily be extended to, for instance, a vector valued or non-linear problem. It also appears as part of many more complicated applications, such as in projection-correction methods in computational fluid dynamics.

For the Poisson equation the local matrix is defined by

$$a_{ij}^k = \int_{\Omega_k} \nabla\varphi_i \cdot A(\mathbf{x}) \nabla\varphi_j \mathrm{d}\mathbf{x}\,, \tag{1}$$

where $A(\mathbf{x})$ is a variable coefficient. If we evaluate the integral by transforming to a reference element and integrating numerically using Gaussian quadrature, we obtain

$$a_{ij}^k = \sum_q \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q)\right) \cdot A(\boldsymbol{\xi}_q) \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q)\right) w_q |\det J_k(\boldsymbol{\xi}_q)|\,, \tag{2}$$

where the $\boldsymbol{\xi}_q$ and w_q are reference-element quadrature points and weights respectively. Furthermore, J_k^{-1} is the inverse Jacobian of the transformation from element k to the reference element, and $\nabla_{\boldsymbol{\xi}}$ denotes a reference-space gradient. Here, we see that only the variable coefficient $A(\boldsymbol{\xi}_q)$, the inverse Jacobians and Jacobian determinant need to be stored individually for each element. For Q_p elements in d dimensions integrated using $(p+1)^d$ quadrature points, these amount to $(d^2+2)(p+1)^d$ entries which is less than the $(p+1)^d(p+1)^d$ entries of the local matrix already for element degree $p > 1$.

Motivated by this observation, we plug (2) into the local multiplication,

$$v_i = \sum_j \sum_q \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q)\right) \cdot A(\boldsymbol{\xi}_q) \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q)\right) w_q |\det J_k(\boldsymbol{\xi}_q)| u_j\,. \tag{3}$$

Rearranging the summations, this operation can be performed by three successive steps. In the first one, we compute the gradients at reference quadrature points,

$$\nabla_{\boldsymbol{\xi}} u_q = \sum_j \nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q) u_j\,. \tag{4}$$

In the second step, we perform quadrature-point-wise operations; we first transform to the real element, then perform any local operations – in this case multiplying by the coefficient $A(\boldsymbol{\xi}_q)$, transform back to reference element and multiply by quadrature weights and Jacobian determinant,

$$s_q = w_q |\det J_k(\boldsymbol{\xi}_q)| J_k^{-T}(\boldsymbol{\xi}_q) A(\boldsymbol{\xi}_q) J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} u_q\,. \tag{5}$$

In the last step, we multiply by basis function gradients and integrate,

$$v_i = \sum_q \nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q) \cdot s_q\,. \tag{6}$$

2.2 Evaluation Using Sum-Factorization

As we saw, formulating a matrix-free algorithm leads to a significant reduction in the bandwidth footprint, but this comes at the price of additional operations. In particular, both when evaluating the quadrature point gradients in (4) and when performing the final integration in (6), d matrix-vector products with vectors of size $(p+1)^d$ must be performed. However, for tensor-product elements, such as quadrilateral and hexahedral elements, considerable simplifications can be made.

For such elements, the reference-element shape functions are tensor products of one-dimensional shape functions, i.e. in 3D,

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi)\psi_\nu(\eta)\psi_\sigma(\zeta)\,, \tag{7}$$

where we have introduced a multi index (μ,ν,σ) corresponding to the single index i, with each component running from 1 to $p+1$. For the basis function gradients, we get

$$\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}) = \begin{pmatrix} \psi'_\mu(\xi)\psi_\nu(\eta)\psi_\sigma(\zeta) \\ \psi_\mu(\xi)\psi'_\nu(\eta)\psi_\sigma(\zeta) \\ \psi_\mu(\xi)\psi_\nu(\eta)\psi'_\sigma(\zeta) \end{pmatrix}. \tag{8}$$

Similarly, the 3D quadrature points can be defined by a tensor product of the one-dimensional points,

$$\boldsymbol{\xi}_q = (\xi_\alpha, \xi_\beta, \xi_\gamma)\,, \tag{9}$$

where $q \to (\alpha,\beta,\gamma)$ is another multi index. Note that we consistently use μ,ν,σ to index in DoF space, and α,β,γ as indices for quadrature points. Using these new indices and the shorthands $\psi_{\alpha\mu} = \psi_\mu(\xi_\alpha)$ and $\vartheta_{\alpha\mu} = \psi'_\mu(\xi_\alpha)$, we can factorize the sum in (4) and get

$$\nabla_{\boldsymbol{\xi}} u_{\alpha\beta\gamma} = \sum_\mu \begin{pmatrix} \vartheta_{\alpha\mu} \\ \psi_{\alpha\mu} \\ \psi_{\alpha\mu} \end{pmatrix} \sum_\nu \begin{pmatrix} \psi_{\beta\nu} \\ \vartheta_{\beta\nu} \\ \psi_{\beta\nu} \end{pmatrix} \sum_\sigma \begin{pmatrix} \psi_{\gamma\sigma} \\ \psi_{\gamma\sigma} \\ \vartheta_{\gamma\sigma} \end{pmatrix} u_{\mu\nu\sigma}\,, \tag{10}$$

where the vector products are to be understood as element-wise multiplication. The same series of operations are obtained for the integration in (6), but with a summation over the quadrature indices instead. In (10), we have a series of d consecutive $(p+1)^2 \times (p+1)^d$ tensor contractions for each d components, which have an operational complexity of $\mathscr{O}\left(2d^2(p+1)^{d+1}\right)$ altogether, compared to $\mathscr{O}\left(2d(p+1)^{2d}\right)$ for the large matrix-vector products in (4). For more details on the sum-factorization evaluation, see Kronbichler and Kormann (2012).

2.3 Parallelization

The matrix-free operator application algorithm contains several types of parallelism. The most apparent one is that the result is computed as a sum of independent contributions from each element. It is thus natural to parallelize the algorithm over the elements, i.e., divide the list of elements into chunks and compute the contribution from each chunk of elements in parallel.

This leads to fairly coarse-grained parallel work tasks consisting of all the local operations on an element. For each element, we first read the local DoF values into local variables and then evaluate values and/or gradients at quadrature points. Then, quadrature-point-local operations such as multiplication with a variable coefficient need to be performed. Finally, we perform numerical quadrature to obtain DoF values, and write back the local DoF values. Throughout these computations, each thread needs to store the local DoF values and the values and/or gradients at quadrature points, plus any additional intermediate variables. These can amount to quite a large quantity of memory per thread.

On a multicore CPU, having a coarse-grained parallelization is usually desirable since per-task overhead is often quite substantial. Also, the amount of local data needed is not an issue, since the threads have a relatively large local memory in the form of cache, which can accommodate all the variables that are necessary for the local operations.

On a GPU on the other hand, parallelism is used to hide memory latency by having more threads than cores so that when a high-latency instruction is encountered, instructions from other threads can be executed instead. Therefore, a higher level of parallelism is often sought for when targeting GPUs. In addition, the per-core local memory is relatively limited, so a too high memory requirement per thread will put a limit on the number of threads per core that can be in flight at once. This motivates looking for more fine-grained alternatives to a parallelization over elements.

From Section 2.2, we recall that the sum-factorization resulted in a series of dense tensor contractions, each of which is of the form

$$v_{\alpha\nu\sigma} = \sum_{\mu} \psi_{\alpha\mu} u_{\mu\nu\sigma} \,. \tag{11}$$

Noting that this operation is essentially a matrix-vector product of ψ with each of the "rows" of v along a given coordinate direction, it is clear that the tensor contractions offer another, more fine-grained level of parallelism. We therefore propose to introduce one thread per DoF on each element, and let threads cooperate in computing each tensor contraction. With this approach, the DoF values and gradients are shared between all the threads in a block, and thus the necessary memory per thread is reduced considerably.

Table 1: Number of cells per CUDA block for different elements Q_p in 2D and 3D

	Q_1	Q_2	Q_3	Q_4
2D	32	8	4	4
3D	8	2	1	1

For low-order elements which have few DoFs per cell, a single cell will constitute a very small CUDA block with too few threads for favorable occupancy. In this case, we pack several elements into one block (see Table 1).

When the contributions from different elements should be assembled into the final result vector, many DoFs will be updated by several threads. To avoid race conditions we use a graph coloring approach, where only sets of elements without shared DoFs are processed in parallel. We use the graph coloring algorithm readily available in `deal.II`, which is described in Appendix A of Turcksin, Kronbichler, and Bangerth (2016).

3 TREATMENT OF HANGING NODES

When using a triangular or tetrahedral mesh, adaptive refinement is easily achieved by subdividing the elements flagged for refinement, and also a small number of surrounding elements in order to keep the mesh conforming. When using quadrilateral and hexahedral elements, this cannot be done as straightforwardly, as subdividing these have a greater impact on the surrounding elements. Instead, for such meshes, a non-conforming refinement strategy is often used, which allows for hanging nodes, i.e. nodes on an edge which only belong to one of the elements sharing the edge. In order to enforce the standard smoothness properties of the solution, the unknowns located on these nodes are not actually independent, but must fulfill some linear constraint coupling it to other unknowns. In this paper, we assume the common case that neighboring cells differ in refinement level by at most one, which guarantees that a hanging node is only coupled to non-hanging nodes. See Section 3.3 in Bangerth, Burstedde, Heister, and Kronbichler (2011) for more details on hanging-node constraints.

For matrix-based methods, it is conceptually simple to first perform the assembly as usual, and then eliminate rows and columns of constrained DoFs from the system. However, it is more efficient to eliminate these on the fly during the assembly itself (see Section 3.3.1 in Bangerth, Burstedde, Heister, and Kronbichler 2011).

With the matrix-free approach, one cannot eliminate the constrained unknowns from the mesh since the sum-factorization evaluation requires each element to have a full set of DoFs. Instead, these constraints must be applied every time the affected unknowns are accessed. In `deal.II`, this is done using a list of local constrained DoFs, which during the access are not simply read from a single global DoF, but computed according to the constraint. This approach has been used successfully for multicore CPUs (Kronbichler and Kormann 2012).

On GPUs, processing the constraints one by one in succession is not compatible with our parallelization with one thread per DoF in each element, since its low parallelism would introduce a serial or almost serial section in the otherwise highly parallel algorithm. Instead, we propose a method which exploits specific properties of hanging-node constraints on tensor-product elements, allowing the threads within an element to cooperate in resolving all constraints at once.

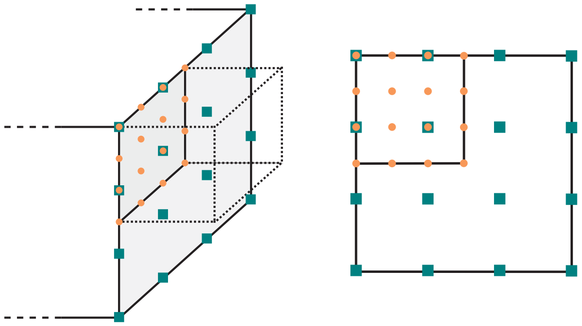


Figure 1: Hanging nodes on a face in 3D, with an extracted 2D view of the face

In Figure 1, we see a typical situation in which constrained DoFs arise in a hexahedral mesh. The DoFs on the fine side (orange dots) are all constrained, and the values they must have to maintain continuity are determined by the value of the basis functions on the coarse side (teal squares). Specifically, the values are computed by evaluating the coarse-side function at the fine points. Since only the basis functions associated with the DoFs on the face are non-zero on the face, only those DoFs will have non-zero weights in the constraint.

For the constrained face shown in Figure 1, each constrained DoF u_i will in general be computed as a unique linear combination of the coarse-side DoFs v_j, weighted by the $(p+1)^2$ shape-function values at that location,

$$u_i = \sum_j \varphi_j(\mathbf{x}_i) v_j\,, \tag{12}$$

where the $\mathbf{x}_i = (x_i, y_i)^T$ are the fine-side DoF locations. This is a relatively large, dense matrix-vector product. However, for the tensor-product elements under consideration, we know that the shape functions factorize into a product of one-dimensional shape functions. This, combined with the similar tensor-product structure of the DoF locations, allows us to make the replacements

$$\varphi_j(\mathbf{x}_i) \rightarrow \psi_\alpha(x_\mu)\psi_\beta(y_\nu) \tag{13}$$

where multi-index substitutions $i \rightarrow (\mu, \nu)$ and $j \rightarrow (\alpha, \beta)$ have been made. If we now insert this into (12), we see that just like in the case of the evaluation of basis functions described in Section 2.3, we can split this operation up into a series of one-dimensional interpolations performed in succession. Using the same multi-indices, the constrained DoFs can be computed as

$$u_{\mu\nu} = \sum_{\alpha} a_{\mu\alpha} \sum_{\beta} b_{\nu\beta} v_{\alpha\beta} \tag{14}$$

where $a_{\mu\alpha} = \psi_{\alpha}(x_{\mu})$ and $b_{\nu\beta} = \psi_{\beta}(y_{\nu})$. This two-step interpolation approach is illustrated in Figure 2.

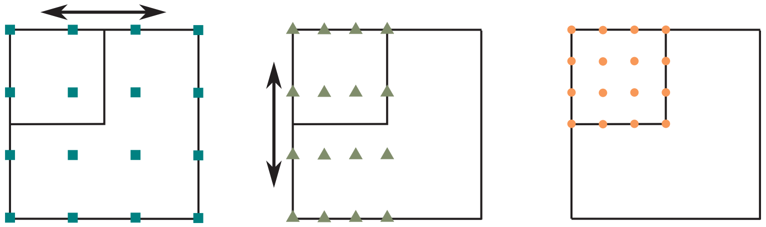


Figure 2: Our method for resolving hanging nodes in a sum-factorization manner; we first interpolate along one coordinate direction, and then the other one.

This approach for the resolution of hanging-node constraints lends itself well to the same fine-grained parallelization as we are using for the basis function evaluation, albeit now restricted to only the DoFs on the face. While this leads to a quite a large number of idle threads, it is still better than the fully serial approach used previously.

In addition to the presently described case of a single constrained face, to which the overwhelming majority of constraints belong, we also include treatment of the exceptional cases where (i) several faces of one element are constrained, and where (ii) only DoFs along an edge of an element are constrained. We do this extending our two-step method to a general three-step process where each step interpolates in one of the coordinate directions, thereby including all applicable constraints on any side or edge at once. In our implementation, we use a 9-bit mask for each element to encode the presence and type of constraints in an efficient and compact manner.

4 DATA STRUCTURES

To optimize the utilization of the GPU memory system, it is very important to choose a data layout which achieves maximum coalescing of memory accesses (NVIDIA Corporation 2016). The memory used in the algorithm can be divided into three types: *per-quadrature data* which is unique to each quadrature point of each element, such as local-to-global DoF index mappings or coefficients; *per-DoF data* such as input and output solution vectors; and *element-independent data*.

With our parallelization with one thread per local DoF, optimal coalescing is achieved if we use an array-of-structure data layout for the *per-quadrature data*, rather than a structure-of-array format which is usually preferred for GPUs. The inverse Jacobian, must be treated in a slightly altered way, since it contains D^2 entries for each quadrature point that are read in succession by the same thread. In this case, an "array-of-structure-of-array" approach is appropriate where for each element we first store the J_{00} components for all quadrature points, then all J_{01} components, etc. In addition, we make sure that memory is properly aligned by padding each chunk of memory to 128 byte boundaries.

For a general mesh, the *per-DoF* solution vectors will be read in a very irregular manner. The fact that many DoFs are shared between elements leads to a conflict of interests where different elements have different opinions on what would be a good ordering of the DoFs, essentially making it impossible to lay out data in a way to coalesce all access. This becomes even more severe for a general unstructured mesh with hanging nodes. The situation becomes somewhat better with higher element order since that increases the portion of non-shared DoFs. However, the solution vectors amount to much less data than the per-quadrature data, namely $2(pn+1)^d / \left((d^2+2)(p+1)^d n^d\right)$ for a d-dimensional mesh with n elements of order p in each dimension, which is roughly 2 - 20% for the elements under consideration. Therefore, we think this issue is not significant in practice.

For element-independent data, such as the values and gradients of one-dimensional shape functions at quadrature points, we first tried using constant memory to allow for caching in the L1 read-only cache. However, it turned out to be more favorable to explicitly stage relevant parts of these in registers.

5 BENCHMARK EXPERIMENTS

We evaluate the performance of our parallelization by running a number of numerical benchmark experiments. In these, we measure the time to compute a multiplication using our matrix-free method by applying the operation 100 times and computing the average time. We do not include time needed to set up the data structures and the time for transferring data from the CPU to the GPU, as in an iterative solver, all the data would reside on the GPU throughout the whole computation. We are using double-precision numbers throughout, since this is necessary for proper convergence of iterative linear solvers.

The benchmarks are based on the same Poisson problem considered earlier, with a 2D or 3D hyper ball domain, homogeneous Dirichlet conditions, and a variable coefficient defined by $A(\mathbf{x}) = 1/\left(0.05+2\|\mathbf{x}\|^2\right)$. This geometry ensures a mesh with non-Cartesian elements, making it general enough to be representable for more complicated geometries. The relatively coarse base mesh is refined uniformly to create a series of successively finer meshes, which lets us study how performance scales with problem size. To see the performance impact of our treatment of hanging-node constraints, we also run the benchmark on a series of meshes with adaptive refinement. Specifically, the coarsest mesh is refined on a series of non-aligned spherical shells, which simulates adaptive refinement along some wave fronts. An example of such a mesh can be seen in Figure 3. Table 2 lists the largest meshes used for each element configuration.

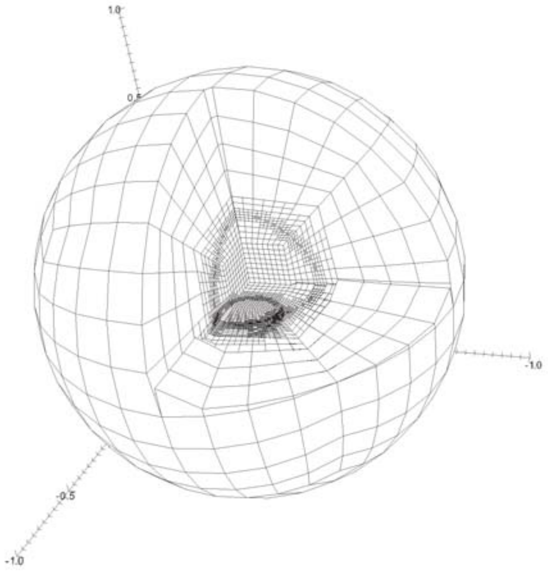


Figure 3: The adaptively refined mesh used to benchmark our method for resolving hanging-node constraints.

We compare our GPU implementation of the matrix-free method with the highly optimized CPU implementation of same matrix-free method by Kronbichler and Kormann (2012), which is readily available in `deal.II`. We also compare against a sparse-matrix-based version for GPU using Nvidias official sparse-matrix library

Table 2: Meshes used in the experiments

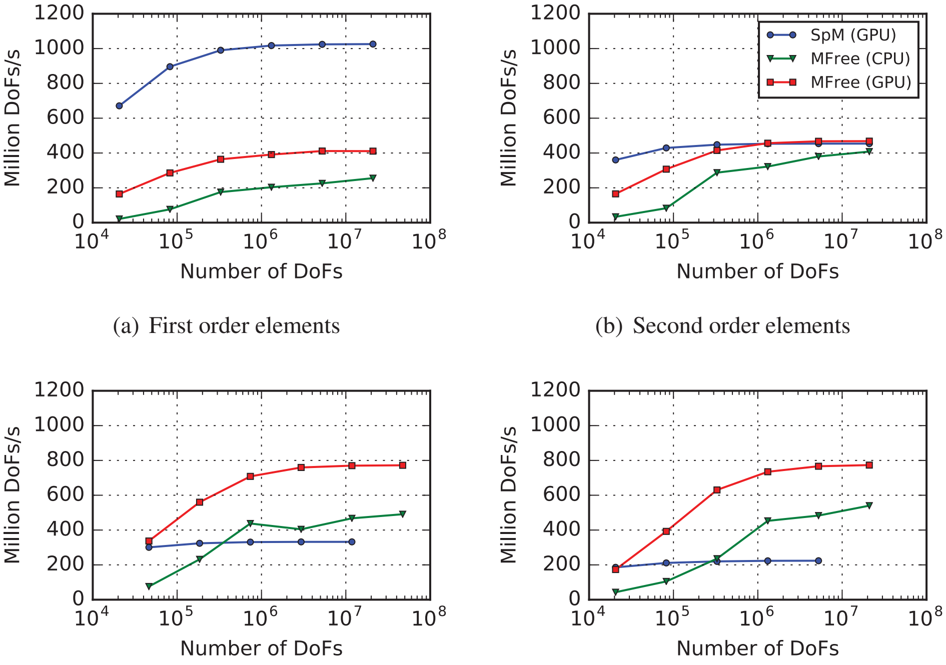
(a) Uniform refinement

d	*p*	# cells	# DoFs
2	1	20 971 520	20 975 617
2	2	5 242 880	20 975 617
2	3	5 242 880	47 192 065
2	4	1 310 720	20 975 617
3	1	1 835 008	1 847 617
3	2	1 835 008	14 729 857
3	3	229 376	6 221 281
3	4	229 376	14 729 857

(b) Adaptive refinement

d	*p*	# cells	# DoFs	% HN
2	1	18 271 217	18 572 584	3.23
2	2	4 847 393	19 992 109	4.51
2	3	4 847 393	44 680 720	3.37
2	4	1 350 302	22 356 367	4.70
3	1	1 223 845	1 522 469	38.3
3	2	1 223 845	11 564 225	25.4
3	3	180 964	5 784 259	24.8
3	4	180 964	13 205 632	19.6

CUSPARSE (NVIDIA Corporation 2013). The GPU experiments were run on a server with an Nvidia Tesla K40, an Intel Core i5-3550 quad-core, 16 GB RAM and CUDA 8 RC. The CPU experiments were run on a system with two Intel Xeon E5-2680 eight-core processors, and 64 GB RAM. The K40 GPU has a TDP of 235 W whereas the two Xeon processors has a combined TDP of 260W.



(a) First order elements

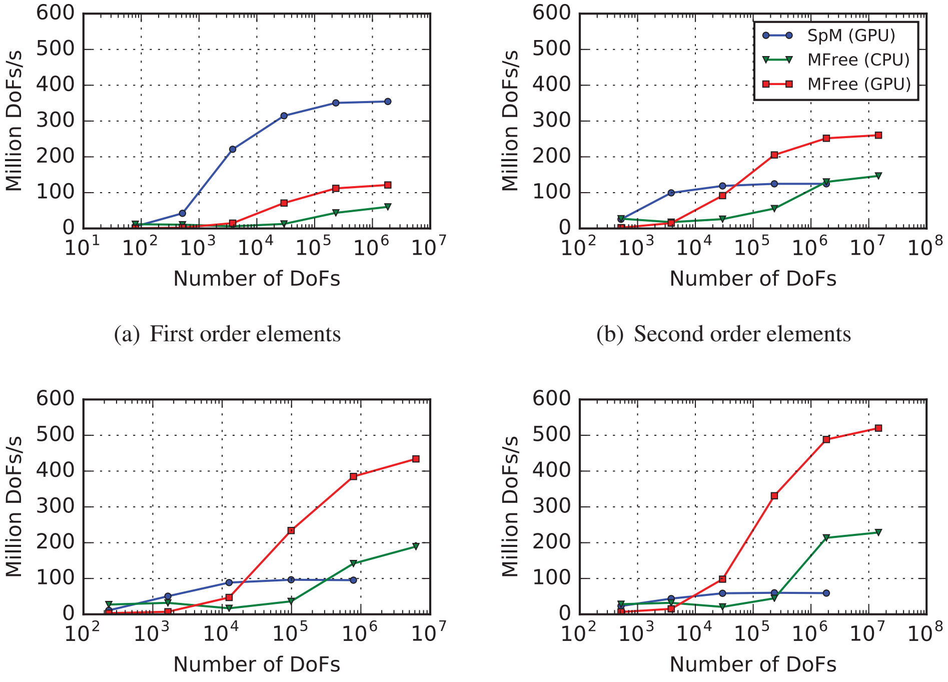
(b) Second order elements

(c) Third order elements

(d) Fourth order elements

Figure 4: Throughput vs problem size for the 2D uniformly refined mesh.

In Figure 4 and 5, the results for the uniformly refined mesh are shown. To facilitate comparison across different problem sizes and implementations with different memory and compute patterns, we present performance as throughput in terms of DoFs processed per second, rather than flop/s or bandwidth. Firstly, we see that, as expected from theory, the matrix-free method is slower than the SpMV version for Q_1 elements, and faster for second order elements and higher. In 2D, the performance gain is very small for second order elements (≈ 3%), but from third order we see speedups of 2.3 - 3.4×. In 3D, we get substantial speedups of 2 - 8.2× already from second order elements. Secondly, our implementation is consistently faster than the CPU version; between 15% and 61% in 2D, and between 1.8× and 2.3× in 3D. Finally, we note that when using a matrix, we get problems fitting it in memory for elements of order 3 and higher in 2D as indicated by



(a) First order elements

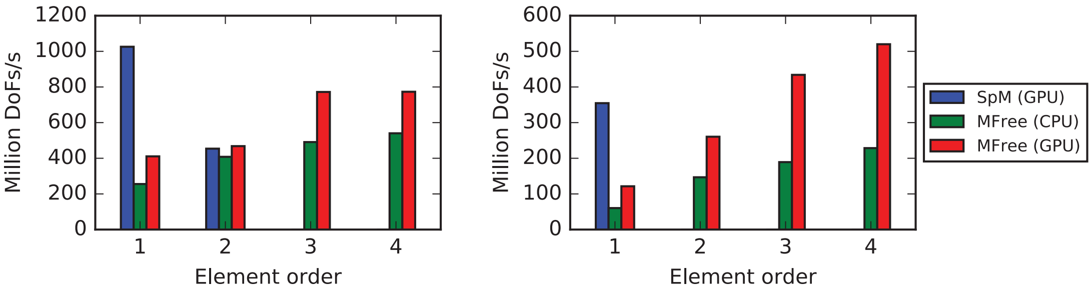
(b) Second order elements

(c) Third order elements

(d) Fourth order elements

Figure 5: Throughput vs problem size for the 3D uniformly refined mesh.

the truncated lines for SpM. In 3D, this happened already from second order elements. In Figure 6, we have summarized the results for the largest problems considered.

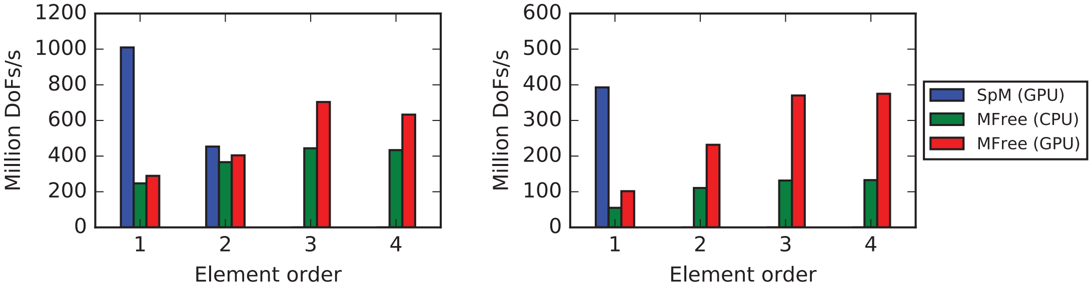


(a) 2D

(b) 3D

Figure 6: Performance for the largest problems solved (uniform refinement). The missing bars for SpM (GPU) indicate that the matrix did not fit in memory.

If we compare Figure 6 with Figure 7, which shows the corresponding results for the adaptively refined mesh, we see that there is a moderate overhead from resolving the hanging nodes. Specifically, the overhead is about 10 - 40% for our GPU implementation, which is lower than the overhead of the CPU version which reaches 70% for some meshes. The matrix-based version eliminates the constrained DoFs once and for all during the assembly, and thus does not see any noticeable overhead. Still, the much better efficiency of our method makes it continue outperforming the matrix-based one from element order 3 in 2D (2 - 2.6×), and from element order 2 in 3D (34% - 4.6×).



(a) 2D

(b) 3D

Figure 7: Performance for the largest problems solved on the adaptively refined ball.

Finally, we add that for Q_4 in 3D the matrix-free version achieves 35% bandwidth utilization out of the specified 288 GB/s, compared to 52% when using SpMV. Considering that 13× less memory is accessed, this is a notably small reduction in bandwidth utilization. Still, there is clearly room for further improvements.

6 CONCLUSIONS

We have developed a framework for matrix-free finite-element computations on graphics processors, supporting adaptively refined meshes with hanging nodes. As expected, our method is faster than highly a matrix-based version from elements of order two and higher, reaching speedups of up to 8× over the highly optimized CUSPARSE library. Compared to a state-of-the-art CPU implementation of the same matrix-free algorithm, our implementation for GPUs is 15% - 61% faster in 2D, and 1.8× - 2.3× faster in 3D. Comparing highly optimized implementations of the same algorithm on GPUs and CPUs of similar power consumption, this suggests that GPUs are about 2 times more power efficient than CPUs for this kind of computations in 3D. While our algorithm for resolving hanging node constraints on the GPU introduces some overhead, this overhead is relatively low, both compared to the overhead of the matrix-free CPU implementation, and in the sense that we still outcompete the CUSPARSE version for elements of order 3 and higher in 2D, and for elements 2 and higher in 3D. Finally, we note that the matrix-free method has the additional benefit of eliminating the matrix assembly, and allowing for solution of problems at least 4 times larger in 2D and 8 times larger in 3D, on a given GPU.

Ongoing work includes implementation of a multigrid linear solver to enable competitive full PDE solution on GPUs. Also, we are working on further generalization to systems of equations allowing for simulation of more general applications, such as fluid flow and structural mechanics. Finally, there is an ongoing effort towards an official inclusion of the GPU implementation in `deal.II`.

ACKNOWLEDGMENTS

The author thanks M. Kronbichler, TU Munich, for valuable discussions and advice. All experiments have been executed on hardware provided by the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

REFERENCES

Bangerth, W., C. Burstedde, T. Heister, and M. Kronbichler. 2011. “Algorithms and data structures for massively parallel generic adaptive finite element codes”. *ACM Trans. Math. Softw.* vol. 38, pp. 14/1–28.

Bangerth, W., D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. 2016. "The deal.II Library, Version 8.4". *Journal of Numerical Mathematics* vol. 24.

Brenner, S. C., and L. R. Scott. 2002. *The mathematical theory of finite element methods*. Springer.

Brown, J. 2010. "Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D". *Journal of Scientific Computing* vol. 45 (1), pp. 48–63.

Cantwell, C. D., S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. 2011. "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements". *Computers & Fluids* vol. 43 (1, SI), pp. 23–28.

Carey, G. F., and B.-N. Jiang. 1986. "Element-by-element linear and nonlinear solution schemes". *Communications in Applied Numerical Methods* vol. 2 (2), pp. 145–153.

Cecka, C., A. J. Lew, and E. Darve. 2011. "Assembly of finite element methods on graphics processors". *International Journal for Numerical Methods in Engineering* vol. 85, pp. 640–669.

Dehnavi, M. M., D. M. Fernandez, and D. Giannacopoulos. 2010, AUG. "Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units". *IEEE Transactions on Magnetics* vol. 46 (8), pp. 2982–2985. 17th International Conference on the Computation of Electromagnetic Fields (COMPUMAG 09), Santa Catarina, Brazil, Nov 22-26, 2009.

Göddeke, D., R. Strzodka, and S. Turek. 2005, Sept. "Accelerating Double Precision FEM Simulations with GPUs". In *Proceedings of ASIM 2005 – 18th Symposium on Simulation Technique*, pp. 139–144.

Gropp, W. D., D. K. Kaushik, D. E. Keyes, and B. F. Smith. 1999. "Towards Realistic Performance Bounds for Implicit CFD codes". In *Proceedings of Parallel CFD'99*, Elsevier.

Klöckner, A., T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. "Nodal discontinuous Galerkin methods on graphics processors". *Journal of Computational Physics* vol. 228 (21), pp. 7863–7882.

Komatitsch, D., G. Erlebacher, D. Göddeke, and D. Michéa. 2010. "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster". *Journal of Computatinal Physics* vol. 229 (20), pp. 7692–7714.

Kormann, K., and M. Kronbichler. 2011. "Parallel Finite Element Operator Application: Graph Partitioning and Coloring". In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pp. 332–339.

Kronbichler, M., and K. Kormann. 2012. "A Generic Interface for Parallel Cell-Based Finite Element Operator Application". *Computers & Fluids* vol. 63 (0), pp. 135–147.

Ljungkvist, K. 2014. "Matrix-Free Finite-Element Operator Application on Graphics Processing Units". In *Euro-Par 2014: Parallel Processing Workshops*, Volume 8806 of *Lecture Notes in Computer Science*, pp. 450–461. Springer International Publishing.

Markall, G. R., A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. 2013, JAN 10. "Finite Element Assembly Strategies on Multi-Core and Many-Core Architectures". *International Journal for Numerical Methods in Fluids* vol. 71 (1), pp. 80–97.

NVIDIA Corporation 2013, July. *CUDA CUSPARSE Library*.

NVIDIA Corporation 2016, September. *NVIDIA CUDA C Programming Guide*. Version 8.0.

Turcksin, B., M. Kronbichler, and W. Bangerth. 2016, August. "*WorkStream* – A Design Pattern for Multicore-Enabled Finite Element Computations". *ACM Trans. Math. Softw.* vol. 43 (1), pp. 2:1–2:29.

AUTHOR BIOGRAPHY

KARL LJUNGKVIST is a PhD Student at Uppsala University. His research interests lie in high-performance computing and development of scientific software. His email address is karl.ljungkvist@it.uu.se.

EFFICIENT ALGORITHMS FOR ASSORTATIVE EDGE SWITCH IN LARGE LABELED NETWORKS

Hasanuzzaman Bhuiyan
Department of Computer Science
Network Dynamics and Simulation Science Laboratory
Biocomplexity Institute of Virginia Tech
Blacksburg, VA, USA
mhb@bi.vt.edu

Maleq Khan
Department of Electrical Engineering
and Computer Science
Texas A&M University—Kingsville
Kingsville, TX, USA
maleq.khan@tamuk.edu

Madhav Marathe
Department of Computer Science
Network Dynamics and Simulation Science Laboratory
Biocomplexity Institute of Virginia Tech
Blacksburg, VA, USA
mmarathe@bi.vt.edu

ABSTRACT

An assortative edge switch is an operation on a labeled network, where two edges are randomly selected and the end vertices are swapped with each other if the labels of the end vertices of the edges remain invariant. Assortative edge switch has important applications in studying the mixing pattern and dynamic behavior of social networks, modeling and analyzing dynamic networks, and generating random networks. In this paper, we present an efficient sequential algorithm and a distributed-memory parallel algorithm for assortative edge switch. To our knowledge, they are the first efficient algorithms for this problem. The dependencies among successive assortative edge switch operations, the requirement of maintaining the assortative coefficient invariant, keeping the network simple, and balancing the computation loads among the processors pose significant challenges in designing a parallel algorithm. Our parallel algorithm achieves a speedup of 68 – 772 with 1024 processors for a wide variety of networks.

Keywords: assortative edge switch, random network generation, network dynamics, parallel algorithms.

1 INTRODUCTION

Networks (or graphs) are simple representations of many complex real-world systems. Analyzing various structural properties of and dynamics on such networks reveal useful insights about the real-world systems (Newman 2002). Assortative edge switch is an important problem in the analysis of such networks and has many real-world applications. An edge switch (or edge swap, edge flip, edge shuffle, edge rewiring) is an operation on a network where two edges are selected randomly and the end vertices are swapped with each other. More formally, it selects two edges (a,b) and (c,d) uniformly at random and replaces them with edges (a,d) and (c,b), respectively. We refer to this operation as the *regular edge switch*. It is easy to see that this operation preserves the degree of each vertex. It is repeated either as many times as required or a specific criterion is satisfied. Edge switch can be used in the generation of random networks with a given

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

degree sequence (Cooper et al. 2007), independent realizations of graphs with a prescribed joint degree distribution using a Markov chain Monte Carlo approach (Ray et al. 2012), modeling and studying various dynamic networks (Feder et al. 2006) and in many other network analytic problems.

Many variations of the edge switch problem (Cooper et al. 2007, Feder et al. 2006, Ray et al. 2012) have been studied. In this paper, we present efficient sequential and parallel algorithms for such a variant, referred to as the *assortative edge switch*, which is an operation on a labeled network, where each vertex u has an associated label $L(u)$. Such labels can be discrete characteristics (e.g., language, race, and gender in social networks) or scalar properties (e.g., age and degree) of the vertices. An assortative edge switch operation selects two edges (a,b) and (c,d) randomly, and replaces them with edges (a,d) and (c,b), respectively, if $L(a) = L(c)$ and $L(b) = L(d)$. For each vertex u, the degree, as well as the distribution of the labels of the adjacent vertices of u, remains invariant under an assortative edge switch process.

An assortative edge switch operation preserves the *assortative mixing* of a given network, which is a fundamental network feature measuring the tendency of the vertices to associate with similar or dissimilar vertices and is quantified by a metric named the *assortative coefficient* (Newman 2003). In other words, assortativity measures the correlation of vertices based on the vertex labels. Newman (2002) showed that assortative mixing is a pervasive phenomenon found in many real-world networks and has a profound impact on the structural properties and functionalities of the networks. For instance, social (e.g., co-authorship) networks exhibit positive assortativity—more association among the similar types of vertices—whereas technological (e.g., Internet and WWW) and biological (e.g., food-web) networks show negative assortativity—more association among the dissimilar types of vertices (Newman 2002). Assortative edge switch can be used to assess and analyze the sensitivity of mixing patterns and network structural properties on dynamics over a network, such as disease dynamics over a social contact network (Eubank et al. 2010). Random network models often do not capture many structural properties (e.g., assortative mixing) of real-world networks; as a result, to be more realistic, modeling random networks with a prescribed assortative coefficient has gained popularity in the research community (Milo et al. 2003). Assortative edge switch can be combined with regular edge switch process to generate random networks with a prescribed assortative coefficient from a given network (Xulvi-Brunet and Sokolov 2004).

The recent growth of real-world data due to the rapid progression of science and technology poses significant challenges towards efficient processing of massive networks. Dealing with these huge amounts of data efficiently and effectively motivates the need for parallel computing. NetworkX (Hagberg et al. 2008) has a sequential implementation of regular edge switch; however, it does not have an implementation of assortative edge switch. A parallel algorithm for regular edge switch has been presented in (Bhuiyan et al. 2014). However, no effort was given to design a parallel algorithm for assortative edge switch in a network. Although the algorithm by Bhuiyan et al. (2014) can be applied to perform assortative edge switch operations in a network, it can lead to a slow and inefficient algorithm. To perform an edge switch operation in (Bhuiyan et al. 2014), the edges are selected randomly from the entire network, irrespective of the vertex labels, which can result in many failed attempts for an assortative edge switch operation due to dissatisfying the constraint on the vertex labels. As a result, we need a completely new and efficient algorithmic approach.

Our Contributions. In this paper, we first present a sequential algorithm for assortative edge switch; then we present a parallel algorithm based on our sequential algorithm. The dependencies among successive assortative edge switch operations and the requirement of keeping the network simple as well as maintaining the same assortativity during the assortative edge switch process pose significant challenges in designing a parallel algorithm. Moreover, achieving a good speedup through a well-balanced load distribution among the processors seems to be a non-trivial challenge for this problem. For various network size and diverse distribution of the labels, the parallel algorithm provides speedup ranging between 68 and 772 with 1024 processors. To our knowledge, they are the first efficient sequential and parallel algorithms for the problem.

Organization. The remainder of the paper is organized as follows. The preliminaries and datasets used in the paper are briefly described in Section 2. The problem of assortative edge switch and the sequential algorithm are discussed in Section 3. We present the parallel algorithm along with the performance analysis in Section 4. Finally, we conclude in Section 5.

2 PRELIMINARIES AND DATASETS

Below are the notations, definitions, datasets, and computation model used in this paper.

2.1 Notations and Definitions

We are given a simple, labeled network $G = (V, E, L)$, where V is the set of vertices, E is the set of edges, and $L : V \to \mathbb{N}_0$ is the label function. A *simple network* is an undirected network having no self-loops or parallel edges. A *self-loop* is an edge from a vertex to itself. *Parallel edges* are two or more edges connecting the same pair of vertices. There are a total $n = |V|$ vertices with vertex ids $0, 1, 2, \ldots, n-1$ and $m = |E|$ edges in G. Each vertex $u \in V$ has an associated label $L(u)$. There are a total of ℓ distinct vertex labels with label ids $0, 1, 2, \ldots, \ell - 1$, i.e., for each $u \in V$, $L(u) \in [0, \ell - 1]$. For an edge $(u, v) \in E$, we say u and v are *neighbors* of each other. The *adjacency list* of a vertex u contains all the neighbors of u and is denoted by $N(u)$, i.e., $N(u) = \{v \in V | (u, v) \in E\}$. The *degree* of u is $d_u = |N(u)|$. We use K, M, and B to denote thousands, millions, and billions, respectively; e.g., $1B$ stands for one billion. For the parallel algorithm, let there be P processors with ranks $0, 1, 2, \ldots, P-1$, where P_i denotes the processor with rank i.

Definition 1. *An* **edge switch** *operation selects two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ randomly from E and replaces them with new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$, if the resultant network remains simple.*

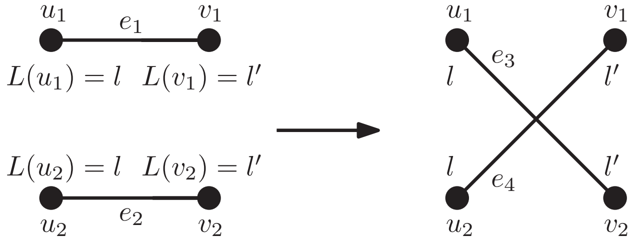


Figure 1: An assortative edge switch operation replaces two randomly selected edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$, if $L(u_1) = L(u_2)$ and $L(v_1) = L(v_2)$.

Definition 2. *An* **assortative edge switch** *operation imposes an extra constraint on the labels of the end vertices of the two selected edges in addition to the regular edge switch constraints. That is, it randomly selects two edges $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ from E, and replaces them by new edges $e_3 = (u_1, v_2)$ and $e_4 = (u_2, v_1)$ (see Figure 1), if the following constraints are satisfied.*

- **Simple network:** *It does not create self-loops or parallel edges, i.e., $u_1 \neq v_2$, $u_2 \neq v_1$, $u_1 \notin N(v_2)$, $v_2 \notin N(u_1)$, $u_2 \notin N(v_1)$, and $v_1 \notin N(u_2)$.*
- **Vertex labels:** *The two edges have the same end vertex labels, i.e., $L(u_1) = L(u_2)$, $L(v_1) = L(v_2)$.*

Definition 3. *Some edges of the given network G are changed (or visited) due to assortative edge switch operations, and some edges do not participate in any such operations and remain unchanged (or unvisited). We define the* **visit rate** *(x) as the fraction of edges of G that have been changed by a sequence of assortative edge switch operations, i.e., $x = \frac{m'}{m}$, where m' is the number of edges changed due to assortative edge switches. To achieve a given visit rate x, the expected number of assortative edge switch operations τ to be performed in G is $\tau = \frac{1}{2} m \ln m$ for $x = 1$, and $\tau = -\frac{1}{2} m \ln(1 - x)$ for $0 < x < 1$ (Bhuiyan et al. 2014).*

Table 1: Datasets used in the experiments. K, M, and B denote thousands, millions, and billions, respectively.

Network	**Type of network**	**Vertices**	**Edges**	**Bins**		**Assort. edge switch**
				Deg. assort.	**Age assort.**	**(Visit rate = 1.0)**
New York	Social Contact	17.88**M**	480.1**M**	85.7**K**	4186	4.80**B**
Los Angeles	Social Contact	16.23**M**	459.3**M**	83.1**K**	4186	4.58**B**
Miami	Social Contact	2.09**M**	51.50**M**	66.1**K**	4186	457.2**M**
Sweden	Social Contact	9.46**M**	406.2**M**	83.2**K**	-	4.03**B**
Orkut	Social	3.07**M**	117.2**M**	2.46**M**	-	1.09**B**
Facebook	Social	3.10**M**	23.67**M**	378**K**	-	200.9**M**
LiveJournal	Social	4.80**M**	42.85**M**	802**K**	-	376.5**M**
ErdosRenyi	Erdős-Rényi	1.00**M**	500.00**M**	36.62**K**	-	5.0**B**
SmallWorld	Small World	4.80**M**	48.00**M**	161	-	424.5**M**
PA	Pref. Attach.	100.0**M**	1.00**B**	1.58**M**	-	10.36**B**

2.2 Datasets and Computation Model

We use both artificial and real-world networks for the experiments. A summary of the networks is given in Table 1. New York (NY), Los Angeles (LA), Miami, and Sweden are synthetic, yet realistic social contact networks (Barrett et al. 2009). Each vertex represents a person in the corresponding city or country and has an associated label denoting the age of the individual, and each edge represents any 'physical' contact between two persons within a 24 hour time period. Orkut is an online social network, Facebook is an anonymized Facebook friendship network, and LiveJournal is a social network blogging site (Abdelhamid et al. 2012). The SmallWorld, ErdosRenyi, and PA networks are random networks generated using the Watts-Strogatz small world network (Watts and Strogatz 1998), Erdős-Rényi network (Bollobás 1998), and Preferential Attachment network (Barabási and Albert 1999) models, respectively. For all the networks, we perform *degree-assortative* edge switch operations, where the degree of each vertex is considered as its label. In addition, we perform *age-assortative* edge switch operations on the social contact networks of Miami, NY, and LA, where each person's age is considered as its label. The age information for the other networks is either inappropriate or unavailable.

We develop algorithms for distributed memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory and can communicate with each other by exchanging messages.

3 A SEQUENTIAL ALGORITHM

We are given a network $G = (V,E,L)$ and the number of assortative edge switch operations τ to be performed. A naïve approach to perform an assortative edge switch operation is selecting two edges (u_1, v_1) and (u_2, v_2) uniformly at random from E and swapping the end vertices of the edges if the constraints are satisfied. If any of the constraints is not satisfied, the selected pair of edges is discarded and a new pair is selected. For a large and relatively sparse network, the number of such discarded attempts (or pairs of edges) due to dissatisfying the constraint of keeping the network simple is almost negligible; however, for the constraint on the vertex labels, the number of discarded attempts can be very large, as shown in Table 2. For example, to perform 10K degree-assortative edge switch operations on the LA network, the numbers of discarded attempts due to the constraints on the simple network and vertex labels are 1 and 179.4M, respectively. Assume that there are $\ell = 100$ different labels uniformly distributed among the vertices of a given network, i.e., for any label i, the number of vertices with label i is $\frac{n}{100}$. Then the probability of randomly selecting two edges (u_1, v_1) and (u_2, v_2) satisfying the constraint on the vertex labels is $(\frac{1}{100})^2$, which is very

Table 2: Number of discarded attempts (due to dissatisfying the two constraints) to perform 10K age- and degree-assortative edge switch operations on different networks.

Constraints	**Miami-Age**	**LA-Deg**	**Facebook**	**LiveJournal**	**ErdosRenyi**
Simple network	2	1	148	34	23
Vertex labels	30.3**M**	179.4**M**	617.2**M**	165.2**M**	125.4**M**

small. As a result, the number of discarded attempts can be very large, and it can make the algorithm slow. To deal with this difficulty, we present an efficient sequential algorithm using a new and efficient algorithmic approach.

3.1 An Efficient Sequential Algorithm

Let us denote a *bin* Z_{ij} to be the set of all edges having the same end vertex labels (i,j), where $0 \leq i,j \leq \ell - 1$, i.e., for an edge $(u,v) \in E$, if $L(u) = i$ and $L(v) = j$, then $(u,v) \in Z_{ij}$. For an undirected network, $Z_{ij} = Z_{ji}$, and we use Z_{ij} such that $j \leq i$. Note that the bins are disjoint and $\bigcup_{j \leq i} Z_{ij} = E$. For convenience, we relabel the bins from two indices (i,j) to a single bin number k. Let b_k be the new label of the bin Z_{ij}, where $k = \frac{i(i+1)}{2} + j$. Assume that there are Y such bins and let us denote them as $b_0, b_1, \ldots, b_{Y-1}$; there can be at most $Y = \binom{\ell}{2} + \ell = O(\ell^2)$ such bins. The *size* of a bin b_i is the number of edges in b_i and is denoted by m_i, i.e., $m_i = |b_i|$. Then the number of possible pairs of edges in b_i is $a_i = {m_i}^2$. Note that the set of edges in a bin b_i changes dynamically during the course of an assortative edge switch process, although m_i remains invariant throughout the process.

First, the algorithm constructs the bins b_i from G. Then an assortative edge switch operation is performed as follows: (*i*) a bin b_i is chosen randomly, where the probability of selecting b_i is $q_i = \frac{a_i}{\sum_{j=0}^{Y-1} a_j}$, (*ii*) a pair of edges is selected uniformly at random from b_i, and (*iii*) the end vertices of the edges are swapped with each other. This operation is repeated until τ pairs of edges are switched. Note that this algorithm guarantees that both of the edges for an assortative edge switch operation are always selected from the same bin irrespective of how many bins there are, thus overcoming the drawback of the naïve approach. A pseudocode of the algorithm is given in Figure 2.

```
1: **Partition** $E$ into minimum number of disjoint bins $b_0, b_1, \ldots, b_{Y-1}$, where for any $i$ and for any
   pair of edges $(u_1,v_1),(u_2,v_2) \in b_i$, $L(u_1) = L(u_2)$ and $L(v_1) = L(v_2)$.
2: **for** $k = 1$ to $\tau$ **do**
3:     $b_i \leftarrow$ a random bin in $[b_0, b_{Y-1}]$ with a probability of $q_i = \frac{a_i}{\sum_{j=0}^{Y-1} a_j}$
4:     $(u_1,v_1),(u_2,v_2) \leftarrow$ two uniform random edges in $b_i$
5:     **if** $u_1 = v_2$, $u_2 = v_1$, $u_1 \in N(v_2)$, or $u_2 \in N(v_1)$ **then**
6:         continue
7:     **Replace** $(u_1,v_1)$ and $(u_2,v_2)$ by $(u_1,v_2)$ and $(u_2,v_1)$
```

Figure 2: A sequential algorithm for assortative edge switch.

Theorem 1. *The time complexity of the sequential algorithm is $O(m + Y + \tau \log d_{max})$.*

Proof. Partitioning the set of edges E into the Y bins (line 1 in Figure 2) takes $O(m+Y)$ time as initializing the bins takes $O(Y)$ time and the bins are constructed from E in $O(m)$ time. The adjacency list of a vertex u can be maintained using a balanced binary tree, and searching (for parallel edges) and updating such a tree takes $O(\log d_u)$ time. Hence, an assortative edge switch operation (lines $3-7$) can be performed

Table 3: Runtime comparison of our sequential algorithm and the naïve approach on various networks. Experiments performed with a visit rate of 0.01.

Network	**Runtime (min.)**		**Faster by a**
	Naïve approach	**Our algo.**	**factor of**
Miami-Age	8.8	0.14	62.86
LA-Deg	929	1.88	494.1
Facebook	69.1	0.08	863.75
LiveJournal	48.87	0.15	325.8
ErdosRenyi	342.5	2.22	154.3

in $O(\log d_{max})$ time, where $d_{max} = \max_u d_u$ and τ such operations (lines $2-7$) take $O(\tau \log d_{max})$ time. Therefore, the time complexity of the algorithm is $O(m+Y+\tau \log d_{max})$. ■

Theorem 2. *The space complexity of the sequential algorithm is* $O(m+Y)$.

Proof. Storing all the m edges into the Y bins takes $O(m+Y)$ space. ■

3.2 Performance Evaluation of the Sequential Algorithm

Table 3 demonstrates the runtime comparison of our algorithm with the naïve approach. We use current calendar time as a random seed and a visit rate of 0.01 for the experiments since the naïve approach takes a large amount of time with a visit rate of 1.0. Our algorithm shows very good overall performance, e.g., for degree-assortative edge switch on the LA network, our algorithm is 494 times faster than the naïve approach.

4 THE PARALLEL ALGORITHM

A parallel algorithm for regular edge switch is presented in (Bhuiyan et al. 2014); however, this algorithm can be very slow for assortative edge switch for the same reasons as explained in Section 3. In this section, we present a novel parallel algorithm for assortative edge switch, which is based on our sequential algorithm, as shown in Figure 2. Recall that the sequential algorithm selects both of the edges for an assortative edge switch operation from the same bin. Hence, assortative edge switch operations in a bin are independent of the other bins. The parallel algorithm exploits this property to perform simultaneous assortative edge switch operations in parallel in different bins. The bins are distributed among the processors such that the computation load distribution is well-balanced. If a bin is very large compared to the other bins, the algorithm might need to partition and distribute the bin among multiple processors, which is discussed later in Section 4.3. For now, assume that each bin is entirely assigned to a single processor.

4.1 The Parallel Algorithm with Each Bin Assigned to a Single Processor

The parallel algorithm should distribute the bins to processors such that the computation cost is equally distributed among the processors. Therefore, we need to estimate the computation cost associated with each bin, which is the number of assortative edge switch operations performed in a bin b_i. It raises the question of how many assortative edge switch operations among the total τ operations are performed in b_i? Let us denote X_i be the number of assortative edge switch operations performed in b_i by the sequential algorithm. Recall that to perform an assortative edge switch operation, the sequential algorithm randomly selects a bin b_i, and then chooses two edges randomly from b_i. Hence, a sequential algorithm does not need to know X_i in advance. However, in the parallel algorithm, all of the processors perform simultaneous assortative edge switch operations in parallel and different processors may need to work on different bins at the same time. As a result, for each i, X_i needs to be determined in advance for the parallel algorithm. It is easy to see that

```
for $i = 0$ to $P-1$ do
    $W_i \leftarrow 0$
    $B_i \leftarrow \emptyset$
Sort the bins in non-increasing order of $X_j$
Assume that $X_0 \geq X_1 \geq \ldots \geq X_{Y-1}$
for $j = 0$ to $Y-1$ do
    Let $P_i$ be the processor with rank $i = \arg\min_k W_k$
    $B_i \leftarrow B_i \bigcup \{b_j\}$
    $W_i \leftarrow W_i + X_j$
```

Figure 3: A load balancing algorithm assigning the Y bins to the P processors.

```
for each bin $b_j \in B_i$ do
    for $k = 1$ to $X_j$ do
        $(u_1, v_1), (u_2, v_2) \leftarrow$ two uniform random edges in $b_j$
        if $u_1 = v_2$, $u_2 = v_1$, $u_1 \in N(v_2)$, or $u_2 \in N(v_1)$ then
            continue
        Replace $(u_1, v_1)$ and $(u_2, v_2)$ by $(u_1, v_2)$ and $(u_2, v_1)$
```

Figure 4: A processor P_i performing assortative edge switch operations in the parallel algorithm.

the random variables X_i are the multinomial random variables generated by a multinomial distribution with parameters $(\tau, q_0, q_1, \ldots, q_{Y-1})$, that is,

$$\langle X_0, X_1, \ldots, X_{Y-1} \rangle \sim M(\tau, q_0, q_1, \ldots, q_{Y-1}) \tag{1}$$

where τ is the number of assortative edge switch operations and $q_i = \frac{a_i}{\sum_{j=0}^{Y-1} a_j}$ is the probability of selecting a bin b_i. The time complexity of the best known sequential algorithm, known as the *conditional distributed method* (Davis 1993), for generating multinomial random variables is $\Theta(\tau)$. To have an efficient parallel algorithm for assortative edge switch, we need a parallel algorithm for generating multinomial random variables. We use the algorithm presented in (Bhuiyan et al. 2014), which has a runtime of $O\left(\frac{\tau}{P} + Y \log P\right)$. Each processor independently computes the multinomial distribution of $\frac{\tau}{P}$ and then the results are aggregated by exploiting a property of the multinomial distribution. An overview of the parallel algorithm is as follows:

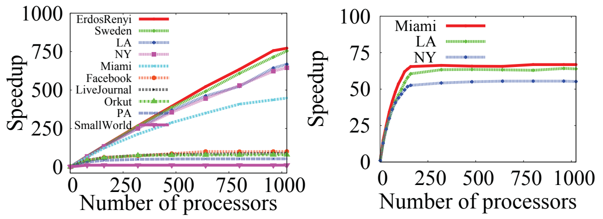
- ***Step 1:*** Generate multinomial random variables X_i in parallel with parameter $(\tau, q_0, q_1, \ldots, q_{Y-1})$ to estimate the computation cost associated with each bin.
- ***Step 2:*** Using the estimated costs X_i, partition the bins $b_0, b_1, \ldots, b_{Y-1}$ among the P processors such that the computation load distribution is well-balanced.
- ***Step 3:*** Each processor P_i simultaneously performs assortative edge switch operations in parallel in the bins assigned to it.

Now we describe the last two steps of the algorithm.

Partitioning (Step 2). Let B_i be the set of bins, $Y_i = |B_i|$ be the number of bins and $M_i = \sum_{b_j \in B_i} m_j$ be the number of edges assigned to a processor P_i. Then the *workload* (or *load*) W_i in P_i is the summation of the computation costs associated with the bins in B_i, i.e., $W_i = \sum_{b_j \in B_i} X_j$. Let W be the *maximum load* among all of the processors, i.e., $W = \max_i W_i$. Now the goal is to distribute the Y bins among the P processors such that the maximum load W is minimized. Finding an assignment of the bins for the optimum solution is an NP-hard problem (Kleinberg and Tardos 2006). The best known approximation algorithm for this problem is presented in (Kleinberg and Tardos 2006), which has an approximation ratio of 1.5. This greedy algorithm sorts the bins in non-increasing order of the loads. To do so, we use a parallel version of quick sort (Grama 2003). Then the algorithm makes one pass over the sorted bins to assign each bin to a processor having the minimum load at the time of the assignment, as shown in Figure 3.

Switching Edges (Step 3). Each processor P_i constructs the bins in B_i and then simultaneously performs W_i assortative edge switch operations in parallel, as shown in Figure 4. The program terminates when all of the processors complete their assortative edge switch operations.

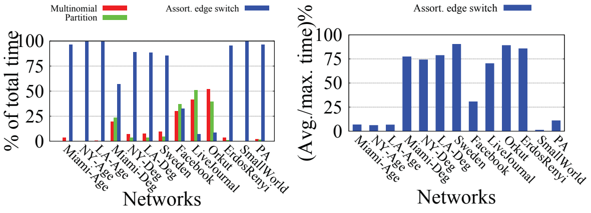
Theorem 3. *The time complexity in each processor P_i is $O(\frac{\tau}{P} + Y \log P + \frac{Y}{P} \log Y + \log^2 P + P + Y_i + M_i + W_i \log d_{max})$.*



(a) Degree-assortative

(b) Age-assortative

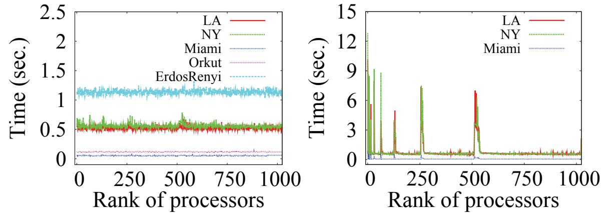
Figure 5: Strong scaling of the parallel algorithm.



(a) Time taken by individual steps of the algorithm

(b) Ratio of the avg. and max. time in the third step

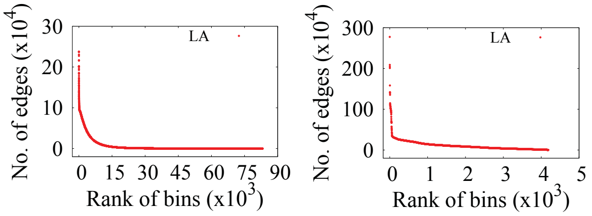
Figure 6: Time at different steps of the algorithm.



(a) Degree-assortative

(b) Age-assortative

Figure 7: Execution time of the individual processors in the third step of the algorithm.



(a) Degree-assortative

(b) Age-assortative

Figure 8: Distribution of edges among the bins for the LA network.

Proof. In step 1, multinomial random variables are generated in parallel in $O\left(\frac{\tau}{P}+Y\log P\right)$ time. In step 2, the parallel version of quick sort and the assignment of the bins take $O\left(\frac{Y}{P}\log Y+\log^2 P\right)$ and $O(P+Y\log P)$ time, respectively. In step 3, each processor P_i constructs the Y_i bins in $O(Y_i+M_i)$ time and then performs the assortative edge switch operations in $O(W_i\log d_{max})$ time. ■

Theorem 4. *The space complexity in each processor P_i is $O(Y_i+M_i)$.*

Proof. Each processor P_i stores all the M_i edges in Y_i bins using $O(Y_i+M_i)$ space. ■

4.2 Performance Analysis of the Parallel Algorithm with Each Bin Assigned to a Single Processor

In this section, we analyze the performance of the parallel algorithm. Table 1 provides a summary of the number of bins and assortative edge switch operations performed on the ten networks. We use a HPC cluster consisting of 64 compute nodes. Each node has a dual-socket Intel Sandy Bridge E5-2670 2.60GHz 8-core processor (16 cores per node) with 64GB memory. The algorithms are developed with MPICH2 (v1.9), optimized for Qlogic QDR Infiniband cards.

4.2.1 Strong Scaling

Figure 5a and 5b illustrate the strong scaling performance for degree- and age-assortative edge switch, respectively, on various networks. The algorithm achieves a speedup between 12 and 772 with 1024 processors. For some networks, the speedup is poor compared to the other networks. Next, we investigate the load distribution among the processors to understand the reason for this poor performance.

4.2.2 Load Distribution

First, we measure the time taken by each of the three steps (step 1: multinomial, step 2: partitioning, step 3: assortative edge switch) of the algorithm with 1024 processors, as shown in Figure 6a. The assortative edge switch step takes the largest amount of time among the three steps for all the networks, except Facebook, LiveJournal, and Orkut, in which, the number of bins Y is significantly higher than that of the other networks (see Table 1). Therefore, generating the multinomial random variables and partitioning the bins take more time for these three networks. Unlike the first two steps, where every processor takes almost an equal amount of time, the execution time of the individual processors in the third step can vary significantly because of a poor load distribution. Figure 6b demonstrates the ratio of T_{avg} and T_{max}, where T_{avg} is the average execution time of the processors in the assortative edge switch step and T_{max} is the maximum time among them. The maximum value of the ratio is 100% for a perfectly-balanced load distribution in the ideal case. A higher value implies a well-balanced load distribution, whereas a lower value indicates a poor load distribution among the processors. We observe a well-balanced load distribution for degree-assortative edge switch on the Miami, NY, LA, Sweden, LiveJournal, Orkut, and ErdosRenyi networks. In contrast, we observe a poor load distribution for age-assortative edge switch on the Miami, NY and LA networks and degree-assortative edge switch on the PA and SmallWorld networks. For the PA network, the poor load distribution in the third step causes the step to take the largest amount of time despite having a large number of bins, which is in contrast to the scenario for the Facebook, LiveJournal, and Orkut networks. For the Facebook network, a few processors contain many small-size bins (with a few edges) and the number of discarded attempts in these bins are very high, yielding a larger execution time in these processors (hence, T_{max} is large). As a result, we observe a low ratio despite each processor performing roughly an equal number of assortative edge switch operations and the low ratio is not a consequence of load balancing. The observations are further supported by Figures 7a and 7b, which show the individual execution time of the processors in the third step of the algorithm. For a better understanding, we analyze the load distribution in detail for the LA network.

Figures 8a and 8b illustrate the distribution of the sizes of the bins for degree- and age-assortative edge switch, respectively, on the LA network. The distribution for age-assortative edge switch is highly skewed, having a few very large bins and many small bins, which in turn makes a poor load distribution among the processors, as shown in Figure 7b. A few processors containing the larger bins are taking significantly more time than the processors containing the smaller bins, which results in a low speedup. On the other hand, for degree-assortative edge switch, there is a good number of both larger and smaller bins among the total 83.1K bins and the differences between the larger and smaller bins are substantially smaller than that of the age-assortative counterpart. The algorithm assigns a few larger bins along with many smaller bins to each processor and consequently exhibits a well-balanced load distribution, as shown in Figure 7a, thus resulting in a good speedup. Note that if the number of bins is less than the number of processors, i.e., $Y < P$, then some processors remain idle in the third step of the computation; and we observe this phenomenon for the SmallWorld network. To deal with the poor load distribution, we present a parallel algorithm with an improved load balancing scheme in the next section.

4.3 The Parallel Algorithm with Improved Load Balancing

As we discussed earlier, some bins can have higher computation costs, i.e., the number of assortative edge switch operations X_i performed in a bin b_i can be significantly larger than $\frac{\tau}{P}$, which can cause a poor load distribution. For a better load balancing, such a bin may need to be partitioned and distributed among multiple processors. Let Δ be some *threshold* such that Δ is larger than $\frac{\tau}{P}$. We call a bin *large* if $X_i \geq \Delta$, and *small*, otherwise. Let Q be the total number of processors assigned for the large bins. First, we explain how to perform assortative edge switch operations in a single large bin with multiple processors.

Assume that a large bin b_i is partitioned among the processors $P_x, P_{x+1}, \ldots, P_y$, where $x \leq j,k,l \leq y$. The bin b_i is partitioned such that a subset of vertices, having consecutive vertex ids, along with their incident edges in b_i are assigned to a partition and each such partition contains almost an equal number of edges. A vertex u's *partial* adjacency list in b_i, i.e., $N_i(u) = \{v \in V | (u,v) \in b_i\}$, entirely belongs to a unique partition. Then each processor P_j performs simultaneous assortative edge switch operations in parallel. Assortative edge switch in a bin is similar to the regular edge switch because the end vertices of the edges in a bin have the same labels. A parallel algorithm for regular edge switch has been presented in (Bhuiyan et al. 2014). We use this algorithm to switch the edges in a large bin. A summary of an assortative edge switch operation performed by a processor P_j is as follows. P_j selects an edge (u_1, v_1) randomly from its own partition. The other edge (u_2, v_2) is chosen in two steps: (*i*) P_j selects a processor P_k with a probability proportional to the number of edges belonging to P_k, and (*ii*) P_j requests P_k to select (u_2, v_2) randomly from its partition. Then P_j and P_k work together to check whether switching the edges (u_1, v_1) and (u_2, v_2) creates any loop or parallel edge. If it creates any loop or parallel edge, the selected pair of edges is discarded, and a new pair is chosen by P_j. Otherwise, P_j and P_k work together to update (u_1, v_1) and (u_2, v_2) by (u_1, v_2) and (u_2, v_1), respectively. In fact, P_j and P_k may require updating an edge in another processor P_l ($P_j \neq P_l \neq P_k$). The details can be found in (Bhuiyan et al. 2014).

Now, we discuss how to determine Δ and Q. Apparently, it seems that any bin b_i with $X_i > \frac{\tau}{P}$ needs to be partitioned and distributed among multiple processors. However, partitioning a bin incurs communication and synchronization overhead due to the need for exchanging messages among multiple processors even for a single edge switch operation. Thus, we partition a bin among multiple processors only when X_i is significantly larger than $\frac{\tau}{P}$, i.e., when the gain achieved by partitioning a bin is larger than the communication and synchronization cost incurred by partitioning the bin. We assume $\Delta = \alpha \times \frac{\tau}{P}$ for some constant $\alpha > 1$. Similarly, to perform X_i assortative edge switch operations in a large bin b_i, we should ideally assign $\lceil \frac{X_i}{\tau/P} \rceil$ processors for b_i. However, due to communication overhead, we need to assign a larger number of processors. Hence, we assign $Q_i = \lceil \beta \times X_i \times \frac{P}{\tau} \rceil$ processors for a large bin b_i for some constant $\beta > 1$. Then $Q = \sum_i Q_i$ and the number of processors assigned for the small bins is $S = P - Q$. The performance of the algorithm greatly depends on Δ and Q, thusly on α and β. We experimented with many different types of networks and find that for $\alpha \in [2.4, 2.75]$ and $\beta \in [12, 15]$, the algorithm exhibits good performance, which is very close to the optimal performance, as shown in the next section.

4.4 Performance Analysis of the Parallel Algorithm with Improved Load Balancing

Figure 9 shows the strong scaling performance of the parallel algorithm and Figure 10 demonstrates a comparison of the speedup improvement. The algorithm achieves a speedup of 277 for age-assortative edge switch on the Miami network with 1024 processors, which is a four-fold improvement achieved by a well-balanced load distribution among the processors, as shown in Figure 11. Next, we analyze the effect of α and β (thus Δ and Q) on the speedup.

Figure 12 shows how the speedup varies with different values of α and β for age-assortative edge switch on the Miami network with 1024 processors. For a fixed value of β (say $\beta = 14$) and with the increase of α, more large-size bins are getting partitioned among the same number of processors, yielding a speedup increase up to some value of α, referred to as *optimal* α (optimal $\alpha = 2.5$ for $\beta = 14$), beyond which the speedup starts decreasing because of the increase of communication and synchronization overhead incurred by the increasing number of bins partitioned. We observe a similar pattern for a fixed value of α and with the increase of β as well. For lower values of β, less number of processors are working on the large bins. As a result, the execution times of the Q processors are higher than that of the S processors working on the small bins. This is further illustrated in Figure 13, which shows the execution time of the individual processors with varying β and a fixed $\alpha = 2.5$. We observe a pattern of many horizontal flat segments, where each horizontal segment is the time taken by the processors working on the same large bin. With the increase of β, thusly Q, the amount of work for each of the Q processors decreases, whereas the times taken by the S processors increase as fewer processors are dealing with the small bins. The highest speedup is achieved when the maximum execution time among the Q processors is somewhat balanced with that of the S processors. We observe similar phenomena for the other networks as well and recommend using a $\alpha \in [2.4, 2.75]$ and $\beta \in [12, 15]$, for which the algorithm achieves a good speedup.

In principle, the parallel algorithm is designed such that it stochastically produces the same effect (by using the multinomial distribution) on a network as the sequential algorithm would do. We also experimentally verify this by showing that the average clustering coefficient and the average shortest path distance of a network change similarly with assortative edge switches by the sequential and parallel algorithms (see Fig-

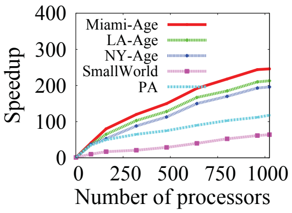


Figure 9: Strong scaling of the parallel algorithm with improved load balancing.

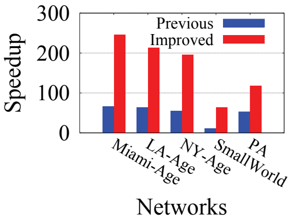


Figure 10: A comparison of speedup improvement with 1024 processors.

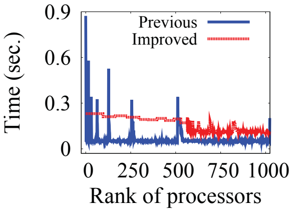


Figure 11: A comparison of runtime of the individual processors for Miami-Age.

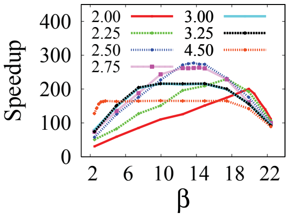


Figure 12: Speedup with increasing β for Miami-Age using different values of α.

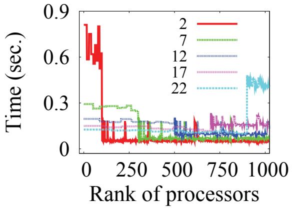


Figure 13: Runtime of the individual processors with different values of β for Miami-Age using a fixed $\alpha = 2.5$.

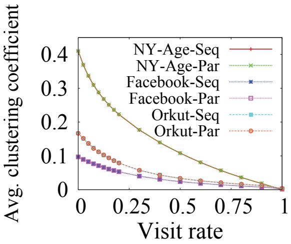


Figure 14: Average clustering coefficient changes similarly with assort. edge switches by the sequential and parallel algorithms.

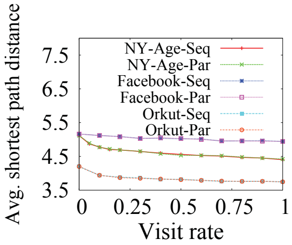


Figure 15: Average shortest path distance changes similarly with assort. edge switches by the sequential and parallel algorithms.

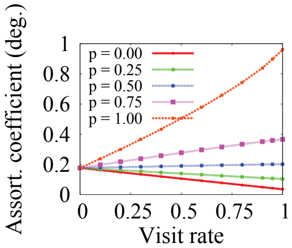


Figure 16: Varying degree-assort. coefficient by age-assort. edge switch on the Miami network through a parameter $p \in [0,1]$.

ure 14 and 15). We use the NY, Orkut, and Facebook networks and vary the visit rate from 0.025 to 1. For both properties, the changes by the sequential and parallel algorithms are very similar; in fact, they overlap with each other, and it is difficult to distinguish them in the figures. We also demonstrate how assortative edge switch can be used to generate random networks by keeping one assortative coefficient A_1 invariant and varying another assortative coefficient A_2 to a desired level through a parameter p ($0 \le p \le 1$). Xulvi-Brunet and Sokolov (2004) proposed one such algorithm, where with probability p, an edge switch operation connects the two higher degree vertices with an edge and the two lower degree vertices with another edge. With probability $(1-p)$, the edges are switched randomly. Figure 16 shows how the degree assortative coefficient changes for different value of p with the age-assortative edge switch process on the Miami network.

5 CONCLUSION

We have developed efficient sequential and parallel algorithms for assortative edge switch in massive networks. They can be used to study the sensitivity of network topology on the dynamics over a network as well as to generate network perturbations of a given network by maintaining the same degree sequence and assortative coefficient.

ACKNOWLEDGMENTS

This work has been partially supported by DTRA CNIMS Contract HDTRA1-11-D-0016-0001, DTRA Grant HDTRA1-11-1-0016, NSF NetSE Grant CNS-1011769, NSF SDCI Grant OCI-1032677, NIH MIDAS Grant 5U01GM070694-11, NSF DIBBs Grant ACI-1443054, and NSF Big Data Grant IIS- 1633028.

REFERENCES

Abdelhamid, S. E., R. Alo, S. Arifuzzaman, P. Beckman, M. H. Bhuiyan et al. 2012. "CINET: A cyberinfrastructure for network science". In *Proceedings of the 8th International Conference on E-Science (e-Science)*, pp. 1–8.

Barabási, A.-L., and R. Albert. 1999. "Emergence of scaling in random networks". *Science* vol. 286 (5439), pp. 509–512.

Barrett, C. L., R. J. Beckman, M. Khan et al. 2009. "Generation and analysis of large synthetic social contact networks". In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pp. 1003–1014.

Bhuiyan, H., J. Chen, M. Khan, and M. V. Marathe. 2014. "Fast parallel algorithms for edge-switching to achieve a target visit rate in heterogeneous graphs". In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)*, pp. 60–69.

Bollobás, B. 1998. "Random graphs". In *Modern Graph Theory*, pp. 215–252. Springer.

Cooper, C., M. Dyer, and C. Greenhill. 2007. "Sampling regular graphs and a peer-to-peer network". *Combinatorics, Probability and Computing* vol. 16 (4), pp. 557–593.

Davis, C. S. 1993. "The computer generation of multinomial random variates". *Computational Statistics & Data Analytics* vol. 16 (2), pp. 205–217.

Eubank, S., A. Vullikanti, M. Khan et al. 2010. "Beyond degree distributions: Local to global structure of social contact graphs". In *Proceedings of the Third International Conference on Social Computing, Behavioral Modeling, and Prediction*, pp. 1–1.

Feder, T., A. Guetz, M. Mihail, and A. Saberi. 2006. "A local switch Markov chain on given degree graphs with application in connectivity of peer-to-peer networks". In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 69–76.

Grama, A. 2003. *Introduction to parallel computing*. Pearson Education.

Hagberg, A., P. Swart, and D. Schult. 2008. "Exploring network structure, dynamics, and function using NetworkX". In *Proceedings of the 7th Python in Science Conference (SciPy)*, pp. 11–15.

Kleinberg, J., and É. Tardos. 2006. *Algorithm design*. Pearson Education.

Milo, R., N. Kashtan, S. Itzkovitz, M. E. Newman, and U. Alon. 2003. "On the uniform generation of random graphs with prescribed degree sequences". *arXiv preprint cond-mat/0312028*.

Newman, M. E. 2002. "Assortative mixing in networks". *Physical Review Letters* vol. 89 (20), pp. 208701.

Newman, M. E. 2003. "Mixing patterns in networks". *Physical Review E* vol. 67 (2), pp. 026126.

Ray, J., A. Pinar, and C. Seshadhri. 2012. "Are we there yet? When to stop a Markov chain while generating random graphs". In *Proceedings of the 9th Workshop on Algorithms and Models for the Web Graph (WAW)*, pp. 153–164.

Watts, D. J., and S. H. Strogatz. 1998. "Collective dynamics of 'small-world' networks". *Nature* vol. 393 (6684), pp. 440–442.

Xulvi-Brunet, R., and I. M. Sokolov. 2004. "Reshuffling scale-free networks: From random to assortative". *Physical Review E* vol. 70 (6), pp. 066102.

AUTHOR BIOGRAPHIES

HASANUZZAMAN BHUIYAN is a Ph.D. candidate in the Network Dynamics and Simulation Science Laboratory and Department of Computer Science at Virginia Tech. His email address is mhb@bi.vt.edu.

MALEQ KHAN is currently an Assistant Professor in the Department of Electrical Engineering and Computer Science at Texas A&M University—Kingsville. His email address is maleq.khan@tamuk.edu.

MADHAV MARATHE is the Director of the Network Dynamics and Simulation Science Laboratory and Professor of Computer Science at Virginia Tech. His email address is mmarathe@bi.vt.edu.

COMPARING ALLINEA'S AND INTEL'S PERFORMANCE TOOLS FOR HPC

Glenn R. Luecke
Department of Mathematics
Iowa State University
Ames, IA, 50011, USA
grl@iastate.edu

Brandon M. Groth
Department of Mathematics
Iowa State University
Ames, IA, 50011, USA
bmgroth@iastate.edu

Nathan T. Weeks
Department of Computer Science
Iowa State University
Ames, IA 50011, USA
weeks@iastate.edu

Marina Kraeva
Information Technology Services
Iowa State University
Ames, IA, 50011, USA
kraeva@iastate.edu

ABSTRACT

To efficiently use HPC machines, it is critical to optimize applications for high performance. To accomplish this, HPC developers must utilize performance tools to find and correct performance problems within large, complex scientific applications. Allinea and Intel offer vendor-supported performance tools that are regularly updated to capture important performance metrics on the latest hardware. In this paper, the authors evaluated and compared Allinea's MAP performance tool and Intel's performance tools to aid in application optimization. The authors found that Allinea's MAP provided useful performance metrics necessary to diagnose and fix performance problems using an intuitive, easy-to-use user interface. Intel's performance tools provided a more detailed and customizable view of application performance, at the expense of a more complicated user interface. The comparison presented in this paper will help HPC developers decide which performance tool is best for them.

Keywords: MAP, Performance Reports, Trace Analyzer, VTune

1 INTRODUCTION

Training application developers to write high performance applications is increasingly important for today's commercial, government and research organizations. This is challenging since HPC architectures and programming models are rapidly changing. Being able to evaluate an application's performance without special tools can be a difficult and time-consuming task. Performance tools are needed to evaluate application performance and identify bottlenecks, but if the tools are difficult to use, then few will use them. The purpose of this study is to evaluate and compare Allinea's and Intel's performance tools not only for the functionality needed to optimize applications, but also for ease-of-use.

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

Allinea offers two tools: Allinea MAP and Allinea Performance Reports. MAP and Performance Reports are currently being used at many universities and US government labs such as Oak Ridge National Laboratory, Los Alamos National Laboratory, and the National Energy Research Scientific Computing Center (NERSC). Intel offers Intel Parallel Studio, which contains the following performance tools used in this study: Intel Profile Function and Loop Execution Time, Intel Trace Analyzer and Collector, and Intel VTune Amplifier. Many organizations use Parallel Studio because it contains Intel's compilers, performance analysis tools, and their optimized libraries.

Some other tools available are TAU (University of Oregon), HPCToolkit (Rice University), Scalasca (Forschungszentrum Jülich), and Vampir (Dresden University of Technology). While TAU, HPCToolkit, and Scalasca are freely available, vendors and the HPC support staff at Iowa State University currently do not support them. Thus, we did not include these tools in this study.

In 2010, Marowka reported on the functionality and ease-of-use of the Intel Thread Profiler for Windows (Marowka 2010). In 2004, Collette et al. published a summary and comparison of many debuggers and performance tools used in HPC at the time (Collette, Corey, and Johnson 2004). Furthermore, Appelbe et al. in 1996 gave a summary of current software as well as recommendations for future improvements (Appelbe and Bergmark 1996). In addition, Moore et al. published a review of performance analysis tools for MPI in 2001 (Moore et al. 2001). The authors are not aware of any recent study dealing with Allinea's or Intel's performance tools.

This paper is organized as follows. Section 2 discusses our methodology and introduces the epiSNP bioinformatics code used in this study. Sections 3 and 4 discuss Allinea's MAP and Performance Reports. Sections 5, 6, and 7 discuss Intel's Profile Function and Loop Execution Time (PFLET), Trace Analyzer and Collector, and VTune Amplifier, respectively. Sections 8 contains the summary and conclusions.

2 METHODOLOGY

The evaluations in this study are for serial programs, and programs using MPI, OpenMP, or both MPI+OpenMP. The following evaluation categories represent what the authors consider to be most important when using a performance tool in an HPC educational environment:

- Was the tool easy to use when compiling and running applications?
- Is the GUI easy to use?
- Does the tool clearly present profiling data for lines, functions, and loops?
- Does the tool clearly present CPU, memory, and I/O data?
- Can the tool detect MPI and OpenMP load-balancing problems?
- Can the tool handle long-running jobs?

The evaluation was performed with a bioinformatics Fortran application comprised of 2000+ lines of code, called epiSNP, written by Ma (Ma et al. 2008) and optimized by Weeks (Weeks et al. 2016) aided by Allinea MAP and Intel's PFLET. There are several versions of the optimized epiSNP, including a serial version and a hybrid MPI+OpenMP version. For this study, we used the optimized serial epiSNP with PFLET and the optimized MPI+OpenMP epiSNP for all other tools. epiSNP was launched on 4 nodes with two 8-core sockets using 1 MPI rank per socket and 8 OpenMP threads per MPI rank. All runs were performed on Iowa State University's CyEnce Cluster (see http://www.hpc.iastate.edu/systems). The MPI+OpenMP epiSNP ran with this configuration had an approximate runtime of 7 hours with the 4.4GB data set used in Weeks (Weeks et al. 2016).

The following lists the versions of the software used for this study:

- Allinea Forge version 6.0
- Allinea Performance Reports version 6.0
- Intel Parallel Studio XE 16.0

Allinea MAP and Intel VTune can be used with accelerators, however we did not evaluate this feature for the following reasons. Allinea Forge requires an additional license, which we did not have at Iowa State University. Furthermore, we were not able to run VTune with the Intel Xeon Phi on CyEnce, even though we carefully followed the Intel documentation.

3 ALLINEA MAP

Allinea MAP is a performance tool that collects statistical samples for each line of code in an application. MAP offers profiling support for code written in Fortran, C, and C++.

To use MAP on an MPI+OpenMP Fortran program, called prog.f90, the only requirement is to compile the application with the "-g" compiler option to add symbolic debugging information to the executable:

```
mpiifort -g -qopenmp prog.f90
map ./a.out
```

The map command will launch the MAP Run Window via X11 forwarding and produce a .map file after the job has completed. When using MPI or OpenMP, one will have to provide the appropriate configuration before submitting the job. MAP can also be run without a GUI from a job script, by prefixing the run command with "`map --profile`", for p MPI processes:

```
map --profile mpirun -np p ./a.out
```

At the end of execution with "−−profile", a .map file will be generated. Once a .map file has been created, it can be used to examine MAP performance data without rerunning the application. This is accomplished by selecting Load Profile Data File in the Start Menu or by issuing:

```
map ./<MAP_file>.map
```

The "−−profile" option is useful for long-running jobs, since the profile data will be lost if the interactive X session is interrupted by network connection loss.

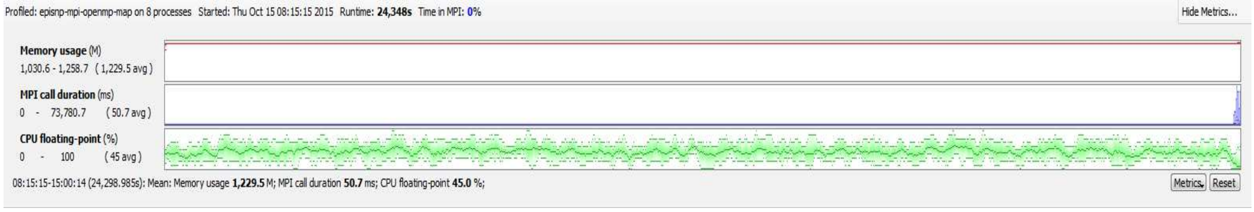
The GUI is laid out as follows:

- Top panel (Metric View): View a time line of several metrics (Figure 1).
- Middle panel (Source Code View): View statistical timing information for each line of the source code (Figure 2).
- Bottom panel (Stacks View): View a top-down tree of longest running lines/functions of code in an application (Figure 3).

In the Metric View, users can zoom in on a certain time interval by performing a "click-and-drag" on the metric's desired time frame. The other views will update to use this interval of time for execution time percentages and hotspots. Load imbalance between MPI processes or OpenMP threads can also be inferred by using the Metrics View.

MAP also has the ability to edit and recompile the source code within the GUI. It has built-in text editing options such as undo, redo, copy, paste, etc. The MAP GUI will not update to any changes until the edited code has been saved, compiled, and profiled.

The MAP GUI uses an X connection. Alternatively, users can download the Allinea Forge Remote Client to connect to remote clusters via SSH without X11 forwarding. The Remote Client provides a more responsive GUI by avoiding the latency of X11 forwarding. This is especially useful when there is a high-latency connection between the workstation and the cluster.



Profiled: episnp-mpi-openmp-map on 8 processes Started: Thu Oct 15 08:15:15 2015 Runtime: **24,348s** Time in MPI: **0%**

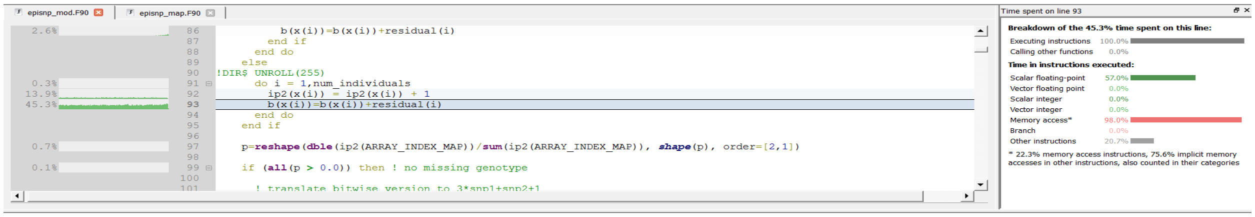
Memory usage (M)
1,030.6 - 1,258.7 (1,229.5 avg)

MPI call duration (ms)
0 - 73,780.7 (50.7 avg)

CPU floating-point (%)
0 - 100 (45 avg)

08:15:15-15:00:14 (24,298.985s): Mean: Memory usage **1,229.5** M; MPI call duration **50.7** ms; CPU floating-point **45.0** %;

Figure 1: MAP Metric View with default metrics.



```
 2.6%   86              b(x(i))=b(x(i))+residual(i)
        87            end if
        88          end do
        89        else
        90  !DIR$ UNROLL(255)
 0.3%   91          do i = 1,num_individuals
13.9%   92            ip2(x(i)) = ip2(x(i)) + 1
45.3%   93            b(x(i))=b(x(i))+residual(i)
        94          end do
        95        end if
        96
 0.7%   97        p=reshape(dble(ip2(ARRAY_INDEX_MAP))/sum(ip2(ARRAY_INDEX_MAP)), shape(p), order=[2,1])
        98
 0.1%   99        if (all(p > 0.0)) then ! no missing genotype
       100
       101          ! translate bitwise version to 3*snp1+snp2+1
```

Time spent on line 93

Breakdown of the 45.3% time spent on this line:

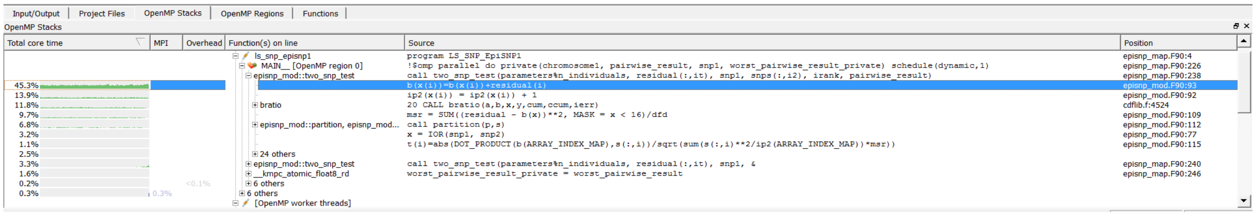
Executing instructions	100.0%
Calling other functions	0.0%

Time in instructions executed:

Scalar floating-point	57.0%
Vector floating point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access*	98.0%
Branch	0.0%
Other instructions	20.7%

* 22.3% memory access instructions, 75.6% implicit memory accesses in other instructions, also counted in their categories

Figure 2: MAP Source Code View and Selected Line View (right).



OpenMP Stacks

Total core time	MPI	Overhead	Function(s) on line	Source	Position
			ls_snp_episnp1	program LS_SNP_EpiSNP1	episnp_map.F90:4
			MAIN__ [OpenMP region 0]	!$omp parallel do private(chromosome1, pairwise_result, snp1, worst_pairwise_result_private) schedule(dynamic,1)	episnp_map.F90:226
			episnp_mod::two_snp_test	call two_snp_test(parameters%n_individuals, residual(:,it), snp1, snps(:,i2), irank, pairwise_result)	episnp_map.F90:238
45.3%				b(x(i))=b(x(i))+residual(i)	episnp_mod.F90:93
13.9%				ip2(x(i)) = ip2(x(i)) + 1	episnp_mod.F90:92
11.8%			bratio	20 CALL bratio(a,b,x,y,cum,ccum,ierr)	cdflib.f:4524
9.7%				msr = SUM((residual - b(x))**2, MASK = x < 16)/dfd	episnp_mod.F90:109
6.8%			episnp_mod::partition, episnp_mod...	call partition(p,s)	episnp_mod.F90:112
3.2%				x = IOR(snp1, snp2)	episnp_mod.F90:77
1.1%				t(i)=abs(DOT_PRODUCT(b(ARRAY_INDEX_MAP),s(:,i))/sqrt(sum(s(:,i)**2/ip2(ARRAY_INDEX_MAP))*msr))	episnp_mod.F90:115
2.5%			24 others		
3.3%			episnp_mod::two_snp_test	call two_snp_test(parameters%n_individuals, residual(:,it), snp1, &	episnp_map.F90:240
1.6%			__kmpc_atomic_float8_rd	worst_pairwise_result_private = worst_pairwise_result	episnp_map.F90:246
0.2%		<0.1%	6 others		
0.3%	0.3%		6 others		
			[OpenMP worker threads]		

Figure 3: MAP OpenMP Stack.

3.1 Evaluation

The user can choose between launching the GUI interface on the cluster using a standard job script or using the Remote Client to profile an application. The authors found that the remote client was helpful in keeping the GUI responsive between workstation and remote cluster. With X11 forwarding, the authors experienced slow response times to any actions performed on the GUI. Whereas, with the remote client, the response time dramatically improved.

The authors found that the initial data view was easy to navigate and the time interval feature of the Metric View was especially useful. With statistical changes occurring in all three views for a particular time frame, this can help pinpoint local bottlenecks in source code, as well as what may be causing the bottleneck based on performance metrics. Furthermore, a user can find load imbalance in their application by using this technique. Also, the pre-made groups in the drop-down menu provide good coverage of important metrics such as vectorization, memory usage, MPI communication, and OpenMP synchronization.

The authors felt that MAP was easy to use, as well provided the functionality needed to optimize programs. However, MAP lacks the ability to track cache misses, whereas Intel VTune does have this functionality (see section 7).

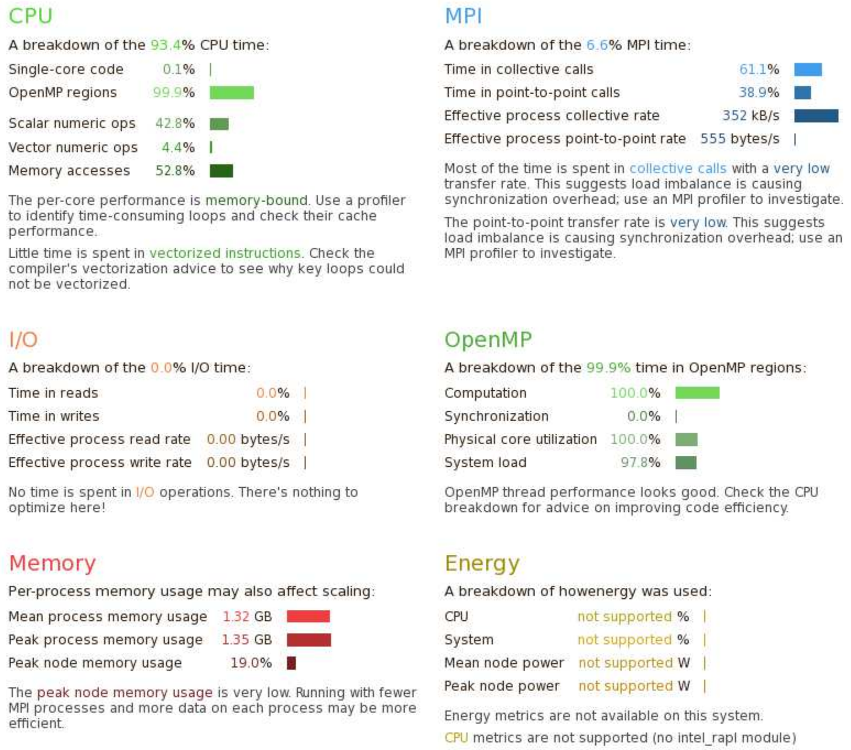
4 ALLINEA PERFORMANCE REPORTS

Allinea Performance Reports (Reports) is a low-overhead tool that produces a one-page report summarizing CPU, MPI, I/O, OpenMP, memory, and energy performance information. In addition, Performance Reports gives high-level suggestions for improving performance. Reports can be used on applications written in C, C++, and Fortran. Performance Reports does not offer performance data at the line, function, or loop levels.

To run Performance Reports, one adds "`perf-report`" before the run command, e.g.:

```
perf-report mpirun -np p ./a.out
```

The user does not have to recompile with the "`-g`" option, as Performance Reports doesn't collect data at the source-code level. After running, Performance Reports generates text and HTML files containing identical information, but the HTML file has accompanying graphics. The text file can be opened with any text editor, while the HTML file can be opened with any web browser. For each of these breakdowns in Figure 4, a message follows telling how well the area performed and gives advice on improving performance, as well what to look for when profiling.



CPU

A breakdown of the 93.4% CPU time:

Single-core code	0.1%
OpenMP regions	99.9%
Scalar numeric ops	42.8%
Vector numeric ops	4.4%
Memory accesses	52.8%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 6.6% MPI time:

Time in collective calls	61.1%
Time in point-to-point calls	38.9%
Effective process collective rate	352 kB/s
Effective process point-to-point rate	555 bytes/s

Most of the time is spent in collective calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

The point-to-point transfer rate is very low. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

OpenMP

A breakdown of the 99.9% time in OpenMP regions:

Computation	100.0%
Synchronization	0.0%
Physical core utilization	100.0%
System load	97.8%

OpenMP thread performance looks good. Check the CPU breakdown for advice on improving code efficiency.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	1.32 GB
Peak process memory usage	1.35 GB
Peak node memory usage	19.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of howenergy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	not supported W

Energy metrics are not available on this system.

CPU metrics are not supported (no intel_rapl module)

Figure 4: Performance Report Breakdowns.

4.1 Evaluation

Allinea Performance Reports is different from other tools included in this survey. To start, there is only one step necessary to use Performance Reports - prefixing the command to be profiled. Users do not have to recompile the application to use Performance Reports. There is no learning curve for viewing the output as it comes in text or a viewable web page. Also, the authors found the optimization advice given for each category to be helpful. Another benefit of Performance Reports is having an idea of the bottlenecks in a user's application before using a full-featured performance tool like Allinea MAP (section 3) or Intel VTune (section 7). Knowing whether an application is memory, compute, or communication bound makes it easier to focus on the most important performance metrics when using a profiler that provides more detailed performance information.

5 INTEL PROFILE FUNCTION OR LOOP EXECUTION TIME

The Intel compiler contains a performance tool called "Profile Function or Loop Execution Time" (PFLET), which is a serial profiler. Information provided by PFLET can help one to decide which code portions to optimize and which to parallelize.

To profile with PFLET, issue

```
ifort -profile-loops=all prog.f90
./a.out
```

Here, "`-profile-loops=all`" enables the compiler to time function calls and loops in the program. Also, the "`-profile-loops-report=2`" option uses additional instrumentation to record loop iteration counts and reports min, max, and average iteration counts.

After the execution ends, PFLET will generate up to two .dump files (one for functions and one for loops), as well as .xml files containing the same data. Using any text editor, users can open the .dump files to view the results (see Table 1). The output is formatted into a table, where each row corresponds to a single function or loop labeled as file:line in the last column.

Table 1: PFLET Dump File Output.

time(abs)	time(%)	self(abs)	self(%)	loop_entries	function	function_file:line	loop_file:line
5745418040135	52.1	5745418040135	52.1	1080869265	episnp_mod_mp_two_snp_test_..0	episnp_mod.F90:57	episnp_mod.F90:92
1535455501662	13.9	1535455501662	13.9	518803997	episnp_mod_mp_two_snp_test_..0	episnp_mod.F90:57	episnp_mod.F90:109
493391216573	4.5	493391216573	4.5	1080869265	episnp_mod_mp_two_snp_test_..0	episnp_mod.F90:57	episnp_mod.F90:77
446037342805	4.0	446037342805	4.0	1080869265	episnp_mod_mp_two_snp_test_..0	episnp_mod.F90:57	episnp_mod.F90:92
351345228991	3.2	351345228991	3.2	1353462421	bpser_	cdflib.f:2307	cdflib.f:2397
201603090096	1.8	201603090096	1.8	518813997	episnp_mod_mp_partition_	episnp_mod.F90:125	episnp_mod.F90:369
66765839894	0.6	57666794224	0.5	721888762	bgrat_	cdflib.f:2714	cdflib.f:2775
46256515042	0.4	46256515042	0.4	518803997	episnp_mod_mp_two_snp_test_..0	episnp_mod.F90:57	episnp_mod.F90:109

5.1 Evaluation

PFLET is designed to only profile loops and functions, and not at the statement level. The text output is easy to interpret, as it doesn't require using a GUI. The authors consider the lack of statement-level profiling to be major drawback, especially for applications with large function or loop bodies. Furthermore, PFLET was not designed to use OpenMP. For these reasons, this tool's utility is limited to identifying loops or functions in a serial program that may benefit from parallelization or other optimizations.

6 INTEL TRACE ANALYZER AND COLLECTOR

Intel Trace Analyzer and Collector (ITAC) is a graphical tool used for understanding MPI application behavior. ITAC can help identify load imbalances and investigate communication correctness using traces.

To compile and collect performance data for ITAC, issue:

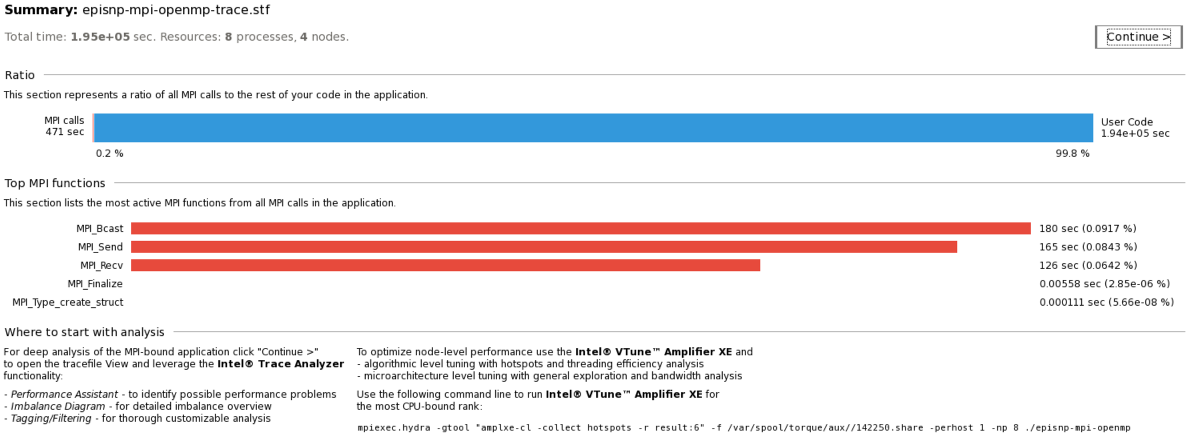
```
mpiifort -g -qopenmp prof.f90
mpirun -trace -np p ./a.out
```

The “`-trace`” option will create trace files, one per MPI process, containing information through the entire execution of the application, including MPI communication. To launch the ITAC GUI, issue:

```
traceanalyzer ./a.out.stf
```

where .stf is a Structured Trace File (STF) generated by “`-trace`”. On start up, ITAC will open the Summary Page for the application, which contains a Ratio bar and a list of longest-running MPI calls (see Figure 5). This allows one to quickly see if the application is compute-bound or communication-bound.

Continuing from the Summary Page, the ITAC GUI opens with the Flat Profile tab on the left. The Load Balance tabs shows ratios of user-code and MPI communication for each MPI process. This data can be viewed in a series of bar graphs or pie charts per MPI process (see Figure 6). There is also a Call Tree and Call Graph in the Flat Profile, where both show different MPI calls executed throughout program. However, we did not find these features to be useful for our application.The Event Timeline displays MPI process activity as MPI communication (red) and user code (blue), with black lines connecting processes that are communicating. Through the Event Timeline, by right-clicking on an MPI process and navigating through a series of windows, users can determine the current line of code.



Summary: episnp-mpi-openmp-trace.stf

Total time: **1.95e+05** sec. Resources: **8** processes, **4** nodes.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Bcast	180 sec (0.0917 %)
MPI_Send	165 sec (0.0843 %)
MPI_Recv	126 sec (0.0642 %)
MPI_Finalize	0.00558 sec (2.85e-06 %)
MPI_Type_create_struct	0.000111 sec (5.66e-08 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use the **Intel® VTune™ Amplifier XE** and
- algorithmic level tuning with hotspots and threading efficiency analysis
- microarchitecture level tuning with general exploration and bandwidth analysis

Use the following command line to run **Intel® VTune™ Amplifier XE** for the most CPU-bound rank:

```
mpiexec.hydra -gtool "amplxe-cl -collect hotspots -r result:6" -f /var/spool/torque/aux//142250.share -perhost 1 -np 8 ./episnp-mpi-openmp
```

Figure 5: ITAC Summary Page.

6.1 Evaluation

Once past the Summary Page, the features of ITAC are hidden behind blank panels. The window is filled with a large amount of unused blank space, with essentially no data displayed. Furthermore, many of the drop-down “arrows" require two clicks for data to be shown. Another issue occurs when using the Application and MPI tables in the Function Profile. By clicking on the first arrow, it makes the second arrow (MPI) move below a list of all MPI ranks. Because of this, the authors consider profiling a job with many MPI processes to be difficult. These are all GUI design flaws that slow user interaction with the profile data.

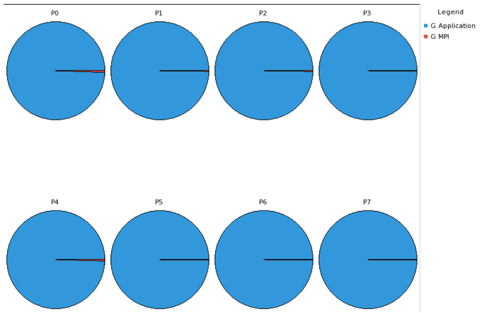


Figure 6: ITAC Load Balance Graph.

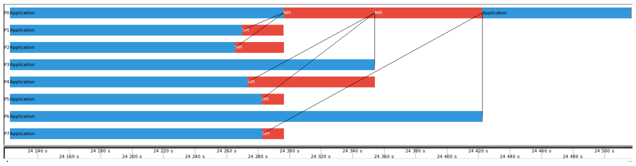


Figure 7: ITAC Event Timeline Chart.

The authors found the Load Balance Graph (Figure 6) and the Event Timeline Chart (Figure 7) to be the most useful in identifying and locating MPI communication problems. The Load Balance Function Profile helps one identify poorly-performing MPI ranks by looking at how long they spend doing computation and communication. The Event Timeline then helps identify when and where the problem occurs. When finding a performance problem with the Event Timeline, one needs to find the source code causing the problem. The authors were unable to access this data, despite following the documentation instructions and found the process of accessing the source code to be unintuitive and tedious. Furthermore, the documentation says that OpenMP regions are viewable from the Event Timeline, but we were unable to activate them.

7 INTEL VTUNE AMPLIFIER

Intel VTune Amplifier XE (VTune) is a performance tool that gathers performance data of serial or shared memory applications using threads. VTune offers support for applications written in C, C++, C#, Fortran, Java, assembly, as well as applications using accelerators such as the Intel Xeon Phi coprocessor and GPUs with OpenCL. Furthermore, VTune supports OpenMP, Intel Threading Building Blocks, and Intel Cilk Plus. VTune supports MPI, however it only stores performance data for MPI processes without communication. To profile communication among MPI processes, MAP or ITAC can be used (see Section 3 or Section 6).

Intel has chosen to use the command "`amplexe`" instead of the common name Vtune for job submission. To profile a serial or pure OpenMP application, the VTune GUI can launch the application for data collection (`amplxe-gui`). To profile MPI or MPI+OpenMP applications with VTune, one must use the command line interface to collect the data (`amplxe-cl`). To compile and collect Basic Hotspots performance data on serial or pure OpenMP code, issue

```
ifort -g -qopenmp prog.f90
amplxe-cl -quiet -collect hotspots ./a.out
```

For MPI+OpenMP jobs, this changes to:

```
mpiifort -g -qopenmp prog.f90
mpirun -np p amplxe-cl -quiet -collect hotspots -result-dir my_dir ./a.out
```

The "`-result-dir`" option is required for MPI profiling, as VTune holds each MPI rank's profile data in different directories. For long-running jobs it is recommended to use the "`-quiet`" option. This will disable VTune's status I/O.

Besides Basic Hotspots, VTune offers many different collection options. These collection options are broken into three groups: Algorithm Analysis, Microarchitecture Analysis, and Platform Analysis. The different collections for each of these groups are as follows:

- Algorithm Analysis: Basic Hotspots, Advanced Hotspots, Concurrency, Locks and Waits, HPC Performance Characterization.
- Microachitecture Analysis: General Exploration, Memory Access, SGX Hotspots, TSX Hotspots, TSX Exploration.
- Platform Analysis: CPU/GPU Concurrency, System Overview.

A feature of interest of relevant interest to HPC is the HPC Performance Characterization. This option collects performance metrics relevant to HPC applications such as CPU usage, FPU usage, GFLOPS, and memory bandwidth information. As of Parallel Studio 2016, this was a preview feature that was not available on our cluster. Intel has since added this into Parallel Studio 2017, but we were unable to test it.

After the collection job has completed, analysis can be performed by using the VTune GUI or command line regardless how the data was collected. For programs on remote servers, the command line version may be faster than the GUI due to X11 forwarding. However, not all of the data that VTune collects can be reported through command line. To view the collection results of rank "`<rank>`" issue

```
amplxe-gui my_dir.<rank>
```

To use the command line with an appropriate report type, issue

```
amplxe-cl -report <report-type> -result-dir my_dir.<rank>
```

Since most people will be using the GUI to analyze the performance data, we continue to use amplxe-gui.

The sections within the Summary pane show execution time, function hotspots, OpenMP execution time, OpenMP hotspots, and OpenMP load balancing. For additional information on each of these sections, see the Intel VTune User Guide.

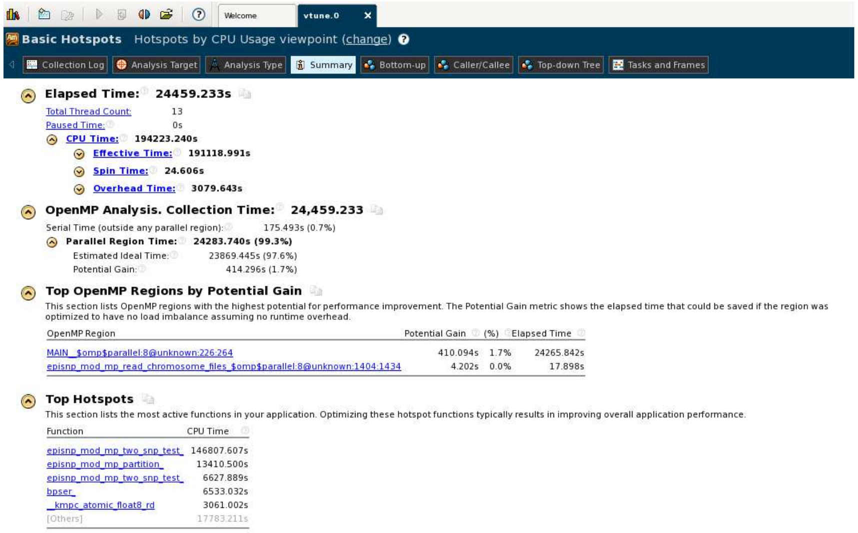
In addition to the Summary pane, there are other windows that use a different format. These windows are:

- Bottom-up: Displays self timings for lines, functions, and loops.
- Caller/Callee: Displays parent and child function calls.
- Top-down Tree: Displays both self timings and total timings.
- Tasks and Frames: Displays a time line for tasks (logical unit of work) and frames (period of time between beginning and end points).

The Bottom Up pane only displays self timings, while the Top Down pane displays both self and total timings. For this reason, the Top Down pane is sufficient for code analysis (Figure 9). Furthermore, by right-clicking and selecting "View Source", one can see the function call site or the largest bottleneck contained within the function if the call site is a call leaf. The order of this tree is controlled by selecting a Call Stack metric, which is in the rightmost drop down menu. The main metrics are CPU Time (default), Wait Time, Wait Count, Spin Time, Context Switch Time, and Task Time. There are also more specialized metrics based on these main metrics, as well as specific hardware event collection metrics.

7.1 Evaluation

Intel VTune is a performance tool that allows users to find problems with computation, memory, and thread performance. It has a steep learning curve because the tool offers many options to the user and the GUI is



Elapsed Time: 24459.233s

Total Thread Count: 13
Paused Time: 0s
CPU Time: 194223.240s
Effective Time: 191118.991s
Spin Time: 24.606s
Overhead Time: 3079.643s

OpenMP Analysis. Collection Time: 24,459.233

Serial Time (outside any parallel region): 175.493s (0.7%)
Parallel Region Time: 24283.740s (99.3%)
Estimated Ideal Time: 23869.445s (97.6%)
Potential Gain: 414.296s (1.7%)

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

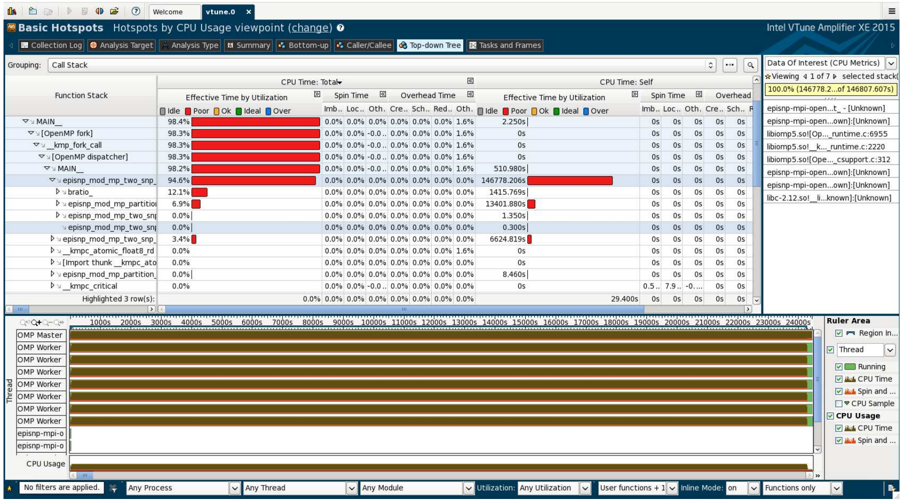
OpenMP Region	Potential Gain	(%)	Elapsed Time
MAIN__ompparallel:8@unknown:226:264	410.094s	1.7%	24265.842s
episnp_mod_mp_read_chromosome_files_ompparallel:8@unknown:1404:1434	4.202s	0.0%	17.898s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	CPU Time
episnp_mod_mp_two_snp_test_	146807.607s
episnp_mod_mp_partition_	13410.500s
episnp_mod_mp_two_snp_test_	6627.889s
bpser_	6533.032s
__kmpc_atomic_float8_rd	3061.002s
[Others]	17783.211s

Figure 8: VTune Summary.



Grouping: Call Stack

	CPU Time: Total								CPU Time: Self					
	Effective Time by Utilization	Spin Time			Overhead Time				Effective Time by Utilization	Spin Time			Overhead	
Function Stack	Idle Poor Ok Ideal Over	Imb..	Loc..	Oth.	Cre..	Sch..	Red..	Oth.	Idle Poor Ok Ideal Over	Imb..	Loc..	Oth.	Cre..	Sch..
MAIN__	98.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.6%	2.250s	0s	0s	0s	0s	0s
[OpenMP fork]	98.3%	0.0%	0.0%	-0.0..	0.0%	0.0%	0.0%	1.6%	0s	0s	0s	0s	0s	0s
__kmp_fork_call	98.3%	0.0%	0.0%	-0.0..	0.0%	0.0%	0.0%	1.6%	0s	0s	0s	0s	0s	0s
[OpenMP dispatcher]	98.3%	0.0%	0.0%	-0.0..	0.0%	0.0%	0.0%	1.6%	0s	0s	0s	0s	0s	0s
MAIN__	98.2%	0.0%	0.0%	-0.0..	0.0%	0.0%	0.0%	1.6%	510.980s	0s	0s	0s	0s	0s
episnp_mod_mp_two_snp_	94.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	146778.206s	0s	0s	0s	0s	0s
bratio_	12.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1415.769s	0s	0s	0s	0s	0s
episnp_mod_mp_partitio	6.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	13401.880s	0s	0s	0s	0s	0s
episnp_mod_mp_two_snp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.350s	0s	0s	0s	0s	0s
episnp_mod_mp_two_snp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.300s	0s	0s	0s	0s	0s
episnp_mod_mp_two_snp_	3.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	6624.819s	0s	0s	0s	0s	0s
__kmpc_atomic_float8_rd	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.6%	0s	0s	0s	0s	0s	0s
[Import thunk __kmpc_ato	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0s	0s	0s	0s	0s	0s
episnp_mod_mp_partition_	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	8.460s	0s	0s	0s	0s	0s
__kmpc_critical	0.0%	0.0%	0.0%	-0.0..	0.0%	0.0%	0.0%	0.0%	0s	0.5..	7.9..	-0....	0s	0s
Highlighted 3 row(s):	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	29.400s	0s	0s	0s	0s	0s

Data Of Interest (CPU Metrics)
Viewing 1 of 7 selected stack(
100.0% (146778.2...of 146807.607s)

episnp-mpi-open...t_ - [Unknown]
episnp-mpi-open...own]:[Unknown]
libiomp5.so![Op..._runtime.c:6955
libiomp5.so!__k..._runtime.c:2220
libiomp5.so![Ope..._csupport.c:312
episnp-mpi-open...own]:[Unknown]
episnp-mpi-open...own]:[Unknown]
libc-2.12.so!__li...known]:[Unknown]

No filters are applied. Any Process | Any Thread | Any Module | Utilization: Any Utilization | User functions + 1 | Inline Mode: on | Functions only

Figure 9: VTune Top-Down Tree Window.

often not intuitive. The VTune's Basic Hotspots analysis presents performance data that is usually sufficient to optimize a program.

Once data collection has been completed, users can further choose between the GUI or the command line to view the data analysis. The authors found the Summary and Top-down Tree windows to provide the most useful performance data. The Summary window does a good job covering generic metrics for a program, including interesting items such as parallel Potential Gain. Furthermore, it offers an easy-to-read load balance chart for OpenMP, which many users will find useful. Statistical timing information for functions and lines of code can be viewed in the Top-down and Bottom-Up Windows. The table format needs improvement as it often contains many columns of data that are empty.There are options to view the source code within VTune by using the "View Source" (via right-click) for particular functions and lines. The authors feel it should be easier to see the source code along with the performance data.

For jobs that have long execution times, the Hardware Event-Based Sampling collection types may lead to significant amounts of data collected. For example, profiling epiSNP with the Advanced Hotspots analysis resulted in large 64GB directories per MPI rank, with four MPI processes used for the job, one per node. When trying to open files this large, the GUI would become unresponsive. It was much simpler using the command line report feature and forgoing the graphics than dealing with the slow response times of the GUI.

8 SUMMARY AND CONCLUSIONS

Using the hybrid MPI+OpenMP version of epiSNP, the authors evaluated Allinea and Intel performance tools for functionality and ease-of-use. Table 2 contains the tool Functionality Summary.For our functionality criteria, the tool that scored the best was MAP, as it contained the features needed for application optimization. MAP has an Yes* for detecting MPI load imbalance because the Metric View can imply a load imbalance, but does not offer direct detection like in ITAC. Intel Trace Analyzer and Collector (ITAC) scores poorly in this category, as it lacks detailed source code support, memory metrics, and I/O metrics. However, the authors felt that ITAC's MPI functionality, specifically the Event Timeline Chart and the Load Balance Graph, were especially useful.

Table 2: Functionality Summary.

Category	MAP	Reports	PFLET	ITAC	VTune
Profiling Lines?	Yes	No	No	No	Yes
Profiling Functions?	Yes	No	Yes	Yes	Yes
Profiling Loops?	Yes	No	Yes	No	No
CPU Performance Data?	Yes	Yes	Yes	Yes	Yes
Memory Performance Data?	Yes	Yes	No	No	Yes
I/O Performance Data?	Yes	Yes	No	No	Yes
Detect MPI Load Imbalance?	Yes*	No	No	Yes	No
Detect OpenMP Load Imbalance?	Yes	Yes	No	No	Yes

To evaluate the performance of an MPI application using Intel's performance tools, one must use two tools: ITAC for MPI communication and either VTune or PFLET to pinpoint bottlenecks in source code.Whereas, Allinea MAP is a single tool that provides both performance data for MPI communication and source-code-level profiling information. The authors consider this an advantage of MAP over ITAC or VTune. Furthermore, the authors consider MAP's user interface to be more intuitive than either ITAC's or VTune's dissimilar interfaces.

Allinea also has a high-level, easy-to-use performance tool called Performance Reports. This tool does not require recompilation. Combined, the ITAC and VTune summary pages provide similar information compared to Performance Reports, but this requires the application to be run twice (once for each tool). In addition, Performance Reports provides suggestions for performance enhancements, whereas Intel's tools do not. The authors feel that these suggestions are valuable and hope that other tools will implement similar analyses to aid in program optimization. However, to find performance problems after using Performance Reports, additional runs with other tools requiring source code compilation will likely be needed.

GUI responsiveness is another important consideration. The Remote Client for MAP provides a significant improvement in usability, as it reduces communication lag compared to X11 forwarding. It also reduces the load on the login node of the cluster, as image rendering is moved to the user's workstation. The Ease-of-Use Summary (Table 3) contains the authors impressions of using the various tools. In this table, we gave two ratings: Satisfactory (S) and Needs Improvement (NI). Here, Performance Reports and Intel Performance Function or Loop Execution Time (PFLET) do not use GUIs, so they were categorized as N/A.

Table 3: Ease-of-Use Summary. S stands for Satisfactory and NI stands for Needs Improvement.

Category	MAP	Reports	PFLET	ITAC	VTune
Navigation of GUI	S	N/A	N/A	NI	S
Data Presentation	S	S	NI	S	NI
Handles Long-running Jobs	S	S	S	NI	NI

ACKNOWLEDGMENTS

The research reported in this paper is partially supported by the HPC@ISU equipment at Iowa State University, some of which has been purchased through funding provided by NSF under MRI grant number CNS 1229081 and CRI grant number 1205413. Personnel time was fully funded by Iowa State University.

REFERENCES

Appelbe, B., and D. Bergmark. 1996. “Software Tools for High Performance Computing: Survey and Recommendations”. *Scientific Programming* vol. 5, pp. 239–249.

Collette, M., B. Corey, and J. Johnson. 2004, December. “High Performance Tools & Technologies”. Technical report, Lawrence Livermore National Laboratory.

Ma, L., H. B. Runesha, D. Dvorkin, J. R. Garbe, and Y. Da. 2008. “Parallel and serial computing tools for testing single-locus and epistatic SNP effects of quantitative traits in genome-wide association studies”. *BMC Bioinformatics* vol. 9 (1), pp. 315.

Marowka, A. 2010, July. “A Study of the Usability of Multicore Threading Tools”. *International Journal of Software Engineering and Its Applications* vol. 4 (3).

Moore, S., D. Cronk, K. London, and J. Dongarra. 2001, September. “Review of Performance Analysis Tools for MPI Parallel Programs”. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*.

Weeks, N. T., G. R. Luecke, B. M. Groth, M. Kraeva, L. Ma, L. M. Kramer, J. E. Koltes, and J. M. Reecy. 2016, July. “High-performance epistasis detection in quantitative trait GWAS”. *The International Journal of High Performance Computing Applications* .

AUTHOR BIOGRAPHIES

GLENN R. LUECKE received his Ph.D. in mathematics from the California Institute of Technology. He is currently Professor of Mathematics and director of HPC education and training at Iowa State University.

BRANDON GROTH is a research assistant in the Department of Mathematics HPC group at Iowa State University. His research interests include scientific computing, mathematical applications in the sciences, and numerical analysis.

NATHAN WEEKS is currently pursuing a Ph.D. in Computer Science at Iowa State University. His research interests include parallel computing, software application optimization, and bioinformatics.

MARINA KRAEVA received her Ph.D. in Computer Science from the State Technical University of Novosibirsk, Russia in 1999 and joined the High Performance Computing group at Iowa State University.

TO SHARE OR NOT TO SHARE: COMPARING BURST BUFFER ARCHITECTURES

Lei Cao
Bradley W. Settlemyer
High Performance Computing
Los Alamos National Laboratory
Los Alamos, NM, USA
{leicao88124,bws}@lanl.gov

John Bent
Seagate Government Systems
Los Alamos, NM, USA
john.bent@seagategov.com

ABSTRACT

Modern high performance computing platforms employ burst buffers to overcome the I/O bottleneck that limits the scale and efficiency of large-scale parallel computations. Currently there are two competing burst buffer architectures. One is to treat burst buffers as a dedicated shared resource, The other is to integrate burst buffer hardware into each compute node. In this paper we examine the design tradeoffs associated with local and shared, dedicated burst buffer architectures through modeling. By seeding our simulation with realistic workloads, we are able to systematically evaluate the resulting performance of both designs. Our studies validate previous results indicating that storage systems without parity protection can reduce overall time to solution, and further determine that shared burst buffer organizations can result in a 3.5x greater average application I/O throughput compared to local burst buffer configurations.

Keywords: Checkpoint-Restart, Burst Buffers, Storage Systems, File Systems and I/O

1 INTRODUCTION

The overwhelming majority of high-performance computing applications are tightly-coupled, bulk-synchronous parallel simulations that run for days or weeks on supercomputers. Due to both the tight-coupling and the enormous memory footprints used by these application several complexities emerge. One complexity is that the interruption of any process destroys the distributed state of the entire application. Unfortunately, as supercomputers become increasingly powerful, interruptions become more frequent and the size of the distributed state grows. To ensure forward progress these applications use checkpoint-restart, the conceptually simple technique of persisting their distributed state at regular time intervals, such that when an application process fails progress can be restarted from the most recently saved checkpoint. A second complexity is that capturing a consistent view of the distributed state is difficult due to messages in flight. To construct a consistent view of the simulation state, applications quiesce messaging and pause their forward progress for the duration of the checkpoint. These large, frequent bulk-synchronous checkpoints require ever more powerful storage systems capable of ingesting parallel data at massive rates.

Burst buffers, specialized high bandwidth storage tiers composed of relatively expensive solid state media, have arisen to address this performance requirement (Liu, Cope, Carns, Carothers, Ross, Grider, Crume, and Maltzahn 2012, Kimpe, Mohror, Moody, Van Essen, Gokhale, Ross, and de Supinski 2012, Wang, Oral, Wang, Settlemyer, Atchley, and Yu 2014). However, these tiers are too expensive to satisfy the capacity requirements of supercomputers, thus a less expensive, larger disk-based backing storage system with longer data retention and high reliability is also needed. Simple in concept, there are myriad possible configurations for these systems and no systematic means to compare them. For example, while procuring the Department

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

of Energy's first extreme-scale production supercomputer with burst buffers, Los Alamos National Laboratory's (LANL's) Trinity, we encountered several storage system design alternatives but had few capabilities to compare alternatives beyond ensuring vendor offerings satisfied our minimum requirements.

In this paper we describe a small portion of the rich design space of possible burst buffer architectures and compare their efficiency using event-driven simulation. In order to ensure accurate measurements of efficiency, we generate realistic workloads based on the Alliance for Application Performance at Extreme Scale (APEX) Workflows document (LANL, NERSC, and SNL 2016), which describes patterns of I/O exhibited by scientific applications over periods of several months. Our results compare the efficiency of multiple burst buffer designs, and in doing so, allows us to identify several new critical design elements of burst buffer design, including the policies used to manage data staging.

2 RELATED WORK

Due to the observation that multi-tier storage systems are the most cost-effective mechanism by which to provide high bandwidth storage systems to HPC platforms (Bent, Settlemyer, and Grider 2016), the design of burst buffers has been an area of recent interest. Three relevant burst buffer architectures have previously been described, along with their relative strengths and weaknesses (Harms, Oral, Atchley, and Vazhkudai 2016). The compute node-local architecture, with storage devices on each compute node, provides exclusive and interference-free access to the storage devices and linear scaling of bandwidth with the number of compute nodes used by an application. However, common file access techniques, such as many readers or writers sharing a single file (called N-to-1 file access) are difficult to enable. The second model, where burst buffer storage is available on dedicated I/O nodes, supports the N-to-1 access model more easily, and provides opportunities for simpler resilience. However, if more than a single job is running on the supercomputer (a common occurrence), then the jobs must share access to the burst buffer nodes, and risk unpredictable performance due to interference. The final described model co-located the fast storage on the nodes hosting long-term storage. This model creates an off-platform burst buffer, which may not provide the highest absolute performance, but may provide significant ease of use benefits via transparent caching. Although in this paper we only explore the first and second models, our simulator has the flexibility to explore additional paradigms such as network-attached storage, rack-local storage, and hybrids between all of these possibilities.

Both analytical models and simulation have been used to better evaluate the tradeoffs within burst buffer storage systems. An analytical modeling exercise determined that unlike traditional HPC storage systems, a burst buffer provides the most forward progress by providing no reliability (Bent, Settlemyer, DeBardeleben, Faibish, Ting, Gupta, and Tzelnic 2015). Although an unreliable burst buffer may repeatedly fail in such a manner that a long running simulation must restart from the very beginning, the probability is quite low and the improved performance provided by disabling parity protection improves overall throughput enough to offset this possibility. In this paper we replicate this result via simulation.

In simulations of a shared burst buffer architecture for the IBM BlueGene/Q supercomputer (Peng, Divanji, Raicu, and Lang 2016), it was determined that network bottlenecks were likely to be the largest performance limiter. In our simulated architecture, it does not appear that network performance is a limiting factor, but the Cray architecture is quite dissimilar from the simulated IBM architecture. Ongoing work using the ROSS simulator (Carothers, Bauer, and Pearce 2000), is similar to ours in that it focuses on creating a set of realistic I/O workloads (Liu, Cope, Carns, Carothers, Ross, Grider, Crume, and Maltzahn 2012). We are not aware of any other work that generates a realistic HPC workload using historical data and detailed user interviews such as those summarized in the APEX workflow documentation.

Table 1: The subset of the workload information provided by the APEX workflows document used to construct our simulator workload.

Workflow	EAP	LAP	Silverton	VPIC
Workload Percentage	60	5	15	10
Walltime (hours)	984	20	192	144
Hero Run Cores	65536	32768	131072	65536
Routine Number of Cores	32768	4096	32768	32768
Number of Workflow Pipelines per Allocation	125	35	36	24
Anticipated Increase In Problem Size By 2020	10 to 12x	8 to 12x	8 to 16x	8 to 16x
Anticipated Increase In Workflow Pipelines Per Allocation By 2020	2x	1x	1x	1x
Storage APIs	POSIX	POSIX	POSIX	POSIX
Routine Number of Analysis Datasets	100	100	225	150
Checkpoint Style	N to 1	N to 1	N to 1	N to N

3 METHODS

The goal of our simulation is to provide high quality estimates of the performance trade offs associated with several candidate burst buffer architectures. While in our past work we used analytical methods to evaluate storage system design points (such as reliability), here we use simulation to better capture the performance dynamics in the presence of randomly generated errors and faults.

We implemented our simulator based on the Simpy discrete-event simulation framework (Matloff 2008). Events operate on the following entities: a scheduler, jobs, compute nodes, burst buffers nodes, a fault generator, and a parallel file system. Note that burst buffer nodes are only included when simulating shared burst buffer configurations; for local configurations, the burst buffers are on the compute nodes and thus not represented independently by the simulator.

3.1 Workload Generation

In order to build a workload generator for our simulation, we have mined requirements from the APEX Workflows white paper (LANL, NERSC, and SNL 2016). One limitation of the APEX Workflow analysis is that large proportions of the possible laboratory workloads were not well described. While some details for the other sites are provided, and we could speculatively extrapolate projections, we instead chose to simply use the descriptions provided by LANL to construct our entire simulated workload. We recognize that even for LANL's supercomputers this is an overly simplistic assumption as fully two-thirds of LANL's computational cycles are generated by external users.

Table 1 shows a subset of the LANL workload information provided in the workflow document. The APEX workflows paper goes into great detail describing the concepts of campaigns, workflows, pipelines, jobs, and checkpoint strategies. In this paper we will briefly define a *workflow* as the computational process a scientist uses to answer some scientific inquiry. Within a scientific workflow, the scientist then uses some number of workflow *pipelines* (Thain, Bent, Arpaci-Dusseau, Arpaci-Dusseau, and Livny 2003), which are dependent sets of jobs, to simulate and evaluate their hypothesis. For example, a cosmologist may be interested in exploring relativistic particle trajectories as two plasmas move through space. The workflow then would use the Vector Particle-in-Cell (VPIC) code to simulate the plasmas and particles, and the workflow may require multiple pipelines as the cosmologist explores multiple types of plasma intersections and collisions. Each of the initial plasma setup pipelines may take weeks of computational time, though multiple pipelines can be executed in parallel (as large series of parallel jobs).

The LANL workload includes 4 major workflows (EAP, LAP, Silverton, and VPIC). Each of these codes can be used to explore multiple physical phenomena; however, the APEX descriptions indicate that the scale of the computational jobs is consistent per pipeline, and the I/O requirements are approximately fixed for each pipeline. In order to generate computational jobs approximating the provided workload, we have constructed a random workload generator that creates 60 workflows that preserve the workflow percentages and the job size distribution described by the APEX workflows. We do not vary the checkpoint sizes, though we are aware that some codes simulate varying levels of entropy over time.

Finally, we have also simplified our workload evaluation by only simulating checkpoint/restart dumps. Note that the APEX authors also described some of the additional outputs for analysis data and emphasize that analysis data is scientifically valuable in and of itself. Therefore a burst buffer may also benefit the input/output of analysis data; however, even without a fast storage subsystem, analysis data will be generated and stored. Further, the workflows described for analysis data are much more complicated than checkpoint/restart interactions, and thus its not clear that our efforts could adequately characterize interactions for scientific analysis data. Because the authors were members of the APEX Workflows team, our future work will include improving the analysis workflows to further improve our simulation results.

3.1.1 Basic Simulator Execution

The simulation begins by selecting a job from the queue of available jobs. As described above, each job includes requirements for the number of processors, the requested runtime, and the percentage of memory required to create a checkpoint. Job's are scheduled onto the simulated cluster using a first-fit algorithm. While simple, first-fit achieves a high degree of system utilization and is appropriate when we are not evaluating scheduling efficiency metrics such as job turnaround time.

The jobs are then simulated as a series of compute cycles followed by checkpoint phases that store data into the burst buffer system. Checkpoints are created at the near-optimal interval as described by Young and Daly (Daly 2006). As enforced by policy on Trinity, no job is allowed to request a "walltime" longer than 24 hours, so pipelines that require weeks of system time must be decomposed into shorter running jobs. While the first job of a campaign pipeline does not restore itself from an existing checkpoint, all subsequent jobs in that pipeline must read the most recent checkpoint file and restart from that point in the simulation.

We also provide a Poisson-based fault generation process that interrupts system nodes (both compute and burst buffer) causing any process currently using that node to fail. In the case of a failed burst buffer node, the checkpoint process fails; however, the application process continues and will attempt to write an additional checkpoint in the future. When a compute node fails, the entire application must stop, and a job is immediately inserted at the head of the queue to attempt to recoup the progress from the last checkpoint. We used the empirically observed mean time to interrupt (MTTI) from the Trinity supercomputer to parameterize the random arrival process with each node having an equivalent probability of failure.

3.2 Shared Burst Buffer Simulation

For shared burst buffers, simulation of flows both in and out of the burst buffer are critical. Each burst buffer polls for incoming requests. For each request received, it starts an *executor* process to handle that request. The processing time of a request is computed using the minimum of the burst buffer bandwidth and the compute node bandwidth. A burst buffer node's bandwidth is affected by the number of its concurrent requests; a compute node's bandwidth is affected by the number of its burst buffer partitions. A burst buffer node will notify all of its active executors that there is a change in its bandwidth when either a new executor begins or an existing executor finishes. As compute nodes write to multiple burst buffer nodes they split their available bandwidth evenly across them. When any write finishes, the available bandwidth is re-partitioned across the remaining writes and those burst buffers are informed that additional client bandwidth is available which may cause those writes to speed up if those burst buffers have available excess bandwidth.

3.3 Local Burst Buffer Simulation Flow

The local burst buffer simulation is very similar to the shared burst buffer simulation, but with a much simpler implementation. There are no separate burst buffer nodes; instead each compute node has an SSD whose bandwidth comes from the input parameter. When there is a fault, this fault will bring down both the compute node and the burst buffer on that node.

3.4 Simulation Configuration

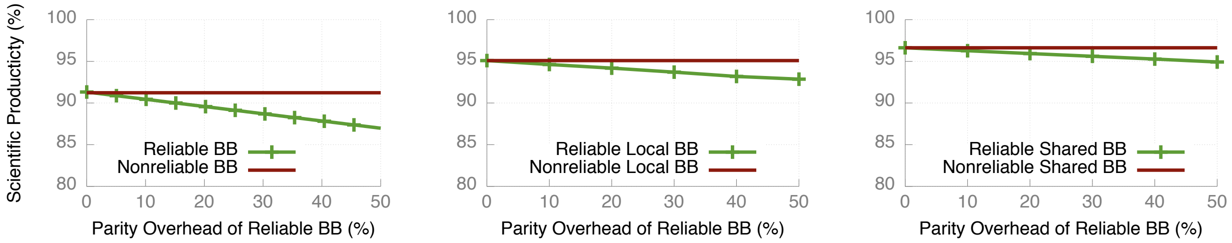
Table 2: Simulation parameters based on LANL's Trinity Supercomputer. Parameters which were varied in order to study their effects are in red italics.

Number of Compute Nodes	9500
Compute Node Memory	128 GB
Compute Node Network Bandwidth	15 GBs
Burst Buffer Type	Local | Shared
Number of Burst Buffer Nodes	If local 9500; if shared 276
Aggregate Burst Buffer Read Bandwidth	1600 GBs
Aggregate Burst Buffer Write Bandwidth	1400 GBs
Parity Overhead	0 | 10 | 20 | 30 | 40 | 50
Burst Buffer Stripes per File	If local 1; if shared 2
Fault Rate	One node per day
Node Recovery Time	1 second | 1 hour

The specification for all simulated shared and local burst buffer architectures is listed in Table 2. Aggregate burst buffer performance is based on empirical measurements using IOR (Various 2016). Similarly, the fault rate is based on the measurement of Trinity's mean time to interrupt (23.8 hours). In order to compare local and shared burst buffer architectures fairly, our local burst buffer simulation splits the aggregate burst buffer bandwidth evenly between each compute node.

4 RESULTS

As discussed in Section 1, our goal in building a burst buffer simulator is to explore various hardware and software design alternatives. Within this study we are limiting ourselves to an exploration of just two possible burst buffer architectures: node-local and shared I/O nodes; however, even within that constraint we have identified several interesting areas of evaluation. In particular we are interested in examining the system impacts related to three different factors. We first examine the impact of whether data in burst buffers is protected with parity, then examine the impact of how long failed nodes take to recover, and finish our evaluation by examining the impact of whether burst buffers are shared or local. The parameters studied are those shown in italicized red in Table 2.



(a) Analytical Model (b) Simulated Local BB (c) Simulated Shared BB

Figure 1: These 3 graphs show the percentage of time dedicated to checkpoint/restart for different burst buffer reliability overheads. Graph (a) shows the analytical results derived in an earlier work, while (b) and (c) show the checkpoint overheads for local and shared burst buffers, respectively. We can see that the simulation results support the earlier analytical result in finding that unreliable burst buffers reduce the overhead related to checkpoint/restart.

4.1 To Protect or Not To Protect

In this portion of our study, we explore the effects of using parity to protect data within the burst buffers. We use BBSim to revisit our earlier analytical study (see Section 2) which showed that unreliable burst buffers (i.e. those that do not add parity protection to data) outperform reliable burst buffers because the performance overhead of adding parity protection outweighs the reliability benefits.

To measure performance, we use the metric of *scientific productivity* which is the amount of time that a job makes forward progress divided by the total amount of time that the job is running. When measuring time spent making forward progress, we exclude checkpointing, restarting, and work that was redone due to failure recovery. Time spent checkpointing and restarting is work required only because the platform is unreliable. The application developers have no interest in checkpoint/restart; rather it is required because long-running simulations cannot complete in the time between job interruptions. Because checkpoint/restart is such a popular technique, some scientists have resorted to analyzing checkpoint files as a sort of crude analysis data, but our current simulation studies are ignoring the costs related to analysis data.

Similarly, we must consider time spent reading the checkpoint data in order to restore the application state as pure overhead. This is slightly unfair as LANL's HPC administrators enforce a scheduler policy that limits jobs to 24 hours of wall clock time for a variety of reasons (fair-sharing, routine maintenance, etc.), and thus restoring from checkpoints is mandatory for any scientific simulation that runs for longer than 24 hours. However, from a pure scientific productivity standpoint we cannot consider time spent reading the most recent checkpoint as directly advancing scientific progress.

Finally, we exclude re-work. Re-work occurs when a job experiences a fault and must be restarted from a prior checkpoint. Although re-work is valid scientific simulation work, it is overhead required only due to the unreliability of the platform and the discrete nature of restoring progress via checkpoint/restart.

In our earlier analytical work, we noted that given a fixed quantity of storage resources, the system could be designed to dedicate some of the resources to storing parity data to improve the system reliability at the cost of some storage system performance (e.g. a 10% parity overhead would result in a 10% storage system slowdown). Although an unreliable storage system will lead to lost checkpoints that result in larger quantities of re-work, the models found that checkpoint overhead was lowest with an unreliable burst buffer due to the much higher incidence of checkpointing to re-work. In Figure 1(a) we present the results from that analysis along with simulation experiments that examine the reliability overheads for local and shared

Table 3: Overall platform efficiency for burst buffer configurations using a 1 second repair/recovery time and a 1 hour repair/recovery time.

Simulation Configuration	Local, 1 Second	Local, 1 Hour	Shared, 1 Second	Shared, 1 Hour
Total Campaign Node-Hours (In Thousands)	18442.563	18696.147	18831.216	18453.385
Productive Compute Node-Hours (In Thousands)	16216.593	16215.224	16447.527	16448.516
Platform Productivity (%)	87.9	86.7	87.3	89.1

burst buffers (Figures 1(b) and 1(c) respectively). The earlier models assumed shared burst buffers; with our simulator, we can now reinforce that finding and extend it to local burst buffers as well: burst buffers, whether local or shared, should not add parity protection for checkpoint data.

Compared to the analytical results, the simulations find higher scientific productivity due to the smaller percentage of memory stored in each checkpoint for the simulated workloads compared to the assumptions included in the analytical models. Note that unlike the analytical study, our simulation results are unable to explore the effects of a reliable burst buffer with 0% reliability overhead, and thus the lines do not cross.

4.2 Effects of Repair/Recovery Time on Overall Productivity

We explore the repair time of nodes because current heuristics to estimate machine productivity have ignored this parameter (essentially assuming that repairing a failed node takes no time). This metric, called $JMTTI/Delta$, is used when procuring HPC platforms and measures the ratio between the mean time to interrupt for a job running across the entire machine and the time to checkpoint that job. In general, a $JMTTI/Delta$ ratio in excess of 200 results in an estimated platform productivity greater than 90%. That is, even in the face of faults causing job restarts and lost work, a job running across the entire machine should make forward progress 90% of the time when $JMTTI/Delta$ is greater than 200.

Although this simple metric is useful in comparing multiple vendor offerings during a procurement cycle, we have long recognized that it is an incomplete picture of the overall scientific productivity of an HPC platform. In addition to other simplifications, it excludes time lost due to repairing failed nodes. However, typical large HPC platforms can have repair/replace times of up to one hour. Therefore, we wanted to check whether $JMTTI/Delta$ can still achieve 90% platform productivity even with repair times up to an hour.

Table 3 shows the results of our simulation studies using a simple model that allows failed nodes (whether a compute node, burst buffer node, or both) to be repaired following a fault in either 1 second or 1 hour. For local burst buffers, we see that productivity decreases slightly with longer repair times as expected. For shared burst buffers, we see a surprising result that productivity *increases* with longer repair times. Upon closer examination, we discovered that this surprising result was due to randomness within our simulator; specifically, faults within our simulation of 1 hour repair times happened unluckily to disproportionately affect larger jobs causing a larger quantity of total cost work. In other words, this unexpected increase in efficiency is due to noise in the simulator.

We therefore generally conclude that the efficiency differences are not very sensitive to recovery time, and rather the mix of jobs and the mean time to interrupt seem to have dominating impacts on overall platform efficiency. In future workloads, with a majority of jobs using extremely large allocations it is likely that we would need to revisit this analysis, but these simulation results indicate that our current $JMTTI/Delta$ procurement metric is likely sufficient.

Table 4: Job checkpoint bandwidths varying both the burst buffer architecture (shared vs. local) and the storage system reliability (unreliable vs. 20% parity overhead.

Simulation Configuration	Local, Unreliable	Local, 20% Parity	Shared, Unreliable	Shared, 20% Parity
Min Application Checkpoint Bandwidth (GB/s)	18.0	14.8	36.1	28.8
Max Application Checkpoint Bandwidth (GB/s)	1424.8	1139.8	1080.4	884.6
Mean Overall Campaign Checkpoint Bandwidth (GB/s)	206.8	165.6	614.54	485.0
Median Overall Campaign Checkpoint Bandwidth (GB/s)	184.8	147.4	645.3	510.6

4.3 To Share or Not to Share

Finally, we examine the effects of organizing the burst buffer into a set of node-local storage devices or as a collection of shared, dedicated burst buffer nodes. In Table 4 we see that the average checkpoint bandwidth encountered throughout the campaign is approximately 3x greater for a shared burst buffer. This is as expected; jobs running with local burst buffers get the exact same percentage of the burst buffer nodes as they do the compute nodes whereas jobs using a small percentage of the compute nodes can use a larger percentage of the shared burst buffer nodes. A more complicated, and interesting, result is that the maximum bandwidth achieved with local burst buffers is higher than the maximum bandwidth achieved with shared. This is because full system jobs using local burst buffers have no resource contention. Our simulation chooses two random burst buffer nodes for each compute node in the shared burst buffer model; this results in some burst buffer nodes being overloaded. The lesson from this result is that shared burst buffers are better for HPC systems that run many small jobs and that shared burst buffers for systems running large jobs need careful mechanisms to reduce contention.

5 CONCLUSION

In this paper we presented BBSim, a simulator for exploring the various effects on scientific productivity arising from the organization of burst buffer storage system. To generate results that go beyond analytical models we used the APEX workflows document to construct a simulation workload derived from real data center workloads. With BBSim we have determined that for the presented workload repair/recovery time following a fault does not have a significant impact on overall platform efficiency. However, in confirmation of earlier analytically derived results, we determined that unreliable burst buffers result in less checkpoint/restart overhead compared to reliable burst buffer configurations. Further, our simulation results validated this conclusion for both shared burst buffer and local burst buffer configurations. Finally, we determined that shared burst buffers result in better overall application checkpoint bandwidth, providing an average storage system bandwidth 3.5x greater than a similarly configured local burst buffer configuration.

In addition to the results published here we have used BBSim to examine the scientific productivity delivered by future system configuration, including hypothetical configurations not currently fielded. BBSim is one of many steps to move our storage system procurement process toward a quantitative evaluation of scientific productivity. The addition of the simulation of analysis and visualization data set creation and access, as well as more robust simulation of the storage system software are two of the key remaining factors aspects needed to gain greater insight into the effects of storage systems to overall scientific productivity.

ACKNOWLEDGMENTS

This work is funded by the RAMSES project, Office of Advanced Computing Research, U.S. Department of Energy and the Ultrascale Systems Research Center (USRC) at Los Alamos National Laboratory supported by the U.S. Department of Energy contract DE-FC02-06ER25750.

This publication has been assigned the Los Alamos National Laboratory identifier LA-UR-17-20080.

REFERENCES

Bent, J., B. Settlemyer, N. DeBardeleben, S. Faibish, D. Ting, U. Gupta, and P. Tzelnic. 2015, July. "On the Non-Suitability of Non-Volatility". In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. Santa Clara, CA, USENIX Association.

Bent, J., B. Settlemyer, and G. Grider. 2016. "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage". *;login: The USENIX Magazine* vol. 41 (2), pp. 34–39. An optional note.

Carothers, C. D., D. Bauer, and S. Pearce. 2000. "ROSS: A High-performance, Low Memory, Modular Time Warp System". In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS '00, pp. 53–60. Washington, DC, USA, IEEE Computer Society.

Daly, J. T. 2006. "A higher order estimate of the optimum checkpoint interval for restart dumps". *Future Gener. Comput. Syst.* vol. 22 (3), pp. 303–312.

Harms, K., H. S. Oral, S. Atchley, and S. S. Vazhkudai. 2016, September. "Impact of Burst Buffer Architectures on Application Portability". Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF).

Kimpe, D., K. Mohror, A. Moody, B. Van Essen, M. Gokhale, R. Ross, and B. R. de Supinski. 2012. "Integrated In-system Storage Architecture for High Performance Computing". In *Proceedings of the 2Nd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '12, pp. 4:1–4:6. New York, NY, USA, ACM.

LANL, NERSC, and SNL. 2016, March. "APEX Workflows". Technical report, Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratory (SNL).

Liu, N., J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. "On the role of burst buffers in leadership-class storage systems". In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11. IEEE.

Matloff, N. 2008. "Introduction to discrete-event simulation and the simpy language". *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August* vol. 2, pp. 2009.

Peng, J., S. Divanji, I. Raicu, and M. Lang. 2016. "Simulating the Burst Buffer Storage Architecture on an IBM BlueGene/Q Supercomputer".

Thain, D., J. Bent, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. 2003, June. "Pipeline and Batch Sharing in Grid Workloads". In *Proceedings of High-Performance Distributed Computing (HPDC-12)*. Seattle, Washington.

Various 2016. "IOR HPC Benchmark".

Wang, T., S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu. 2014, Oct. "BurstMem: A high-performance burst buffer system for scientific applications". In *Big Data (Big Data), 2014 IEEE International Conference on*, pp. 71–79.

AUTHOR BIOGRAPHIES

LEI CAO is a post master student at New Mexico Consortium. He holds a M.S. degree in Electrical and Computer Engineering from Carnegie Mellon University. His research interest is in HPC storage systems. His email address is leicao88124@lanl.gov

BRADLEY W. SETTLEMYER is a storage systems research and systems programmer specializing in high performance computing. He received his Ph.D in computer engineering from Clemson University in 2009 and works as a research scientist in Los Alamos National Laboratory's HPC Design group. He has published papers on emerging storage systems, long distance data movement, network modeling, and storage systm algorithms. His email address is bws@lanl.gov

JOHN BENT is the Chief Architect at Seagate Government Solutions and has researched storage and IO throughout his career. His recent focus is on parallel storage systems for High Performance Computing. He holds a Ph.D in computer science from the University of Wisconsin-Madison.

A FRAMEWORK FOR UNIT TESTING WITH COARRAY FORTRAN

Ambra Abdullahi Hassan
University of Rome Tor Vergata
Rome, Italy
ambra.abdullahi@uniroma2.it

Valeria Cardellini
University of Rome Tor Vergata
Rome, Italy
cardellini@ing.uniroma2.it

Salvatore Filippone
Cranfield University
Cranfield, UK
salvatore.filippone@cranfield.ac.uk

ABSTRACT

Parallelism is a ubiquitous feature of modern computing architectures; indeed, we might even say that serial code is now automatically legacy code. Writing parallel code poses significant challenges to programs, and is often error-prone. Partitioned Global Address Space (PGAS) languages, such as Coarray Fortran (CAF), represent a promising development direction in the quest for a trade-off between simplicity and performance. CAF is a parallel programming model that allows a smooth migration from serial to parallel code. However, despite CAF simplicity, refactoring serial code and migrating it to parallel versions is still error-prone, especially in complex softwares. The combination of unit testing, which drastically reduces defect injection, and CAF is therefore a very appealing prospect; however, it requires appropriate tools to realize its potential. In this paper, we present the first CAF-compatible framework for unit tests, developed as an extension to the Parallel Fortran Unit Test framework (pFUnit).

Keywords: Coarray Fortran; Test-Driven Development; Unit tests; pFUnit; Refactoring

1 INTRODUCTION

Scientific software tends to have rather peculiar characteristics (Carver, Kendall, Squires, and Post 2007): its requirements gathering process is unique in that nature often plays a prominent role, it is often the outcome of research projects with little or no development planning, it is often written by scientists whose primary research interest is not software-related, and scientific software packages tend to have very long lifetimes.

Scientific software development then is rarely accompanied by techniques such as Test-Driven Development (TDD) that are very frequent in other application areas, nor is it often the case that (semi-)automated tools are employed. Many authors have advocated the use of disciplined design strategies in this context, see e.g., (Rouson, Xia, and Xu 2011).

The introduction of parallel features into existing serial codes is a necessary step in today's world; unfortunately, parallel programming techniques open many opportunities to introduce subtle bugs and puzzling

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

behavior in the software under consideration. Therefore, we deem appropriate to also introduce development strategies to keep these risks under control and to increase our confidence in the numerical results produced by the software.

1.1 Parallelization and PGAS Languages

With the proliferation of multicore processors and many-core accelerators, any serial code is automatically "legacy", and ought to be parallelized in order to perform effectively (Radhakrishnan, Rouson, Morris, Shende, and Kassinos 2013). The Message Passing Interface (MPI) offers a rich set of functionalities for parallel applications on distributed systems, but parallelization must be handled manually by the programmer, and taking care of very low-level data transfer details. The Partitioned Global Address Space (PGAS) parallel programming model is an effective and interesting alternative that combines the advantages of the Single Program Multiple Data (SPMD) programming style for distributed memory systems with the data referencing semantics of shared memory systems. Unified Parallel C (UPC) (UPC Consortium 2005), Coarray Fortran (CAF) (Numrich and Reid 1998), Chapel (Chamberlain, B.L. 2015), X10 (Saraswat, Bloom, Peshansky, Tardieu, and Grove 2012), SHMEM are all based on the PGAS model. In the Fortran case, this model has been integrated in the Fortran 2008 standard with the introduction of coarrays.

CAF communications are largely abstracted at a much higher level than in MPI; moreover, code written in CAF can easily work on both shared and distributed memory architectures. Additionally, CAF syntax allows to express a parallel algorithm in a simpler style. CAF has been tested on different scientific applications (Ashby and Reid 2008), (Hasert, Klimach, and Roller 2011), (Cardellini, Fanfarillo, and Filippone 2016) and it has been shown that CAF code efficiency is comparable to that of MPI code. In the trade-off between readability and efficiency, we think that CAF is a winning choice when parallelizing a scientific legacy code.

1.2 Paper Contributions

However, CAF currently lacks the support of unit testing tools and this makes the developer task harder. The goal of our paper is to fill this gap and to provide a contribution toward the test-driven development of scientific applications written in CAF, in such a way to make the development schedule much more predictable. To this end, we extend pFUnit, the parallel Fortran Unit testing framework, with the support for CAF, thus contributing with a framework for unit testing with Coarray Fortran.

The rest of the paper is organized as follows. In Section 2 we provide some background on CAF and testing of scientific software, including TDD and pFUnit. In Section 3 we describe the proposed extension of pFUnit that supports CAF, also providing an example of testing a simple CAF code with pFUnit. In Section 4 we describe a case study of unit testing related to the migration from MPI to CAF of PSBLAS, a library of Basic Linear Algebra Subroutines for parallel sparse applications. We conclude with Section 5 providing some hints for future work.

2 BACKGROUND

2.1 Coarray Fortran

Coarray Fortran achieves a good trade-off between readability and performance: it introduces minimal syntactic extensions, it allows for very clear specification of data movement, and it allows a smooth path for code migration and incremental parallelization. Compared with the parallelization of application under

MPI, the parallel features are easier to follow, the code tends to be far shorter, and the impact of the parallel statements is much smaller.

CAF started as an extension of Fortran for parallel processing and is now part of the Fortran 2008 standard. CAF adopts the PGAS model for SPMD parallel programming, where multiple "images" share their address space; the shared space is partitioned, and each image has a local portion. The standard is very careful not to constrain what an "image" should be, so as to allow different implementations to make use of underlying threads and/or processes. From a logical, user-centered point of view, each CAF program is replicated across images, and the images execute independently until they reach a synchronization point, either explicit or implicit. The number of images is not specified in the source code, and can be chosen at compile, link or run-time, depending on the particular implementation of the language. Images are assigned unique indices through which the user can control the flow of the program by conditional statements, similar to common usage in MPI. Coherently with the language default rules, image indices range from 1 to a maximum index which can be retrieved at runtime through an appropriate intrinsic function `num_images()`. Each image has its own set of data objects; some of these objects may also be declared as coarray objects, meaning that they can be accessed by other images as well. Coarrays are declared with a so-called codimension, indicated by square brackets. The codimension spans the space of all the images:

```
integer :: i[*]
real :: a(10)[*]
real :: b(0:9)[0:4,*]
```

As already mentioned, one of the main advantages of CAF over MPI is its simplicity. Much of the parallel bookkeeping is handled behind the scenes by the compiler, and the resulting parallel code is shorter and more readable; this reduces the chances of injecting defects in the code. As an example, let us consider the burden of passing a non-contiguous array (for example, a matrix row in Fortran) using `MPI_DATATYPE`:

```
! Define vector
call MPI_Type_vector(n,nb,n,MPI_REAL, myvector,ierr)
call MPI_Type_commit(myvector, ierr)
! Communicate
if(n_rank == 0) then
   call MPI_Send(a,1,myvector,1,100, MPI_COMM_WORLD, ierr)
endif
if(n_rank == 1) then
   call MPI_Recv(a,1,myvector,0,100, MPI_COMM_WORLD, mystatus, ierr)
endif

call MPI_Type_free(myvector, ierr)
```

All of the above code is equivalent to just a single CAF statement:

```
a(1,:)[2]=a(1,:)[1]
```

with no need to write separate code for sending and receiving a message.

Currently, the number of compilers implementing CAF has increased: the Intel and Cray compilers support it. The GCC compiler provides CAF support via a communication library; the base GCC distribution only handles a single image, but the OpenCoarrays project provides an MPI-based and a GASNet-based communication library (Fanfarillo, Burnus, Cardellini, Filippone, Nagle, and Rouson 2014).

2.2 Unit Test and Test-Driven Development

Software testing can be applied at different levels: *unit tests* are fine-grained tests, focusing on small portions of the code (a single module, function or subroutine) (Osherove 2015), while *regression tests* are coarse-grained tests, that encompass a large portion of the implementation and are used to verify that the software still performs correctly after a modification.

Many scientific softwares rely on regression, disregarding unit tests. This choice presents several disadvantages, spanning from the impossibility to perform verification in the early stages of software's lifecycle, to difficulties in locating the error responsible for failure, and to a long time required to run tests.

Additionally, the search for performance through parallelism increases complexity of the code and may cause new classes of errors: race conditions and deadlocks are two such examples.

Race conditions occur when multiple images try to access concurrently the same resource. For example, in the code fragment:

```
count[1]=0
do i=1 , n
  count[1] = count[1] + 1
enddo
```

the final value of the variable `count[1]` is not uniquely determined by the code, and depends upon the order in which the various images execute the statements.

A deadlock occurs when an image is waiting for an event that cannot possibly occur, e.g., when two images wait for each other; as a consequence, the program makes no further progress. With CAF a deadlock occurs in the following example:

```
if (this_image == 1) then
   sync images(num_images())
endif
```

because image 1 is waiting for synchronization with the last image, but the last image is not executing the matching `sync` statement.

These new classes of errors enforce the need for systematic unit testing.

Test-Driven-Development (TDD) is a software development practice relying strongly on unit tests. TDD combines a test-first approach with refactoring: before actually implementing the code, the programmer writes automated unit tests for the functionality to be implemented.

Writing unit tests is a time-consuming process, and if the associated effort is perceived to be excessive, the programmer may be inclined to reduce or skip it altogether. Unit testing frameworks are tools intended to help the developers to write, execute and review tests and results more efficiently. Many testing frameworks exist; they are often called "xUnit" frameworks, where "x" stands for the (initials of the) name of the language they are developed for.Usually, a unit testing framework provides code libraries with basic classes, attributes and assert methods; it also includes a test runner used to automatically identify and run tests, and provides information about their number, failures, raised exceptions, location of failures. Good testing frameworks are critical for the acceptance of TDD.

2.3 pFUnit: A Unit Testing Framework for Parallel Fortran

Given the above discussion, it should by now be clear that a unit testing framework for Coarray Fortran applications is a desirable tool. A good basis for a CAF compatible unit testing framework ought to have certain characteristics. First, it should be conceived for the Computational Science & Engineering (CSE) and HPC development communities. The ideal framework should also handle parallelism and should be able to detect the peculiar errors caused by concurrently execution of different images. Additionally, it must be easy to extend, for example through object-oriented (OO) features.

pFUnit from NASA (Clune and Rood 2011) is a tool that satisfies all of these requirements. pFUnit is a unit testing framework implemented by keeping in mind the open source xUnit family, and is developed in Fortran 2003, using OO design techniques. While patterned after Junit, pFUnit is tailored to the CSE/HPC environment and supports comparison of single/double precision quantities with optional tolerance and test for infinity and *NaN* (thus permitting to check for subtle numerical issues that can affect result quality). Moreover, the available support for parallelism and its OO structure, make it an ideal starting point to start for the creation of a CAF-compliance unit testing framework.

The simplest way to write tests using pFUnit is through a preprocessor input file. The preprocessor input file is not written in standard Fortran and has extension ”$.pf$”. It is a Fortran free format file with preprocessor directives added. The parser automatically generates one test suite per file/module, and suite names are derived from the file or module containing the tests. Once the preprocessor is invoked, it generates a Fortran file that when compiled and linked with pFUnit will provide tests subroutine. Finally, pFUnit provides a driver, that is, a short program that bundles all of the test suites, runs the tests and produces a short summary.

3 EXTENDING PFUNIT WITH CAF SUPPORT

We now present how we provide support for Coarray Fortran in pFUnit. To this end, we created a set of CAF classes, extending those already available inside pFUnit. They are shown in Figure 1 and include:

CafTestCase This class allows a single test procedure to be executed multiple times with different input values. By analogy with MpiTestCase, it is a special subclass of the more general ParameterizedTestCase and allows to run tests using pFUnit custom support for parameterized tests.

CafTestParameter This class provides procedures for setting/getting the number of images for the running test and ensures that image causing the failure is correctly reported.

CafTestMethod This class permits test fixture by specifying setUp() and tearDown() methods.

CafContext This class provides some communication routines such as gather and reduce.

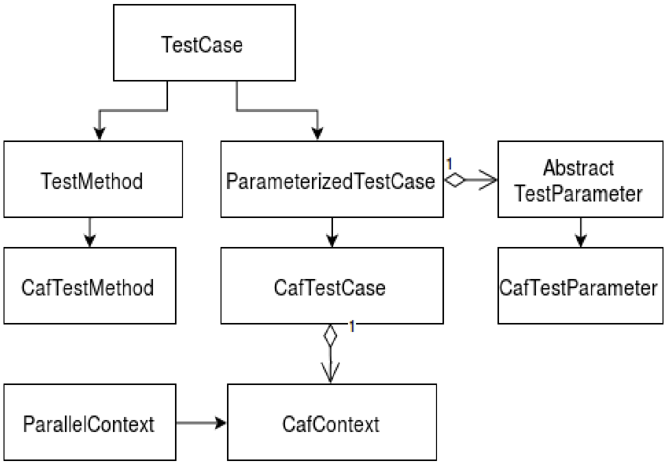


Figure 1: CAF classes inside pFUnit.

In the `CAF_context` class, some collective operations need to be defined. Collectives for coarrays have been proposed and scheduled for inclusion in a future revision of the Fortran standard, but are not currently available in many compilers. For this reason a simple `co_sum` function and a logical reduction, as well as some gather subroutines are provided in this class.

The implementation of these collectives is not fully optimized, since they are intended to be superseded by the runtime of the compiler, but this is not too restrictive since they will will run only once for each test suite.

From the user perspective, CAF tests have a single, mandatory argument of type CafTestCase and with `intent(INOUT)`. It provides two methods `getNumImages()` and `getImageRank()`, returning the total number of images running for the test and the rank of the current image. In CAF, test failing assertions of any types provide information about the rank of the process (or the ranks of the processes) that detects the failure. This permits to recover not only on the test that has failed, but on the specific image that causes the failure.

3.1 An Example of CAF Unit Test Using pFUnit

Let us now consider a simple example of testing a CAF code with pFUnit. The example has been run on a Linux laptop, using GCC 6.1.0, MPICH 3.2, OpenCoarrays 1.6.2 and our development version of pFUnit. Let us assume that we have an application where the CAF images are organized in a linear array, and our computations need to deal with elements of an index space. Let us also assume that the amount of computation per point in the index space is constant; naturally, we want our workload to be distributed as evenly as possible. A possible solution would be to use the following routine which determines, for each image, the size of the local number of indices and the first index assigned to it:

```fortran
subroutine data_distribution(n,il,nl)
  integer, intent(in)  :: n
  integer, intent(out) :: il, nl
  integer :: nmi
  ! Compute first local index IL and number of local indices NL. We want the data to
  ! be evenly spread, i.e., for all images! NL must be within 1 of n/num_images()
  ! For all images we should have  that IL[ME+1]-IL[ME]=NL
  associate(me=>this_image(),images=>num_images())
    nmi = mod(n,images)
    nl = n/images  + merge(1,0,me<=nmi)
    il = min(n+1,merge((1+(me-1)*nl), &
         & (1+ nmi*(nl+1) + (me-nmi-1)*nl),&
         & me <= nmi))
  end associate
end subroutine data_distribution
```

It is easy to see that on `NP` images, this routine will assign to each image either `N/NP` indices or `(N/NP)+1`, as necessary to make sure the indices add up to `N`. To check that our routine is working properly with pFUnit we need to specify clearly which properties our data distribution is supposed to have; in our case we have:

1. The local sizes must add up to the global size `N`;
2. The local sizes must differ by at most 1 from `N/NP`;
3. For each image `me<num_images()`, `il[me+1]-il[me]` must be equal to `nl`.

Checking the first property is very simple when the underlying compiler (like GNU Fortran 6.1.0) supports the collective intrinsics:

```fortran
  @test(nimgs=[std])
  subroutine test_distribution_1(this)
    implicit none
    Class(CafTestMethod), intent(inout) :: this
    integer, parameter :: gsz=27
    integer :: ngl, info
    integer, allocatable  :: il[:], nl[:]
    allocate(il[*],nl[*],stat=info)
    @assertEqual(info,0,"Failed␣allocation!")
    ! Set up checks
    associate(me=>this_image(),images=>num_images())
      call data_distribution(gsz,il,nl)
      ! Build reference data to check against
      ngl = nl
      call co_sum(ngl)
      @assertEqual(ngl,gsz,"Sizes␣do␣not␣add␣up!")
    end associate
  end subroutine test_distribution_1
```

The macro `@test(nimgs=[std])` indicates that the test we are running is a CAF test that runs on all available images. The `this` argument is mandatory and must have `intent(inout)`; the assertion `@assertEqual` is used to verify that the actual result matches the expected output.

The coarrays remote access facilities make it very easy to check for consistency across process boundaries. Since the OpenCoarrays installation we are using is built on top of MPICH, we execute the tests with the `mpirun` command, as shown in Figure 2. The output gives us confidence that the data distribution is computed correctly; the run with 15 processes tells us that the border case where we have more processes than indices (in our case 13) is being handled correctly. Since the test has been run on an quad-core laptop, running 15 MPI processes has a substantial overhead, which is entirely normal; the run with 4 processes was very fast, as expected.

```
[localhost CAF_pFUnit] mpirun -np 4 ./testCAF
...
Time:          0.004 seconds
 OK
 (3 tests)
[localhost CAF_pFUnit] mpirun -np 15 ./testCAF
...
Time:          6.207 seconds
 OK
 (3 tests)
```

Figure 2: Test output

What happens if there is an error? To demonstrate this, we wrapped the `data_distribution` routine injecting two errors on image 2: we alter both the local number of indices as well as the starting index. The test output is shown in Figure 3. We get a fairly precise indication of what went wrong and where:

- An error on all images because the total size does not match the expected value;
- An error on image 2 because the local number of indices is not in the expected range `(N/NP):(N/NP)+1`;
- An error on images 1 and 2, because having injected an error in the start index on image 2 affects the checks on both of these images.

3.2 Limitations: Team Support

It is often desirable to be able to run tests using varying number of processes/images; this is because some bugs reveal themselves only when running on a certain number of processes. In pFUnit, the user is al-

```
[localhost CAF_pFUnit] mpirun −np 4 ./testCAF
.F.F.F
Time:         0.013 seconds

 Failure in: CAF_distribution_test_mod_suite.test_distribution_1[nimgs=4][nimgs=4]
   Location: [testCAF.pf:25]
Distribution does not add up! expected 13 but found: 17;  difference: |4|. (IMG=1)

..........

 Failure in: CAF_distribution_test_mod_suite.test_distribution_2[nimgs=4][nimgs=4]
   Location: [testCAF.pf:50]
One image is getting too many entries expected 4 to be less than or equal to: 1. (IMG=2)

 Failure in: CAF_distribution_test_mod_suite.test_distribution_3[nimgs=4][nimgs=4]
   Location: [testCAF.pf:81]
Start indices not consistent expected 4 but found: −4;  difference: |8|. (IMG=1)

 Failure in: CAF_distribution_test_mod_suite.test_distribution_3[nimgs=4][nimgs=4]
   Location: [testCAF.pf:81]
Start indices not consistent expected 7 but found: 11;  difference: |4|. (IMG=2)

 FAILURES!!!
Tests run: 3, Failures: 3, Errors: 0
 there is an error
```

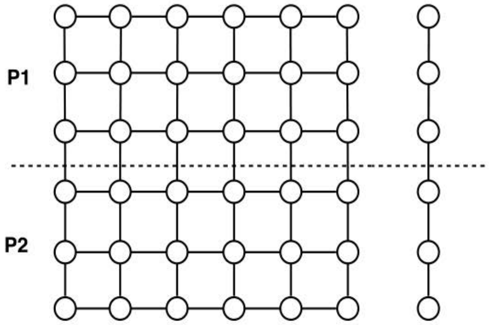
Figure 3: Test output with errors

lowed to control this aspect by passing an optional argument `nimgs=[<list>]` (in the case of CAF) or `nprocs=[<list>]` (in the case of MPI): in this way the test procedure will execute once for each item in `<list>`. MPI makes this possible by constructing a sub-communicator of the appropriate size for each execution. In principle, CAF allows for clustering of images in teams, which are meant to be similar to MPI communicators; at any time an image executes as a member of a team (the current team). Constructs are available to create and synchronize teams.

Unfortunately, at the time of this writing, the only compiler supporting teams is the OpenUH one (Khaldi, Eachempati, Ge, Jouvelot, and Chapman 2015). However, this compiler does not support other standard Fortran features; most importantly, it does not support the OO programming features, and therefore cannot be used with pFUnit. As a result, the current version of pFUnit only allows the CAF user-defined tests to be run with all images, and the only accepted values for the optional argument `nimgs` is `[std]`.

4 CASE STUDY: PSBLAS

We illustrate a case study that shows how pFUnit can be used to detect errors. We applied pFUnit to perform unit tests on the PSBLAS library (Filippone and Colajanni 2000) (Filippone and Buttari 2012) during the library migration from MPI to CAF. At the time of writing, we have written a total of 12 test suites and 241 unit tests. PSBLAS is a library of Basic Linear Algebra Subroutines that implements iterative solvers for sparse linear systems and includes subroutines for multiplying sparse matrices by dense matrices, solving sparse triangular systems, and preprocessing sparse matrices, as well as additional routines for dense matrix operations. It is implemented in Fortran 2003 and the current version uses message passing to address a distributed memory execution model. We converted the code gradually from MPI to coarrays, thus having MPI and CAF coexisting in the same code. In PSBLAS, we detected three communication patterns that had to be modified: 1) the halo exchange, 2) the collective subroutines, and 3) the point-to-point communication in data distribution. In PSBLAS, data allocation on the distributed-memory architecture is driven by the



Before Halo Exchange

P1			P2		
loc(i)	**glob(i)**	**val(i)**	**loc(i)**	**glob(i)**	**val(i)**
1	1	1	1	4	2
2	2	1	2	5	2
3	3	1	3	6	2
4	4	4	4	3	3

After Halo Exchange

P1			P2		
loc(i)	**glob(i)**	**val(i)**	**loc(i)**	**glob(i)**	**val(i)**
1	1	1	1	4	2
2	2	1	2	5	2
3	3	1	3	6	2
4	4	2	4	3	1

Figure 4: Example of halo exchange on 2 images.

discretization mesh of the original PDEs. Each variable is associated to one point of the discretization mesh. If $a_{ij} \neq 0$ we say that point `i` depends on point `j`. After the partition of the discretization mesh into subdomains, each point is assigned to a parallel process. An halo point is a point which belongs to another domain, but there is at least one point in this domain that depends on it. Halo points are requested by other domains when performing a computational step (for example a matrix-vector product) : every time this happens we perform an halo exchange operation that can be considered as a sparse all-to-all communication.

The subroutine `psb_halo` performs such an exchange and gathers values of the halo elements: we used our framework to unit test this procedure. We need to run the whole suite of tests multiple times if we want to change the number of images participating in the test. To avoid that, in each test we distributed variables not among all the available images, but only on a subset of them. In this way, while all tests run on the same number of images, communication actually takes place only between some of the images. Of course this has to be taken into account when asserting equality on all images (we have to do that, otherwise a deadlock can occur). In the following example, we test the halo exchange of a vector. The variables are distributed only between two images. When calling the `assert` statement, the expected solution check and the obtained result `v` are multiplied by `1` if the image takes part to the communication, `0` otherwise.

```
@test(nimgs=[std])
subroutine test_psb_dhalo_2imgs_v(this)
  implicit none
  Class(CafTestMethod), intent(inout) :: this
  integer :: me, true
  real(psb_dpk_), allocatable :: v(:), check(:)
  type(psb_desc_type):: desc_a
  !Distributing point, Creating input vector v and expected solution check.
  ...
  !Calling the halo subroutine
  call psb_halo(v, desc_a, info)
  @assertEqual(0,info, "ERROR␣in␣psb_halo")

  if ((me==1).or.(me==2)) then
    true = 1
  else
    true=0
  endif

  @assertEqual(true*check,true*v)

  !Deallocate free and exit psblas
  ...
end subroutine test_psb_dhalo_2imgs_v
```

When writing a test for a given functionality, the programmer should ensure to reach the maximum code coverage. This means to test all the implementations of a given interface and to consider all the possible branches. Let us consider, for example, the collective subroutine `psb_amx` in PSBLAS which implements a maximum absolute value reduction. By looking at its interface, we can see that for its testing we need at least 15 different unit tests.

```
interface psb_amx
  module procedure psb_iamxs, psb_iamxv, psb_iamxm, psb_samxs, psb_samxv, psb_samxm,&
       & psb_camxs, psb_camxv, psb_camxm,&
       & psb_damxs, psb_damxv, psb_damxm,&
       & psb_zamxs, psb_zamxv, psb_zamxm
end interface
```

Additionally, it admits one optional argument `root` indicating which image holds the final value. If `root = -1`, then the final result is shared among images, thus performing an all-reduce operation. This parameter leads to a branch in the code, thus doubling the number of unit tests needed.

```
if (root_ == -1) then
  ! All reduce
     ...
else
  ! Reduce, root_ is the root process
     ...
endif
```

Finally, we test the utility subroutine `psb_matdist` to distribute a matrix among images according to a user defined data distribution.

```
subroutine test_psb_dmatdist1(this)
  implicit none
  Class(CafTestMethod), intent(inout) :: this
  integer :: me, np, info, iunit=12, nv, i,&
       & nz, last, j, irow, icontxt
  type(psb_desc_type):: desc_a
  type(psb_dspmat_type) :: a, a_out
  integer, parameter :: m_problem = 10
  integer, allocatable :: ipv(:), ivg(:), ia(:),&
       & ja(:), ia_exp(:), ja_exp(:)
  real(psb_dpk_), allocatable :: val(:), val_exp(:),&
       & a_exp(:,:), a_aux(:,:)
  me = this_image()
  np = num_images()
  call psb_init(icontxt,np,MPI_COMM_WORLD)
  call mm_mat_read(a,info,iunit=iunit,&
       & filename="matrix1.mtx")
  allocate(ivg(m_problem),ipv(np))
  do i=1,m_problem
    call part_block(i,m_problem,np,ipv,nv)
    ivg(i) = ipv(1)
  enddo

  !Getting the expected solution
  call a%csgetrow(1,10,nz,ia,ja,val,info)
  allocate(ia_exp(nz),ja_exp(nz), val_exp(nz))
  last = 0
  do i=1, m_problem
    if (me == ivg(i) + 1) then
      irow=i
      do j=1, nz
        if (ia(j)  == irow) then
          last = last + 1
          ia_exp(last)=ia(j)
          ja_exp(last)=ja(j)
          val_exp(last)=val(j)
        endif
      enddo
    endif
  enddo
  if (allocated(a_exp)) deallocate(a_exp)
  allocate(a_exp(m_problem,m_problem))
  a_exp = 0.0d0
  do i=1,last
    a_exp(ia_exp(i),ja_exp(i))=val_exp(i)
  enddo

  !Test subroutine
  call psb_matdist(a, a_out, icontxt, &
       & desc_a,info, v=ivg)
  call a_out%csgetrow(1,m_problem,nz,ia,ja,val,info)
  !Convert to global indices
  call psb_loc_to_glob(ia, desc_a, info)
  call psb_loc_to_glob(ja, desc_a, info)
  if (allocated(a_aux)) deallocate(a_aux)
  allocate(a_aux(m_problem,m_problem))
  a_aux = 0.0d0
  do i=1,last
    a_aux(ia(i),ja(i))=val(i)
  enddo
  @assertEqual(a_aux,a_exp)

  !Free
  deallocate(a_aux, a_exp, ia, ja, val)
  deallocate(ipv, ivg, ia_exp, ja_exp, val_exp)
  call psb_spfree(a, desc_a, info)
  call psb_cdfree(desc_a, info)
  call psb_exit(icontxt)
end subroutine test_psb_dmatdist1
```

We use an auxiliary input file containing a matrix of size `m_problem` in the Matrix Market format. The test runs on all images and with all input matrices as long as the parameter `m_problem` is changed accordingly. We use a block partition distribution, through a call to `part_block`. It returns the vector `vg` of size m_problem: variable i belongs to process j if vg(i)=j. We use this vector to manually create the partition, building the matrix a_exp that represents the expected solution. We then create the partition through a call to the psb_mat subroutine, and we build an auxiliary local matrix a_aux. Finally, we check the correctness of the solution by asserting the equality of the two matrices.

5 CONCLUSIONS

Coarray Fortran (CAF) makes parallelism syntactically simpler than using MPI, and CAF code is much easier to write and maintain. We believe that CAF is particularly suitable for parallelization of legacy Fortran codes, where the trade-off between readability and performance is an issue.

Support tools for CAF programmers are rare; the inadequacy of common unit testing frameworks, for example, is a major difficulty in applying TDD techniques in the case of CAF code. To improve this situation, we have extended the existing Parallel Fortran Unit Testing framework (pFUnit) to support CAF. pFUnit has been proved to be well suited to this task thanks to its object-oriented architecture and its design tailored to the needs of the HPC and CSE communities.

Future work will include the enablement of the team support as soon as it becomes available in the underlying compiler(s), and the collection of more usage data in the context of new application development. We are currently in contact with the authors of pFUnit to arrange for general availability of our extensions.

ACKNOWLEDGMENTS

We wish to thank Dr. Tom Clune of NASA for giving us full access to the latest pFUnit source code, as well as for starting the pFUnit project in the first place.

REFERENCES

Ashby, J. V., and J. K. Reid. 2008. “Migrating a scientific application from MPI to coarrays”. In *Proc. of 2008 Cray User Group Conf.*, CUG ’08.

Cardellini, V., A. Fanfarillo, and S. Filippone. 2016. “Heterogeneous CAF-based load balancing on Intel Xeon Phi”. In *Proc. of 2016 IEEE Int’l Parallel and Distributed Processing Symposium Workshops*, IPDPSW ’16, pp. 702–711.

Carver, J. C., R. P. Kendall, S. E. Squires, and D. E. Post. 2007. “Software development environments for scientific and engineering software: A series of case studies”. In *Proc. of 29th Int’l Conf. on Software Engineering*, ICSE ’07, pp. 550–559, IEEE.

Chamberlain, B.L. 2015. “Chapel”. In *Programming Models for Parallel Computing*. MIT Press.

Clune, T. L., and R. B. Rood. 2011. “Software testing and verification in climate model development”. *IEEE Software* vol. 28 (6), pp. 49–55.

Fanfarillo, A., T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. 2014. “OpenCoarrays: Open-source transport layers supporting Coarray Fortran compilers”. In *Proc. of 8th Int’l Conf. on Partitioned Global Address Space Programming Models*, PGAS ’14, pp. 4:1–4:11, ACM.

Filippone, S., and A. Buttari. 2012. “Object-oriented techniques for sparse matrix computations in Fortran 2003”. *ACM Transactions on Mathematical Software* vol. 38 (4), pp. 23:1–23:20.

Filippone, S., and M. Colajanni. 2000. “PSBLAS: A library for parallel linear algebra computation on sparse matrices”. *ACM Transactions on Mathematical Software* vol. 26 (4), pp. 527–550.

Hasert, M., H. Klimach, and S. Roller. 2011. “CAF versus MPI-applicability of coarray Fortran to a flow solver”. In *Recent Advances in the Message Passing Interface*, EuroMPI ’11, pp. 228–236, Springer.

Khaldi, D., D. Eachempati, S. Ge, P. Jouvelot, and B. Chapman. 2015. “A team-based methodology of memory hierarchy-aware runtime support in Coarray Fortran”. In *Proc. of 2015 IEEE Int’l Conf. on Cluster Computing*, CLUSTER ’15, pp. 448–451.

Numrich, R. W., and J. Reid. 1998. “Co-Array Fortran for parallel programming”. *ACM Sigplan Fortran Forum* vol. 17 (2), pp. 1–31.

Osherove, R. 2015. *The Art of Unit Testing*. MITP-Verlags GmbH & Co. KG.

Radhakrishnan, H., D. W. Rouson, K. Morris, S. Shende, and S. C. Kassinos. 2013. “Test-driven coarray parallelization of a legacy Fortran application”. In *Proc. of 1st Int’l Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pp. 33–40. ACM.

Rouson, D., J. Xia, and X. Xu. 2011. *Scientific Software Design*. Cambridge University Press.

Saraswat, V. and Bloom, B. and Peshansky, I. and Tardieu, O. and Grove, D. 2012. “The X10 language specification, v2.2.3”.

UPC Consortium 2005. “UPC language specifications, v1.2”. Technical Report LBNL-59208, Lawrence Berkeley National Lab.

AUTHOR BIOGRAPHIES

AMBRA ABDULLAHI HASSAN is a PhD student at the University of Rome Tor Vergata, Italy. Her research interests include high performance computing and software for computational linear algebra.

VALERIA CARDELLINI, PhD, is Associate Professor at the University of Rome Tor Vergata, Italy. Her research interests are in the field of distributed and parallel computing systems. She has published more than 80 papers in international conferences and journals, has served as TPC member of conferences and co-chair of workshops and participated in EU projects on IT topics, including EoCoE and Cost Action ACROSS.

SALVATORE FILIPPONE, PhD, is a Lecturer at Cranfield University, UK. His main research interests are in software development for High Performance Computing; he has served as TPC member of a number of international conferences, as a reviewer and evaluator of EU projects, and is Associate Editor for the ACM Transactions on Mathematical Software.

SCALING CONSTITUENT ALGORITHMS OF A TREND AND CHANGE DETECTION POLYALGORITHM

Rishu Saxena
Layne T. Watson

Departments of Computer Science
and Mathematics
VPI & SU, Blacksburg, VA 24061
rishus@vt.edu

Valerie A. Thomas
Randolph H. Wynne

Department of Forest Resources and
Environmental Conservation
VPI & SU, Blacksburg, VA 24061
thomasv@vt.edu

ABSTRACT

Earth observation satellites (EOS) such as Landsat provide image datasets that can be immensely useful in numerous application domains, by extracting information via time series analysis. While the literature is replete with algorithms, the size of the datasets itself is prohibitive, currently of the order of petabytes and growing, which makes them computationally unwieldy — both in storage and processing. An EOS image stack typically consists of multiple images of a fixed area on the Earth's surface (same latitudes and longitudes) taken at different time points. Meaningful time series analysis on one such interannual, multitemporal stack with existing state of the art codes can take several days on multicore servers. This work lays the foundation for a polyalgorithm based on two change detection algorithms, EWMACD and BFAST, for time series analysis of satellite image stacks, and presents speedup results for those two algorithms.

Keywords: Time series analysis, big data, change detection, parallel computing, load balancing.

1. INTRODUCTION

Land use and land cover change (LULCC) is of crucial importance globally. With anthropogenic activities such as deforestation and urbanization increasing exponentially through the past century, there have been significant changes in land cover in several parts of the world [8]. Simultaneously, significant changes in the global climate have also been observed, driven in part by LULCC (e.g., [7]). LULCC also has impacts on a wide variety of other ecosystem services. Much research is, therefore, being directed towards Earth monitoring.

Earth observation satellites (EOS) such as Landsat provide image datasets that, if harnessed well, can be immensely helpful towards LULCC monitoring. These images hold valuable information that can be very helpful in understanding and managing our natural resources.

Time series analysis (or, temporal trajectory analysis) is an excellent way of analyzing the satellite datasets for Earth monitoring and has been receiving increasing attention in the last decade, specifically, after the Landsat data became freely accessible in 2008 [17]. In time series analysis, several images of the scene under consideration, taken over a period of time, are stacked together chronologically, and are subsequently analyzed. Typically, the data set is converted into a collection of time series, each time series corresponding to a particular spectral band for one pixel. The objective is to discover a 'trend' in how different relevant variables (indicators) evolve over time. The analysis made is based on the behaviors of the time series of these variables. When the trajectory of one or more of the variables departs from the normal (or, predicted), a change is detected. Several time series analysis algorithms have been proposed by different groups in the remote sensing community.

While several time series analysis algorithms have been proposed, the size of the datasets itself is prohibitive, currently of the order of petabytes and growing. An EOS image stack typically consists of multiple images of a fixed area on the Earth's surface (same latitudes and longitudes) taken at different time points. Experiments on multicore servers indicate that carrying out meaningful time series analysis on a single interannual, multitemporal stack with existing state of the art codes can take several days. An HPC platform for time series analysis of satellite images obtained from MODIS was presented in [3]. In contrast with Landsat images, MODIS images have much coarser spatial resolution but much better temporal resolution. Knowing the need for scalable computations in remote sensing, several architectures have also been proposed. These mainly include massive parallel clusters, heterogeneous clusters, grid computing, GPUs and the like [9], [12], [14]. Considerable HPC work on classification (e.g., [13]) has also been carried out.

This paper presents parallel results for the constituents of a polyalgorithm under development for trend and change detection in Landsat images. The polyalgorithm consists of two algorithms, fundamentally distinct from each other, both by construction and in the phenomenon they capture. The algorithms are implemented in the scientific programming language Fortran 2003. Parallelization across pixels is implemented and further possibilities for speeding up the individual algorithms as well as the combined algorithm are discussed. Experimental results of applying the codes to an image with approximately 10^8 pixels are presented.

The two algorithms are presented in Section 2. Results for the algorithms individually are presented and discussed in Section 3. Section 4 describes parallel implementations, with conclusions and future work on the polyalgorithm in Section 5.

2. ALGORITHMS AND IMPLEMENTATION

Notation and definitions: For an $m \times n$ matrix A, an n-vector x, $I \subset \{1, \ldots, m\}$, $J \subset \{1, \ldots, n\}$, let A_{IJ} denote the submatrix of A formed from the rows indexed by I and the columns indexed by J, and x_J denote the subvector of x indexed by J. $A_{I\cdot}$ $(A_{\cdot J})$ are the rows (columns) of A indexed by I (J), respectively. An image is an $R \times C$ matrix D, where each D_{rc} (pixel) is an $S \times B$ matrix, whose (s, b) element $(D_{rc})_{sb}$ is the signal value at time index s and frequency band index b.

2.1. Exponentially Weighted Moving Average Change Detection (EWMACD) [4], [5]

```
Algorithm EWMACD.
for band $b := 1$ step 1 until $B$ do
  for row $r := 1$ step 1 until $R$ do
    for column $c := 1$ step 1 until $C$ do
      begin
```

Step 1: Write the time series data in the column $(D_{rc})_{\cdot b}$ as $(D_{rc})_{\cdot b} = \begin{pmatrix} u \\ v \end{pmatrix}$, where the M-dimensional vector u is deemed training data and the $(S - M)$-dimensional vector v as the test data. Let

$$X = \begin{bmatrix} 1 & \sin t_1 & \cos t_1 & \cdots & \sin Kt_1 & \cos Kt_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \sin t_M & \cos t_M & \cdots & \sin Kt_M & \cos Kt_M \end{bmatrix}$$

be the Gram matrix for the time points $t_1, \ldots, t_M$, using K harmonics, where $M > 2K+1$. The least squares fit to the training data u is then written as $u(t) = \alpha_0 + \sum_{i=1}^{K} (\alpha_{2i-1} \sin it + \alpha_{2i} \cos it)$ with coefficients $\alpha = (X^t X)^{-1} X^t u$ and residual $E(\alpha) = u - X\alpha$.

REMARK 1. In practice α is computed via a QR factorization of X, not by computing $(X^tX)^{-1}$ explicitly.

Next let $I = \{i \mid |E(\alpha)_i| < \gamma_1\}$, where γ_1 is a user defined threshold and $|I| > 2K+1$. Calculate the coefficients for an improved fit to the underlying signal as $\alpha^* = \big((X_{I\cdot})^t X_{I\cdot}\big)^{-1}(X_{I\cdot})^t\, u_I$. With the refined coefficients α^*, calculate the residuals for

(i) the complete time series $(D_{rc})_{\cdot b}$ as $E^*(\alpha^*) = (D_{rc})_{\cdot b} - \bar{X}\alpha^*$, where $\bar{X}_{s\cdot} = (1, \sin t_s, \cos t_s, \ldots, \sin Kt_s, \cos Kt_s)$, for $s = 1, \ldots, S$.

(ii) the outlier-free time series as $\big(E^*(\alpha^*)\big)_{\bar{I}}$, where $\bar{I} = \{s \mid |E^*(\alpha^*)_s| < \gamma_2\}$, γ_2 is a user defined threshold, and

(iii) the outlier-free training set $\hat{I} = \bar{I} \cap \{1, \ldots, M\}$ as $\big(E^*(\alpha^*)\big)_{\hat{I}} = u_{\hat{I}} - X_{\hat{I}\cdot}\alpha^*$, where $|\hat{I}| > 2K+1$.

REMARK 2. In the present implementation,

$$\gamma_2 = \begin{cases} 1.5\eta, & i \in [1, M], \\ 20\eta, & i \in (M, S], \end{cases}$$

where η is the standard deviation of the first M elements of the residual vector $E^*(\alpha^*)$.

Step 2: Define the control limit vector τ by $\tau_i = \mu + \sigma L\sqrt{\frac{\lambda}{2-\lambda}\big(1-(1-\lambda)^{2i}\big)}$, for $i = 1, 2, \ldots, |\bar{I}|$, where $\mu = 0$ is used here, σ is the standard deviation of the outlier-free training data errors $\big(E^*(\alpha^*)\big)_{\hat{I}}$, L is the multiple of this standard deviation σ, and $\lambda \in (0, 1]$ is the weight given to the most recent residual in the exponentially weighted moving average (EWMA) defined next. L is typically set to 3 or slightly smaller depending on the value of λ.

Step 3: Let $\bar{I} = \{j_1, j_2, \ldots, j_{|\bar{I}|}\}$, $j_1 < j_2 < \cdots < j_{|\bar{I}|}$. Define the vector z by

$$z_1 = \big(E^*(\alpha^*)\big)_{j_1}, \quad z_i = (1-\lambda)z_{i-1} + \lambda\big(E^*(\alpha^*)\big)_{j_i}, \quad i = 2, \ldots, |\bar{I}|.$$

This is the exponentially weighted moving average (EWMA) of the residual $\big(E^*(\alpha^*)\big)_{\bar{I}}$.

Step 4: Define the flag history S-vector f by

$$f_s = \begin{cases} \operatorname{sgn}(z_i)\big\lfloor |z_i/\tau_i| \big\rfloor, & s = j_i \in \bar{I}, \\ 0, & \text{otherwise}. \end{cases}$$

If there is a run of $+1$ or -1 in the values $\operatorname{sgn}(\Delta f_s) = \operatorname{sgn}(f_{s+1} - f_s)$ of length ϖ, called the 'persistence', signal a change at the index s beginning the (nonzero) run.

REMARK 3. Missing data is automatically handled by not assuming that the time points t_i are equally spaced. Alternatively, missing data for time point t_k can be handled by including t_k in the sequence $(t_1, t_2, \ldots, t_S)$, but excluding t_k from the training sequence $(t_1, t_2, \ldots, t_M)$ and k from the sets I, $\bar{I}$, and $\hat{I}$, which is equivalent to treating $(D_{rc})_{kb}$ as an outlier and to setting the flag $f_k = 0$.

end

2.2. Breaks For Additive and Seasonal Trend (BFAST) [16]

Algorithm BFAST.

Let $T = (t_1, \ldots, t_S)$ be the sequence of given time points and the S-vector u denote the time series data in the column $(D_{rc})_{\cdot b}$, i.e., $u = (D_{rc})_{\cdot b}$. Assume that the general model is of the form $u = \mathcal{V} + \mathcal{W} + \epsilon$, where $\mathcal{V}$ and $\mathcal{W}$ denote the iteratively computed trend and seasonal components, respectively, present in the data and ϵ is the noise. The trend $\mathcal{V}$ may be piecewise linear and the seasonal component $\mathcal{W}$ may be piecewise harmonic. Let N be the maximum number of iterations, n be the iteration number, and $\mathcal{V}^n$ and $\mathcal{W}^n$ be the trend and seasonal components, respectively, computed at the nth iteration. Let $h \in (0, 1)$ denote the proportion of data points by which two consecutive breakpoints t_i and t_j (including t_1 and t_S) must be separated. Thus $\lceil Sh \rceil \leq j - i - 1$. Take the length of moving windows to be $\lceil Sh \rceil$, initialize the iteration number $n := 1$, and initialize the seasonal component as $\mathcal{W}^0(T) = (w_1^0, \ldots, w_S^0)$.

```
for band $b := 1$ step $1$ until $B$ do
  for row $r := 1$ step $1$ until $R$ do
    for column $c := 1$ step $1$ until $C$ do
      begin
```

Step 1.1: Determine the possibility of breakpoints in trend.

Eliminate the seasonal component from the data $u^n = u - \mathcal{W}^{n-1}(T)$.

The ordinary least squares (OLS) estimator for the trend is given as $\alpha = (X^t X)^{-1} X^t u^n$ where X is the Gram matrix for linear regression given by

$$X = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_S \end{pmatrix}.$$

The prediction error (or residual vector or the OLS residual) is defined as $E^o = u^n - X\alpha$, where the superscript 'o' is used to signify the fact that these residuals are OLS regression based. Consider the process defined by the moving sums (MOSUM) of these OLS residuals

$$Q^o = \left\{ \frac{1}{\sigma\sqrt{\lceil Sh \rceil}} \sum_{i=k-\lceil Sh \rceil+1}^{k} E_i^o \right\}_{k=\lceil Sh \rceil}^{S},$$

where σ is the sample standard deviation of all the OLS residuals.

Compute the OLS-MOSUM test statistic $\hat{f}^o = \max\limits_{1 \leq k \leq S - \lceil Sh \rceil + 1} |Q_k^o|$ as the maximum absolute value of this process, then compute the asymptotic critical value of the OLS-MOSUM test using the two-sided boundary-crossing probability $p_T = P[f^o > \hat{f}^o]$, where p_T is read from the Brownian Bridge table.

A p-value less than a user defined parameter $\tau_{\mathcal{V}} \in (0, 1)$ indicates the presence of breakpoints.

Remark 1: As discussed in [6], under the null hypothesis, the OLS-MOSUM process converges in distribution to the increments of a Brownian Bridge process.

Step 1.2: Locate trend breakpoints.

Suppose $p_T \leq \tau_{\mathcal{V}}$. To locate the breakpoints, consider all possible partitions of the domain, compute OLS fits for each partition, and settle with a partition that yields minimum squared error.

Let $X_{[i,j]}$ denote the matrix formed from rows i through j of the matrix X, and $\alpha_{[i,j]}$ denote the least squares coefficients computed using the matrix $X_{[i,j]}$ with time points $t_i, \ldots, t_j$, and data $u_{[i,j]}^n = u_{\{i,\ldots,j\}}^n$. For

$i = 1, \ldots, S - \lceil Sh \rceil + 1$, consider each window $[t_i, \ldots, t_{j-1}]$, $i + 2 \leq j \leq S$, and the linear fit in this window. The recursive residual at point t_j is then defined as the weighted prediction error

$$E^r_{ij} = \frac{u^n_{[j,j]} - X_{[j,j]}\alpha_{[i,j-1]}}{\sqrt{1 + X_{[j,j]}\left(X^t_{[i,j-1]}X_{[i,j-1]}\right)^{-1}X^t_{[j,j]}}}.$$

The superscript 'r' is used to signify the fact that the process/statistic is recursive residual based.

Suppose a breakpoint has been found at t_i. Then the cost of placing the next breakpoint at t_k is calculated as the accumulated sum of squared recursive residuals in the interval $[t_i, t_{k-1}]$, i.e., $\rho_{ik} = \sum_{j=i+2}^{k-1}\left(E^r_{ij}\right)^2$. All possible positions for the breakpoints can thus be calculated by considering the moving sums of squared recursive residuals, i.e., the process defined by

$$Q^r = \left\{\left\{\sum_{j=i+2}^{k}\left(E^r_{ij}\right)^2\right\}_{k=i+2}^{S}\right\}_{i=1}^{S-\lceil Sh \rceil+1}.$$

Given the number μ of desired interior breakpoints, let $k_1, \ldots, k_\mu$ be integers such that $k_{i+1} - k_i > \lceil Sh \rceil$, $k_1 > \lceil Sh \rceil + 1$, and $k_\mu < S - \lceil Sh \rceil$. Determine $K = (1, k_1, \ldots, k_\mu, S)$ to minimize the moving sums of squared recursive residuals

$$\sum_{i=3}^{k_1-1}\left(E^r_{1,i}\right)^2 + \sum_{i=k_1+2}^{k_2-1}\left(E^r_{k_1,i}\right)^2 + \sum_{i=k_2+2}^{k_3-1}\left(E^r_{k_2,i}\right)^2 + \quad \cdots + \sum_{i=k_\mu+2}^{S}\left(E^r_{k_\mu,i}\right)^2.$$

Then $(t_{k_1}, \ldots, t_{k_\mu})$ are the interior breakpoints in the trend component.

REMARK 2: The breakpoints t_1, t_{k_1}, $\ldots$, t_{k_μ}, t_S are optimal in the sense of the above moving sums of squared recursive residuals criterion.

REMARK 3: If $p_T > \tau_{\mathcal{V}}$, then there are only two breakpoints (t_1 and t_S) and no interior breakpoints. So this step is skipped and there is simply one linear fit over the entire domain $[t_1, t_S]$ (Step 1.3).

Step 1.3: Let $k_0 = 1$, $k_{\mu+1} = S$, and $I_0 = [t_{k_0}, t_{k_1})$, $I_1 = [t_{k_1}, t_{k_2})$, $\ldots$, $I_\mu = [t_{k_\mu}, t_{k_{\mu+1}}]$. For each interval I_i, determine the linear regression coefficients

$$\gamma^i = \left(X^t_{[k_i,k_{i+1}]}X_{[k_i,k_{i+1}]}\right)^{-1}X_{[k_i,k_{i+1}]}u^n_{[k_i,k_{i+1}]}$$

and construct the (discontinuous) piecewise linear fit $\mathcal{V}^n(t) = \sum_{i=0}^{\mu}\Gamma^i(t)$, where

$$\Gamma^i(t) = \begin{cases}\gamma^i_0 + \gamma^i_1 t, & t \in I_i,\\ 0, & \text{otherwise.}\end{cases}$$

Let $\mathcal{V}^n(T) = (v^n_1, \ldots, v^n_S)$ be the sequence of values estimated at $t_1, \ldots, t_S$ using this piecewise linear fit.

Step 2.1: Determine the possibility of breakpoints in seasons.

Eliminate the estimated trend component from the observed data $\tilde{u}^n = u - \mathcal{V}^n(T)$. The Gram matrix for the seasonal (harmonic) component is given by

$$Y = \begin{pmatrix} 1 & \sin t_1 & \cos t_1 & \cdots & \sin Kt_1 & \cos Kt_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \sin t_S & \cos t_S & \cdots & \sin Kt_S & \cos Kt_S \end{pmatrix},$$

where K is the degree of the trigonometric polynomial used for regression. The trigonometric regression coefficients for the seasonal component are computed as $\beta = (Y^tY)^{-1}Y^t\tilde{u}^n$. The prediction error for this fit is defined as $E^o = \tilde{u}^n - Y\beta$. The OLS-MOSUM process for these errors is given by

$$Q^o = \left\{ \frac{1}{\sigma\sqrt{\lceil Sh \rceil}} \sum_{i=k-\lceil Sh \rceil+1}^{k} E_i^o \right\}_{k=\lceil Sh \rceil}^{S},$$

and the OLS-MOSUM test statistic is $\hat{g}^o = \max\limits_{1 \le j \le S-\lceil Sh \rceil+1} |Q_j^o|$. The two-sided boundary-crossing probability $p_S = P[g^o > \hat{g}^o]$ is read from the Brownian Bridge table.

A p-value less than a user defined parameter $\tau_{\mathcal{W}} \in (0,1)$ indicates the presence of seasonal breakpoints.

Step 2.2: Locate seasonal breakpoints.

Suppose $p_S \le \tau_{\mathcal{W}}$. Using the same notation as for the trend breakpoints,

$$E_{ij}^r = \frac{\tilde{u}^n_{[j,j]} - Y_{[j,j]}\beta_{[i,j-1]}}{\sqrt{1 + Y_{[j,j]}\left(Y^t_{[i,j-1]}Y_{[i,j-1]}\right)^{-1}Y^t_{[j,j]}}}$$

is the recursive residual at time t_j, obtained by trigonometric regression in the time window $[t_i, t_{j-1}]$.

Given the number ν of desired seasonal interior breakpoints and a minimum number of data points separating breakpoints (as for the trend), let l_1, …, l_ν be integers such that $l_{i+1} - l_i > \lceil Sh \rceil$, $l_1 > \lceil Sh \rceil + 1$, and $l_\nu < S - \lceil Sh \rceil$. Determine $L = (1,\, l_1,\, \ldots,\, l_\nu,\, S)$ to minimize the moving sums of squared recursive residuals

$$\sum_{i=3}^{l_1-1} \left(E^r_{1,i}\right)^2 + \sum_{i=l_1+2}^{l_2-1} \left(E^r_{l_1,i}\right)^2 + \sum_{i=l_2+2}^{l_3-1} \left(E^r_{l_2,i}\right)^2 + \qquad \cdots + \sum_{i=l_\nu+2}^{S} \left(E^r_{l_\nu,i}\right)^2.$$

Then $(t_{l_1}, \ldots, t_{l_\nu})$ are the interior breakpoints in the seasonal component.

Remark 4: If $p_S > \tau_{\mathcal{W}}$, then there are only two breakpoints (t_1 and t_S) and no interior breakpoints. So this step is skipped and there is simply one trigonometric polynomial fit over the entire domain $[t_1, t_S]$ (Step 2.3).

Step 2.3: Let $l_0 = 1$, $l_{\nu+1} = S$, and $J_0 = [t_{l_0}, t_{l_1})$, $J_1 = [t_{l_1}, t_{l_2})$, …, $J_\nu = [t_{l_\nu}, t_{l_{\nu+1}}]$. For each interval J_j determine the trigonometric polynomial regression coefficients

$$\delta^j = \left(Y^t_{[l_j,l_{j+1}]}Y_{[l_j,l_{j+1}]}\right)^{-1}Y_{[l_j,l_{j+1}]}\tilde{u}^n_{[l_j,l_{j+1}]}$$

and construct the (discontinuous) piecewise trigonometric polynomial $\mathcal{W}^n(t) = \sum_{j=0}^{\nu} \Delta^j(t)$, where

$$\Delta^j(t) = \begin{cases} \delta_1^j + \sum\limits_{k=1}^{K} \delta_{2k}^j \sin kt + \delta_{2k+1}^j \cos kt, & t \in J_j, \\ 0, & \text{otherwise.} \end{cases}$$

Let $\mathcal{W}^n(T) = (w_1^n,\, \ldots,\, w_S^n)$ be the sequence of values estimated at t_1, …, t_S using this piecewise trigonometric polynomial approximation.

Step 3: Compare the breakpoints between iterations $n-1$ and n.

If the Hamming distance between the two breakpoint vectors $(t_{k_1}, \ldots, t_{k_\mu}, t_{l_1}, \ldots, t_{l_\nu})$ at iterations $n-1$ and n is less than some defined tolerance or the number of iterations has reached N, then exit. Otherwise, increment the iteration number n and repeat Steps 1.1 to 3.

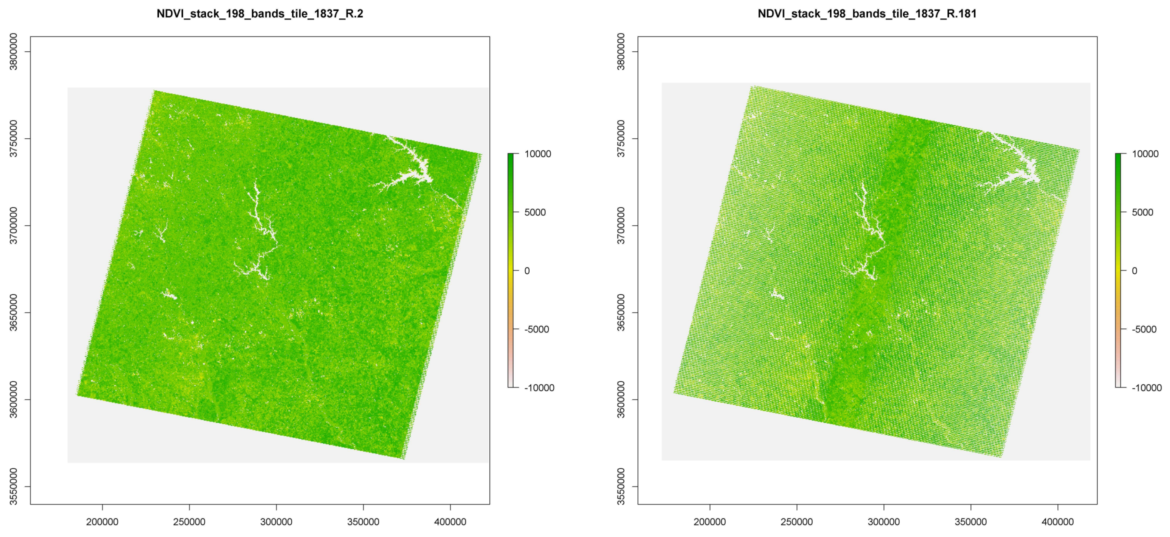


Figure 1. Processed NDVI values from Landsat images for 2009 (left) and 2014 (right). The x- and y-axes represent relative pixel coordinates of the extracted image.

end

3. RESULTS AND DISCUSSION

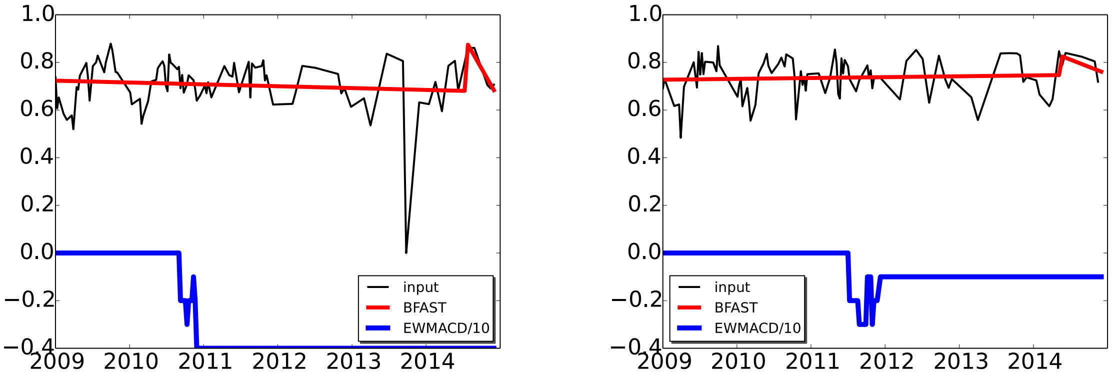
The algorithms EWMACD and BFAST were tried on study areas located in Oregon and South Carolina, with results presented for the latter. Figure 1 shows the satellite images taken from Landsat path 18, row 37, on dates January 3rd, 2009, and February 16th 2014, corresponding to the beginning and end of the image stack (henceforth referred to as SC1837) under consideration, which has 198 time points and one band, the normalized difference vegetation index [10]

$$\text{NDVI} = \frac{\text{NIR} - \text{R}}{\text{NIR} + \text{R}},$$

where NIR is the near infrared (band 4, biomass) and R is the visual red (band 3, vegetation slopes). NDVI is known to be a good metric for vegetation cover, where negative values of NDVI are deemed irrelevant as they correspond to water, clouds, or missing observations. For processing, positive values of NDVI are scaled by 10,000 and negative values are masked out (set to $-9,999$). The image stack dimensions are $R = 7411$, $C = 8801$, $S = 198$, $B = 1$. Like most time series analysis algorithms, both algorithms rely on user defined parameters, requiring, in general, *a priori* knowledge of the scene. The experiments here adhered to the published values of the parameters for each of these algorithms, which are listed in Table 1. For EWMACD, all the time points in the years 2009 to 2011 were used as training data, so the length M of the training period varied from pixel to pixel.

Table 1. Algorithm parameters

EWMACD	BFAST
$K = 2$	$K = 1$
$L = 0.5$	$\mu = \nu = 2$
$\lambda = 0.3$	$h = 0.05$
$\varpi = 7$	$\tau_{\mathcal{V}} = \tau_{\mathcal{S}} = 1.0$



TCC(2009)=89.67, TCC(2013)=67.2

TCC(2009)=78.9, TCC(2014)=97.2

Figure 2. BFAST is the piecewise linear/piecewise harmonic model showing the breakpoints. The EWMACD flag history (divided by 10) signals times of change by runs of increases or decreases in the flags.

Validation is based on tree canopy cover (TCC) data [15]. For a $30m \times 30m$ area (one pixel) the TCC is defined as the proportion of the area that is covered by tree canopy versus "not tree canopy". Methods to measure the TCC for a pixel are known. For the study area SC1837, TCC data exists for 457 pixels. For each of these pixels, at the NDVI data band, the two algorithms were run on the image stack. Results for two such pixels are displayed in Figure 2. TCC for the pixel in Figure 2(left) reduced from 89.67% in 2009 to 67.2% by the year 2013, an approximately 22% loss. BFAST captures the loss correctly indicating a decline in vegetation cover, with a quick, short recovery towards the end of the year 2014 (the available TCC data does not cover late 2014). EWMACD clearly captures the loss correctly.

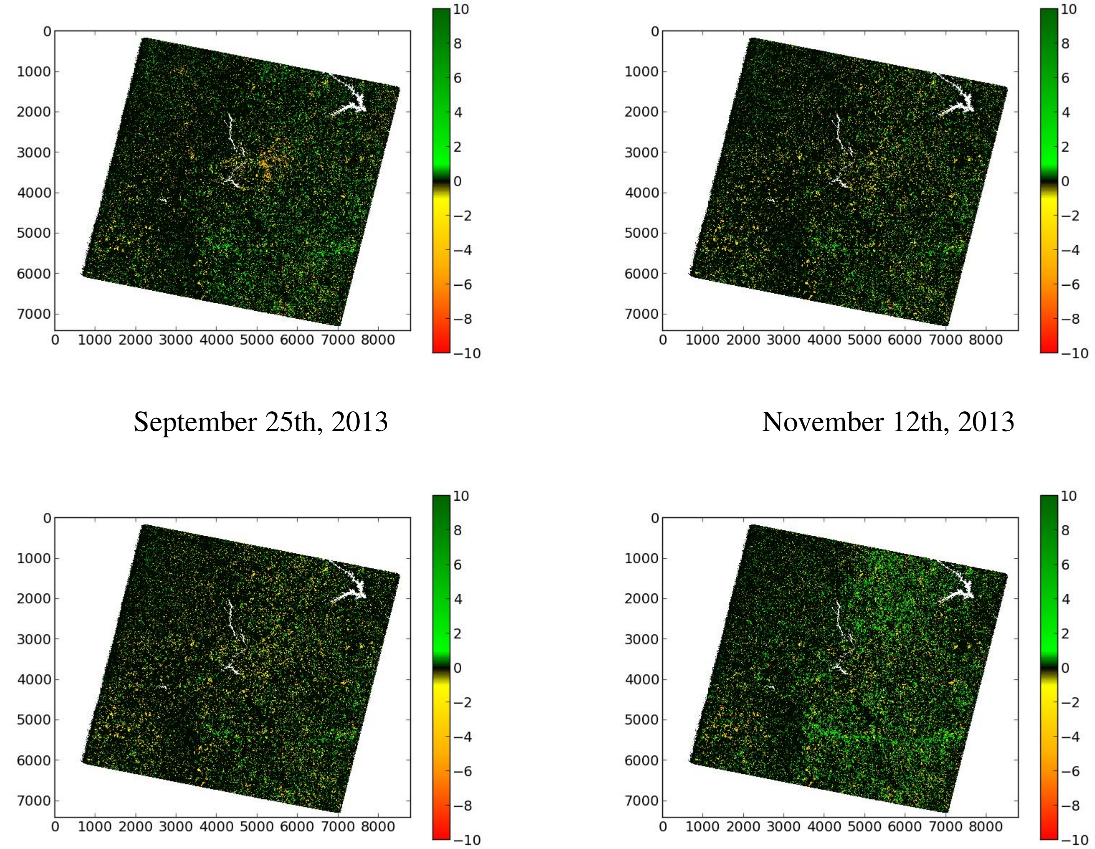
The pixel displayed in Figure 2(right), on the other hand, gained approximately 18% in TCC. The NDVI values also, in general, show an increase in mean. BFAST correctly captures the trend and indicates gradual recovery throughout, with some disturbance towards the end of 2014. EWMACD, however, indicates a disturbance just after it's training period (beginning of 2012) — a sharp loss followed by some recovery and stability thereafter. This behavior can be attributed to ill-chosen parameter values for this pixel.

The logical mathematical description of an image stack uses the index order (r, c, s, b), but because of Fortran array element storage order and the hardware effects of cache misses and paging, the image stack is actually stored and processed in the index order (s, c, r, b). Failure to use this latter index order can result in a cache miss rate as high as 28%. The outcomes of both the sequential and the parallel (Fortran 2003) codes match with those of the original codes (written in R) completely for EWMACD and closely for BFAST. The slight deviation in results for BFAST can be attributed to its recursive use of models, which makes the algorithm sensitive to round-off error. Figure 3 displays the EWMACD results on a few (4) time points. At any given time point, a black pixel indicates being flagged by EWMACD as having no disturbance, a green pixel as in recovery, and a red pixel as in loss. So an area to the northeast of pixel $(4500, 4000)$ was in vegetation loss in September 2013 while by February 2014, the entire area to the right of $(4000, :)$ had substantially recovered.

Both algorithms involve a fair number of intermediate array variables, so global arrays were used for all work arrays, and Fortran vector instructions were utilized wherever possible.

4. PARALLEL IMPLEMENTATION

One image from a given Landsat path/row typically consists of more than 10^7 pixels. The sequential implementation of BFAST in Fortran takes over 7 hours to analyze the time series of a $1000 \times 1000 = 10^6$



September 25th, 2013

November 12th, 2013

December 30th, 2013

February 16th, 2014

Figure 3. EWMACD flags for four late time points of image stack when the data in 2009–2011 was used as training data.

pixel image. EWMACD is faster but still takes 1 hour and 20 minutes for 10^6 pixels. Knowing that (i) the image stack discussed in this paper consists of only 198 time points (2009 to 2014) while there are currently 900 time points (1984 to 2014) actually available, (ii) the run times discussed here are for a single path/row only while there are 450 path/rows in the US alone, and other similar facts, scaling the codes is imperative for any meaningful analysis. The sequential codes described in the previous sections were parallelized using OpenMP. The hardware bottlenecks and computational hot spots are systematically identified and addressed.

The sequential code already harnesses vector instructions wherever possible. The input and output arrays are the only large arrays; the indexing mentioned in Section 3 ensures good memory locality. The remaining significant hardware bottleneck is load imbalance. Specifically, since the time series processing for any given pixel is independent of that for any other pixel, the algorithms are apparently embarrassingly parallel with respect to pixels. Landsat images, however, suffer from missing observations (due to factors beyond human control), thereby resulting in ‘invalid’ pixels (a pixel is declared invalid if there are fewer than $2K+1$ observations in the entire time span, cf. Section 2.1). These invalid pixels are randomly distributed across the data. This induces a very high work load imbalance across the pixels. Attempting to weed out invalid pixels in a preprocessing step and execute the PARALLEL DO loop for only valid pixels leads to CPU underutilization (from 99.9% to 70–80%), simultaneously increasing the OpenMP time. This presumably is

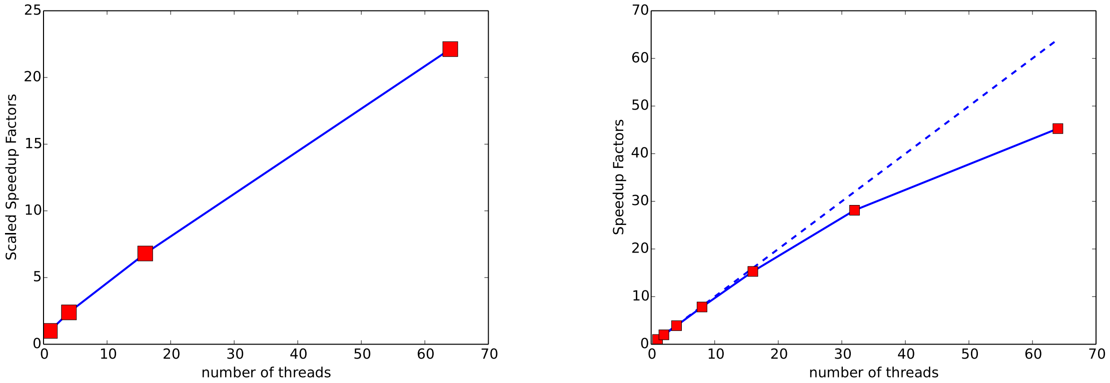


Figure 4. BFAST scaled speedup for a base size $B = 50 \times 50$ (left); speedup in parallel sections of EWMACD code (right).

due to memory contention: the memory access pattern for the latter approach is such that multiple threads try to access the same memory bank(s). Furthermore, even amongst the valid pixels, the total number of observations (S) available for one pixel can be much less than the number of observations available for some other pixel. So, even with this preprocessing approach the work load balance is not guaranteed.

Finally, allocating/deallocating arrays within each thread is inefficient. Allocatable global arrays in modules can be used by threads via THREADPRIVATE, but this data copy mechanism does not work with dynamic loop scheduling [11], which is desirable because of the large variance in pixel analysis times (including missing data for a pixel). The best alternative is using Fortran automatic arrays with OpenMP PRIVATE.

Next, the computational hot spots are identified. For EWMACD, more than 50% of the time is spent in least squares fitting (cf. Section 2, Step 1), specifically in DGELS (LAPACK) calls. LAPACK [2] is already optimized for the hardware. 22% of the total time is in the calculation of residuals (again, cf. Section 2.1, Step 1). This subroutine has two DO loops with dependencies and cannot be vectorized.

For BFAST, approximately 97% of the OpenMP time is spent in computing the recursive residuals (cf. Section 2.2, steps 1.2 and 2.2). Essentially, linear and harmonic least squares fits are done in every permissible interval, and the least squares fitting is already optimized.

In summary, after considering and testing several alternatives, the best approach found was to (1) perform the raw binary stream input data order (r, c, s, b) conversion to (s, c, r, b) order in parallel; (2) cull invalid (including missing) pixels inside the subroutines EWMACD and BFAST, which are called from within a PARALLEL DO (a pixel is declared invalid if there are fewer than $2K + 1$ observations in the entire time span, cf. Section 2.1); (3) convert the nested DO loop `DO r=1,R; DO c=1,C` into a single PARALLEL DO loop `DO k=1,R*C,A`; (4) use OpenMP `SCHEDULE(DYNAMIC,1)`; (5) process a chunk of A pixels indexed by k on each call to EWMACD and BFAST; (6) use automatic rather than allocatable arrays for all small work arrays in the subroutines, and allocate/deallocate just one large work array in both EWMACD and BFAST; (7) perform all I/O outside parallel OpenMP constructs to reduce memory and disk contention. Note that manually collapsing the nested loops and chunking within the pixel processing subroutine is more efficient than collapsing and chunking at the OpenMP directive level, since the latter would call the subroutines, which allocate and deallocate numerous work arrays, for each pixel index. The difficulty of load balancing “embarrassingly parallel” applications is analyzed theoretically in [1].

Figure 4(left) shows the scaled speedup for BFAST, i.e., increasing both the problem size and the number of cores. The isoefficiency (constant efficiency as both the problem size and number of cores are scaled up) decreases significantly, indicating some combination of poor load balancing (the pixel chunk size $A = 100$), main memory contention, and increasing thread management overhead, as yet unresolved.

Parallel results for the full scene, which consists of $7411 \times 8801 = 65224211$ pixels at one band, processed only with EWMACD, are shown in Figure 4(right). Using 64 cores, the full scene is processed in 1.86 minutes, with a speedup of roughly 46. The input binary image is 25GB. For this image, the code needs 74GB of memory. When a single thread (core) is used, the cache miss rate is 0.566% and 1.03 instructions per cycle are executed. For 64 cores, the cache miss rate is 0.721% and 0.52 instructions per cycle are executed. On 64 cores, the FLOPS performance of the EWMACD code is 48.54 GFLOPs. The peak theoretical performance for this machine is 358.4 GFLOPs, yielding performance to peak ratio of $48.54/358.4 = 13.54\%$.

For the algorithms to be used together in a polyalgorithm, speeding up BFAST for a single pixel needs to be aggressively explored, lest BFAST be used only on a need basis (> 30 hours for this full image).

For all the Fortran codes, the input (as well as output) image stacks were in binary file format. Fortran I/O with streaming access was utilized to read and write these files. The results presented in this paper were obtained on a single machine: 64 core AMD Opteron 6276, 1.4GHz CPU, 2MB cache per core, 265 GB main memory, CentOS, gfortran compiler version 4.8.

5. CONCLUSIONS AND FUTURE WORK

Given the inconsistent trend predictions between the different algorithms, the sometimes erratic behavior of a given algorithm on a given image stack, the sensitivity to parameters for some algorithms, and the prohibitive execution times for serial codes, there is clearly a need for a parallel polyalgorithm (an intelligent, adaptive union of multiple algorithms). The work begun here, assessing the scalability and memory footprint, the parameter sensitivity, and the range of applicability of individual algorithms is but the first step toward such a parallel polyalgorithm for hypertemporal Landsat image stacks.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their useful comments, Jill Derwin for remote sensing technical assistance and Dr. E. B. Brooks for useful discussions. This work was supported in part by NSF Grant CNS-1565314 and USDA Grant 422350.

REFERENCES

Ahn, T.-H., A. Sandu, L. T. Watson, C. A. Shaffer, Y. Cao, and W. T. Baumann. 2015. “A framework to analyze the performance of load balancing schemes for ensembles of stochastic simulations”. *International Journal Parallel Programming* vol. 43 (4), pp. 597–630.

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1992. *LAPACK Users’ Guide*. SIAM.

Bergh, F. V. D., K. J. Wessels, S. Miteff, T. L. V. Zyl, A. D. Gazendam, and A. K. Bachoo. 2012. “HiTempo: a platform for time-series analysis of remote-sensing satellite data in a high-performance computing environment”. *International Journal of Remote Sensing* vol. 33 (15), pp. 4720–4740.

Brooks, E. B., V. A. Thomas, and R. H. Wynne. 2012. “Fitting the multitemporal curve: a Fourier series approach to the missing data problem in remote sensing analysis”. *IEEE Transactions on Geosciences and Remote Sensing* vol. 50 (9), 3340–3353.

Brooks, E. B., R. H. Wynne, V. A. Thomas, C. E. Blinn, and J. W. Coulston. 2014. “On-the-fly massively multitemporal change detection using statistical quality control charts and Landsat data”. *IEEE Transactions on Geoscience and Remote Sensing* vol. 52 (6), pp. 3316-3332.

Chu, C.-S. J., K. Hornik, and C.-M. Kuan. 1995. “MOSUM tests for parameter constancy”. *Biometrika* vol. 82 (3), pp. 603–617.

Fall, S., D. Niyogi, A. Gluhovsky, R.A. Pielke, Sr., E. Kalnay, and G. Rochon. 2010. “Impacts of land use and land cover on temperature trends over the continental United States: assessment using the North American Regional Reanalysis”. *International Journal of Climatology* vol. 30 (13), pp. 1980-1993.

Hansen, M. C., P. V. Potapov, R. Moore, M. Hancher, S. A. Turubanova, A. Tyukavina, D. Thau, S. V. Stehman, S. J. Goetz, T. R. Loveland, A. Kommareddy, A. Egorov, L. Chini, C. O. Justice, and J. R. G. Townshend. 2013. “High-resolution global maps of 21st-century forest cover change”. *Science* vol. 342 (6160), pp. 850–853.

Kalluri, S. N. V., J. JáJá, D. A. Bader, Z. Zhang, J. R. G. Townshend, and H. Fallah-Adl. 2000. “High performance computing algorithms for land cover dynamics using remote sensing data”. *International Journal of Remote Sensing* vol. 21 (6 & 7), pp. 1513–1536.

Kriegler, F. J., W. A. Malila, R. F. Nalepka, and W. Richardson. 1969. “Preprocessing transformations and their effects on multispectral recognition”. In *Proceedings of the Sixth International Symposium on Remote Sensing of Environment*. pp. 97–131.

Lee, C. A., S. D. Gasster, A. J. Plaza, C.-T. Chang, and B. Huang. 2011. “Recent developments in high performance computing for remote sensing: a review”. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* vol. 4 (3), pp. 508–527.

LLNL. . *OpenMP*. https://computing.llnl.gov/tutorials/openMP/#THREADPRIVATE.

Phillips, R. D., L. T. Watson, and R. H. Wynne. 2007. “Hybrid image classification and parameter selection using a shared memory parallel algorithm”. *Computers & Geosciences* vol. 33 (7), pp. 875–897.

Plaza, A. J., and C.-I. Chang. 2007. *High Performance Computing in Remote Sensing*. CRC Press.

Toney, C., G. Liknes, A. Lister, and D. Meneguzzo. 2012. “Assessing alternative measures of tree canopy cover: photo-interpreted NAIP and ground-based estimates”. In *Proceedings of Monitoring Across Borders: 2010 Joint Meeting of the Forest Inventory and Analysis (FIA) Symposium and the Southern Mensurationists. Edited by W. McWilliams and F.A. Roesch. USDA Forest Service, Southern Research Station, Asheville, North Carolina, e-Gen. Tech. Rep. SRS-157*. pp. 209–215.

Verbesselt, J., R. Hyndman, G. Newnham, and D. Culvenor. 2010. “Detecting trend and seasonal changes in satellite image time series”. *Remote Sensing of Environment* vol. 114 (1), pp. 106–115.

Woodcock, C. E., R. Allen, M. Anderson, A. Belward, R. Bindschadler, W. Cohen, F. Gao, S. N. Goward, D. Helder, E. Helmer, R. Nemani, L. Oreopoulos, J. Schott, P. S. Thenkabail, E. F. Vermonte, J. Vogelmann, M. A. Wulder, and R. H. Wynne. 2008. “Free access to Landsat imagery”. *Science* vol. 320 (5879), pp. 1011–1011.

AUTHOR BIOGRAPHIES

R. SAXENA (Ph.D., Arizona State, 2008) has interests in image processing, numerical analysis, high performance computing, and big data analytics.

LAYNE. T. WATSON (Ph.D., Michigan, 1974) has interests in numerical analysis, mathematical programming, bioinformatics, and data science. He has been involved with the organization of HPCS since 2000.

VALERIE A. THOMAS (Ph.D., Queen’s, Ontario, 2006) has interest in the use of remote sensing to examine forest canopy physiology, structure, and function across space and time.

RANDOLPH H. WYNNE (Ph.D., Wisconsin, 1996) has interests in the applications of remote sensing to forest monitoring, modeling, and management. He has long worked at the interface of computing with basic and applied remote sensing science.

GLOBAL DETERMINISTIC AND STOCHASTIC OPTIMIZATION IN A SERVICE ORIENTED ARCHITECTURE

Chaitra Raghunath
Tyler H. Chang
Dept. of Computer Science
Virginia Polytechnic Institute
& State University
Blacksburg, VA 24061
thchang@vt.edu

Layne T. Watson
Dept. of Computer Science
Dept. of Mathematics
Dept. of Aerospace & Ocean Eng.
Virginia Polytechnic Institute
& State University

Mohamed Jrad
Rakesh K. Kapania
Dept. of Aerospace & Ocean Eng.
Virginia Polytechnic Institute
& State University

Raymond M. Kolonay
AFRL/RQVC
2210 8th Street, Bldg. 146
Wright-Patterson Air Force Base
Dayton, OH 45433

ABSTRACT

Service ORiented Computing EnviRonment (SORCER) is a Java-based network-centric computing platform. SORCER provides a service oriented architecture, which enables the implementation of parallel algorithms in a dynamic distributed computing environment. SORCER is often used for multidisciplinary aircraft design analysis and optimization. However, the current approach often assigns intense optimization algorithms to run entirely on single overloaded nodes, rather than evenly distributing the workload. The goal of this work is to provide lower-level optimization algorithms as integrated SORCER services and study the overhead of doing so. VTDIRECT95, a Fortran 95 implementation of D. R. Jones' algorithm DIRECT, is a highly parallelizable derivative-free deterministic global optimization algorithm. QNSTOP is a parallel quasi-Newton algorithm for stochastic optimization problems. The potential benefit of integrating VTDIRECT95 and QNSTOP into the SORCER framework is to provide dynamic load balancing among computational resources at the optimization level, resulting in a dynamically scalable process.

Keywords: service-oriented computing, deterministic global optimization, stochastic optimization, multi-disciplinary design

1 INTRODUCTION

This paper discusses the implementation and overhead of integrating two global optimization algorithms, VTDIRECT95 and QNSTOP, into a SORCER framework. SORCER is a large-scale, distributed computing environment for high fidelity multidisciplinary design optimization (MDO). The algorithms VTDIRECT95 and QNSTOP were chosen because of their relevance to aerospace engineering and their scalability on distributed computing applications. The process of integrating VTDIRECT95 and QNSTOP with SORCER is described in detail. The added overhead of the SORCER service is then assessed for an aircraft design application implementing VTDIRECT95 and QNSTOP on a SORCER grid.

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

Aerospace systems today exhibit strong interdisciplinary interactions and require a multidisciplinary, collaborative approach (Raymer 2006). Multidisciplinary design optimization aims to achieve an optimal design over several disciplines. The first step in the design process, conceptual design, often requires optimization with a large number of design variables belonging to multiple disciplines. Traditional conceptual design is carried out over a large set of configurations with low fidelity models, and suffers from poor accuracy. However, new physics based modeling tools used with high end computing resources can provide accurate multiphysics analysis and in the early stages of design. The drawback of these high fidelity models is that they are often prohibitively complex.

High performance computing (HPC) systems are critical for complex large scale design studies (Kodiyalam et al. 2004). While HPC systems deliver high computational power (capability computing) or high throughput (capacity computing), they are static resources with little scalability or flexibility. Service-oriented architecture (SOA) addresses the challenges faced by HPC systems in terms of scalability, availability, flexibility, and reliability. SOA not only incorporates the features of HPC systems, but also promises a world of orchestrated services by creating dynamic processes and agile applications that span platforms and organizations(Georgakopoulous and Papazoglou 2008).

SORCER, a Java based network centric computing framework (maintained by SORCERsoft.com, a subsidiary of SMT S. A. group), is a federated service-to-service (S2S) metacomputing environment that treats service providers as network peers with well-defined semantics of a federated service object-oriented architecture (Raghunath 2015). SORCER provides a platform for high fidelity multidisciplinary design optimization, combining models from various disciplines into one integrated model. SORCER accommodates dynamic distribution of service providers and on-demand provisioning of resources, resulting in significant speedups and effective utilization of computational resources.

VTDIRECT95 and QNSTOP were chosen for implementation on a SORCER grid because of their relevance to aircraft design problems. VTDIRECT95, a massively parallel Fortran 95 implementation of D. R. Jones' algorithm DIRECT, is widely used in MDO (Gao et al. 2013, Ghommem et al. 2012, Mehmood et al. 2011). QNSTOP is a class of parallel quasi-Newton methods for stochastic optimization and deterministic global optimization. QNSTOP for stochastic optimization problems synthesizes ideas from numerical optimization and response surface methodology, and demonstrates potential for stochastic robust design optimization and stochastic MDO problems.

The paper is organized as follows. Section 2 outlines the SORCER framework and the two optimization algorithms, VTDIRECT95 and QNSTOP. Section 3 presents details about conversion of VTDIRECT95 and QNSTOP to SORCER services. The study of an aircraft design application using VTDIRECT95 and QNSTOP as SORCER services is presented in Section 4. Section 5 briefly discusses the results of this work.

2 BACKGROUND

2.1 SORCER Overview

Service-oriented computing is a computing paradigm that utilizes self-describing, platform-agnostic services as the fundamental constructs to support rapid, cost-effective composition of distributed applications (Papazoglou et al. 2007). Services are self-adapting, dynamic processes that effectively communicate with one another to perform user-requested tasks in a distributed computing environment. The service-oriented computing paradigm, derived from the SOA model, allows interoperability, reusability, and loose coupling of its components in a dynamic environment, where computer resources are assigned to services as and when necessary. As indicated in Figure 1, the interaction between software agents is facilitated by message exchanges between service providers and service requestors. The service provider determines a description for a service and publishes it to a service discovery agency. This, in turn, is made discoverable to a service requestor. To invoke a service, the service requestor retrieves the service description from a registry and binds with the service provider based on the service description. In short, SOA addresses the challenges of distributed computing by enabling service discovery, integration, and use (Georgakopoulous and Papazoglou 2008).

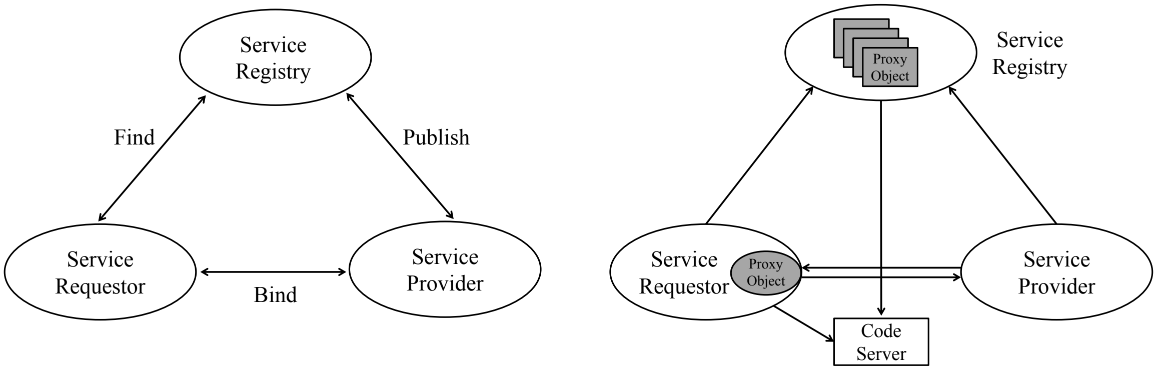


Figure 1: Service-oriented architecture (SOA) (left) vs. service object-oriented architecture (SOOA)(right).

SORCER is based on the concepts of SOA and also incorporates features of the service object-oriented architecture (SOOA), where service providers are objects accepting remote invocations (Raghunath 2015). As shown in Figure 1, the service requestor binds to the service provider by creating a proxy for remote communication. SOOA permits great flexibility in terms of communication between agents. These proxies, known as smart proxies, grant access to local and remote resources, regardless of who initially created the proxy. In SORCER, providers broadcast their availability, registries intercept broadcasted announcements and cache proxy objects to their service providers (Raghunath 2015). The SORCER operating system (SOS) looks up proxies by sending queries to registries and making selections from the available services. In short, providers use discovery/join protocols to publish services in the network, and SOS uses discovery/join protocols to obtain services in the network. From an object-oriented programming point of view, service providers are represented as independent network objects, locating each other via service registries and communicating through protocols such as remote method invocation (RMI), simple object access protocol (SOAP), common object request broker architecture (COBRA), etc.

Further, SORCER introduces three layers of converged programming abstractions: exertion-oriented programming (EOP), var-oriented programming (VOP), and var-oriented modeling (VOM) (Raghunath 2015). The EOP abstraction manages object-oriented distributed system complexity introduced by the complex network of metacomputers. VOP is a paradigm based on dataflow principles where changing the value of a var automatically forces recalculation of the interdependent values of vars. VOM, a modeling paradigm using vars, defines heterogeneous multidisciplinary var-oriented models in large scale multidisciplinary models. Thus, the SORCER framework incorporates the power of object-oriented programming and exertion-oriented programming to create an infrastructure that is modular, extensible, and reusable.

Based on successful implementation of large scale engineering applications with SORCER (Raghunath 2015), several desirable features related to multidisciplinary aircraft analysis and design optimization are as follows:

- Large scale, distributed, decentralized: SORCER dynamically federates processes and smartly distributes the load across all machines in the network.
- Leveraging the power of HPC: SORCER provides the features and computing power of HPC and SOA to form a dynamic distributed engineering collaboration platform.

- Reusability: The incorporation of object-orient modularity enables a high level of reuse when moving from one study to the next.
- Cost effective: SORCER accommodates physics based modeling via HPC for faster evaluation of higher fidelity configurations at the preliminary level of design when compared to traditional practices.
- Better utilization of computational resources: SORCER enables collaborative design studies across organizational boundaries and maximum utilization of all compute resources on the network, ranging from personal computers to high performance computing machines.
- Distributed resource management: SORCER employs Jini Connection technology (now called Apache River) with its JavaSpaces service to implement computational resource management across the network. The JavaSpaces technology facilitates the implementation of a self-load limiting grid computing system that can dynamically grow and shrink during the course of an optimization study (Freeman, Hupfer, and Arnold 1999). The loosely coupled space-based service federation allows asynchronous communication between computers in the network in a reliable manner (Raghunath 2015).

2.2 VTDIRECT95

VTDIRECT95 is a Fortran 95 software package using massively parallel dynamic data structures to implement the algorithm DIRECT by Jones, Perttunen, and Stuckman (1993). The algorithm DIRECT (DIviding RECTangles) is a deterministic global optimization algorithm that performs Lipschitzian optimization without the Lipschitz constant, and can be classified as a derivative free direct search algorithm.

Let E^n denote real n-dimensional Euclidean space, $D = \{x \in E^n \mid \ell \leq x \leq u\}$ be a box in E^n, and $f : D \to E$ a Lipschitz continuous function. The problem is to find a global minimum point $\bar{x}$ of f over D, $f(\bar{x}) = \min_{x \in D} f(x)$. The original (serial) algorithm by Jones, Perttunen, and Stuckman (1993)is described in six steps as below:

Step 1 (initialization): Normalize the feasible set D to be the unit hypercube. Sample the center point c_i of this hypercube and evaluate $f(c_i)$. Initialize $f_{min} := f(c_i)$, evaluation counter $m := 1$, and iteration counter $t := 0$.

Step 2 (selection): Identify the set S of "potentially optimal" boxes (subregions) of D. A box is potentially optimal if, for some Lipschitz constant, the function value within the box is potentially smaller than that in any other box (a formal definition with parameter ϵ is given by Jones, Perttunen, and Stuckman (1993)).

Step 3 (sampling): For any box $j \in S$, identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length. Sample the function at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.

Step 4 (division): Divide the box j containing c into thirds along the dimensions in I, starting with the dimension with the lowest value of $w_i = \min\{f(c + \delta e_i), f(c - \delta e_i)\}$, and continuing to the dimension with the highest w_i. Update f_{min} and m.

Step 5 (iteration): Set $S := S \setminus \{j\}$. If $S \neq \emptyset$, go to Step 3.

Step 6 (termination): Set $t := t + 1$. If iteration limit or evaluation limit has been reached, stop. Otherwise, go to Step 2.

VTDIRECT95 has numerous modifications from DIRECT in order to improve performance and load balancing on large scale parallel systems. The massively parallel implementation VTDIRECT95 distributes data among processors to share the memory burden imposed by storing all current boxes. The parallel scheme for SELECTION concentrates on distributing data among multiple masters to share the memory burden. Functional parallelism for SAMPLING is achieved by fully distributed control allocating function evaluation tasks to workers. A detailed discussion of the implementation of the serial and parallel subroutines in VTDIRECT95 is presented in He, Watson, and Sosonkina (2009).

2.3 QNSTOP

QNSTOP is a class of quasi-Newton methods for stochastic optimization with variations for deterministic global optimization (Amos et al. (2014b)). In iteration k, QNSTOP methods compute the gradient vector $\hat{g}_k$ and Hessian matrix $\hat{H}_k$ of a quadratic model

$$\hat{m}_k(X - X_k) = \hat{f}_k + \hat{g}_k^T(X - X_k) + \frac{1}{2}(X - X_k)^T \hat{H}_k(X - X_k)$$

of the objective function f centered at X_k, where $\hat{f}_k$ is generally not $f(X_k)$. QNSTOP methods progress by

$$X_{k+1} = \left[X_k - [\hat{H}_k + \mu_k W_k]^{-1}\hat{g}_k\right]_\Theta,$$

where μ_k is the Lagrange multiplier of a trust region subproblem, W_k is a scaling matrix, and $[\cdot]_\Theta$ denotes projection on the feasible set Θ. The exact steps taken at each iteration are outlined below:

Step 0 (initialization): Given a function evaluation budget $\widetilde{B}$ per start point and operating mode (deterministic or stochastic), set values for $\tau_0 > 0$, $\mu_0 > 0$, $\gamma \geq 1$, $\eta \geq 0$, $\zeta \geq 0$, N, X_0, $k := 0$, $W_0 := \hat{H}_0 := I_p$. It is recommended to run QNSTOP multiple times from different starting points.

Step 1 (regression experiment): Compute the ellipsoidal design regions given by

$$E_k(\tau_k) = \left\{X \in E^p : (X - X_k)^T W_k (X - X_k) \leq \tau_k^2\right\}$$

where τ_k is decayed at some rate depending on the mode. Next, uniformly sample $\{X_{k1}, \ldots, X_{kN}\} \subset E_k(\tau_k) \cap \Theta$, where Θ denotes the feasible set, and observe the response vector Y_k, where y_{ki} is modeled by $y_{ki} = \hat{f}_k + X_{ki}^T\hat{g}_k + \epsilon_{ki}$, with ϵ_{ki} accounting for the lack of fit. Finally, compute the least squares estimate for the gradient $\hat{g}_k$ using

$$\left(D_k^T D_k\right)\hat{g}_k = D_k^T Y_k.$$

where D_k denotes the absolute deviations of X_{ki}.

Step 2 (secant update): If $k > 0$, compute the model Hessian matrix $\hat{H}_k$ using BFGS (deterministic) or SR1 variant (stochastic) update.

Step 3 (update iterate): Calculate the next iterate X_{k+1}. In the deterministic case, X_{k+1} is the solution to the optimization problem

$$\min_{X \in E_k(\rho_k)} \; \hat{g}_k^T(X - X_k) + \frac{1}{2}(X - X_k)^T \hat{H}_k(X - X_k).$$

In the stochastic case, X_{k+1} is obtained by directly updating the Lagrange multiplier μ_k as described in Castle (2012), using the update:

$$X_{k+1} = X_k - \left[\hat{H}_k + \mu_k W_k\right]^{-1} \hat{g}_k$$

In both cases the computed point X_{k+1} is projected onto the feasible set Θ.

Step 4 (update subsequent design ellipsoid): Compute an updated scaling matrix $W_{k+1} \in W_\gamma$ as described in Castle (2012)and Amos et al. (Amos et al. (2014b)).

Step 5: If $(k+2)(N+1)+1 < \widetilde{B}$ then increment k by 1 go to **Step 1**. Otherwise, the algorithm terminates. (f is also observed at each ellipsoid center X_k.)

The algorithm QNSTOP has three significant sources of parallelism: the individual function evaluations, the loop over the samples in an experimental design, and the loop over the start points. A master-slave paradigm is a reasonable approach if the individual function evaluations are large scale parallel simulations. On large shared memory systems, ample parallelism is exhibited at the two outer nested loops — the loop over the start points and the loop over the samples. A detailed discussion of the serial and parallel implementations of QNSTOP can be found in Amos et al. (2014b). An analysis of a serial Fortran 95 implementation of QNSTOP is presented in Amos et al. (2014a).

2.4 Discussion

In the context of ever increasing parallelism, higher dimensions, and multidisciplinary design optimization, algorithms like VTDIRECT95 (for deterministic global optimization) and QNSTOP (for stochastic optimization) are excellent candidates for SORCER services. Objective function cost is one of the key parameters that affects the parallel performance under different parallel schemes. High parallel efficiency involves balancing communication overhead with the distribution of evaluation tasks for good load balancing (He et al. 2009a, He et al. 2009b). While SORCER has no control of the definition and granularity of the tasks, it *can* provide robust distributed parallelization and load balancing across computational resources, thus significantly speeding up the evaluation of objective functions in a dynamically scalable metacomputing environment.

3 IMPLEMENTATION AS SORCER SERVICES

3.1 JNI Wrappers

SORCER leverages the power of distributed computing through the use of Java interoperability, Jini, and web services (Raghunath 2015). The adoption of Java as a language for numerical computing presents difficulties. Some obstacles include: overrestrictive floating point semantics, inefficient support for complex numbers and alternative arithmetic systems, and lack of direct support for true multidimensional arrays (Boisvert et al. 2001). Moreover, the task of manually converting existing code in Fortran to Java-based services is both daunting and expensive (Liang 1999).

The Fortran 95 implementations of optimization algorithms considered in this paper, VTDIRECT95 and QNSTOP, incorporate advanced Fortran features that flexibly organize the data on a single machine, effectively reduce the local data storage, and efficiently share the data across multiple processors (He, Watson, and Sosonkina 2009). While Fortran is effective for numerical computing, Java provides flexibility and scalability for dynamic grid-based network architectures. In order to cope with the heterogeneity imposed by various programming languages, Java wrappers for the existing legacy code have been implemented using the JNI (Java Native Interface) libraries.

In developing the wrappers for the existing Fortran 95 implementations of VTDIRECT95 and QNSTOP, a feature of the JNI called the *invocation interface* was used. The invocation interface allows a regular non-Java program running on the native operating system to invoke a JVM to gain access to Java classes and features (Liang 1999). The invocation interface allows developers to embed a JVM implementation into native applications. Native applications can link with a native library that implements the JVM, and then use the invocation interface to execute components written in the Java programming language (Lindsey, Tolliver, and Lindblad 2010). Further, a C or C++ layer (often referred to as “glue code”) is required to gain access to codes written in Fortran.

3.2 Design Analysis with SORCER services

The concept of a service provider, or simply ‘provider’, is the crux of an engineering analysis or design study using SORCER. A provider is Java code that makes a number of Java methods (services) available to users over a network. Each provider is implemented in accordance with the principles of exertion-oriented programming (EOP), where an *exertion* is an object that represents a process by specifying the relationship between services and the information passed between them.

A provider is published on the network using SORCER. The provider's service may then be accessed via a small Java code called a *service requestor*. An individual request running on a single provider is called a *Task*. The input/output data associated with a task exectution is called a *Context*.

Providers are of two kinds — *analysis providers* and *model providers*. Providers that leverage existing domain-specific codes are referred to as analysis providers (Burton, Alyanak, and Kolonay 2012). An 'analysis provider' is an entity that neatly wraps the underlying domain-specific code with Java code so the domain-specific code can be accessed as a service by a remote user. The domain-specific code is generally platform independent and performs the bulk of the engineering-specific computations for a given service. Model providers are explained in the context of optimization problems over the remainder of this section.

In SORCER terminology, a *model* is a collection of service-oriented variables called *vars*. A var is defined by a triplet ⟨*value, evaluator, filter* ⟩, where

- a *value* is an expression yielding a valid quantity;
- an *evaluator* defines the process of how data is produced via remote services, or produced locally;
- a *filter* reduces the data generated by the evaluator to the value of the var.

In this way, SORCER leverages the power of var-oriented programming (VOP) to handle large sets of interconnected variables and does so in accordance with the var-based modeling paradigm var-oriented modelling (VOM).

In the context of optimization, these vars are the design variables and the implementation of the objective and constraint functions. Var instances are used to model both independent and dependent variables in SORCER. While independent vars are used as a container to store a value and perform no calculations, dependent vars implement mathematical functions. When published on the network, these dependent and independent vars that define a specific optimization problem are referred to as a *model provider*. The model provider is characterized by a single state and behaves like shared memory to users over the network.

For each objective function evaluation, a query object containing the name of the model provider, the design variable var names and values, and the var names of the objective function that the user wishes to calculate is constructed. The corresponding published model provider receives the query object and invokes the *setValue* method on its design variables and subsequently the *getValue* method on the user-specified objective function. The query object is then returned to the user with the updated values. To obtain the most recently updated value of the dependent var (the user-specified objective function), the model invokes the *evaluator* function. The evaluator checks the values of its arguments every time it is called, and will only recalculate its dependent vars if it detects a change.

3.3 Subroutines as a SORCER service

3.3.1 Serial Subroutines

For the serial subroutines VTdirect and QNSTOPS, platform independent executables are implemented using JNI (as described in Section 3.1) and tightly coupled with the provider's service.

3.3.2 pVTdirect

Unfortunately, results for pVTdirect (the massively parallel implementation of DIRECT in the package VTDIRECT95) under SORCER are not presented here because pVTdirect is fundamentally incompatible with efficient usage of the SORCER/JavaSpace/table model query paradigm (described in Sections 3.3.3 and 4.1). This paradigm assumes a master-slave parallel computing paradigm, and is not valid for a fully distributed algorithm such as pVTdirect.

3.3.3 QNSTOPP

The parallel (OpenMP) implementation (subroutine QNSTOPP) of QNSTOP incorporates three sources of parallelism: (1) the loop over the start points, (2) the loop over the experimental design samples, or (3) both. For compatibility with SORCER, QNSTOPP is modified at the level of the loop over the experimental design samples such that the function evaluations are chunked in function evaluation calls to SORCER. In this case, QNSTOPP interacts with the published model provider via a table model query. Rather than passing a single design point to the model provider, the JNI wrapper constructs a table containing the name of the design vars and their values for a set of sample points. As with the case of a single model query, a query object containing the name of the model provider, the design variable var names and values, and the var names of the objective function is constructed. The model provider, on receiving the query object, creates new child instances for each row in the table for parallel execution of the table row evaluations. The model provider creates a thread for each child instance and begins to *setValue* and *getValue* on the vars. On completion, the var values for each run are returned to the JNI wrapper in a table object and the child models are discarded.

4 EXPERIMENTS AND RESULTS

The framework *EBF3PanelOpt* facilitates the structural optimization of curvilinearly stiffened panels by considering a number of constraints that have to be satisfied (buckling, von Mises stress, and crippling constraints). The framework, written in Python, interacts with the commercial software MSC Patran (for geometry and mesh creation) and MSC Nastran (for finite element analysis). Given the input parameters and design variables, the script then creates the appropriate session file and submits it to MSC Patran to create the geometry and mesh of the stiffened panel, with which MSC Nastran then carries out a finite element analysis producing the inputs to an optimizer. More details about the framework EBF3PanelOpt can be found in Mulani, Slemp, and Kapania (2013). By decomposing an aircraft wing into multiple local panels bordered with curvilinear spars and ribs, aircraft designers can utilize EFB3PanelOpt to minimize the structural weight of these panels and subsequently the overall wing weight.

This section presents the implementation and results for optimization of curvilinear blade-stiffened panels using VTDIRECT95 and QNSTOP. The section is further divided into two subsections — the Implementation of EBF3PanelOpt as a SORCER service, and the performance results for the optimization of a stiffened panel.

All experiments presented here are conducted on Intel (i7-3770) machines running at 3.4 GHz, each with 16GB of memory and a single quad-core processor, in which each core supports hyperthreading. The experiments are conducted using GNU Fortran 4.9.1, GNU C 4.9.1, Python 2.6.6, Open MPI 1.8.1, and Java 1.8.0_25 on x86_64 running CentOS 6.6. The framework EBF3PanelOpt is configured to use Nastran 2014 and Patran 2014. For all QNSTOPP runs with SORCER presented in this paper, two identical machines are used. The program carrying out optimization as a service and the model provider are started on one machine, and the EBF3PanelOpt provider on the other.

4.1 EBF3PanelOpt as a Service

In order to achieve truly distributed objective function evaluations, the optimization framework EBF3PanelOpt is implemented as an analysis provider. Analysis providers can be dynamically distributed over a variety of computational resources with the help of SORCER's JavaSpaces technology. The JavaSpaces technology provides a type of shared memory where exertion evaluators can drop tasks to be processed by service providers, which use Jini discovery mechanisms to find them on the network. JavaSpaces not only enables computers on the network to communicate reliably, but also provides load balancing capability to cope with dynamic resources. Hence, computing resources can be added during the course of an optimization study, thereby enhancing productivity.

For the parallel implementation of QNSTOP that constructs a table of runs (a modification of the OpenMP parallel code QNSTOPP), the EBF3PanelOpt provider is configured to have a fixed number of worker threads. The number of worker threads determines the number of tasks a provider can process in parallel. The providers are started on multiple machines to distribute the work. During optimization, the model provider first creates child instances for every row in the table and drops these tasks into the space. Then, based on the number of worker threads, each EBF3PanelOpt provider picks up unprocessed tasks from the space and executes them in parallel.

An undocumented alternative to JavaSpaces is Catalog, referred to here as SORCER/Catalog, that has advantages in certain contexts, such as multicore or single large parallel distributed memory machines. Here, service providers publish proxies to the catalog. The requestor passes a service request to the catalog, which "matches" the service request with one of the proxies, which is then passed to the requestor, who uses the proxy to make the remote call directly to the provider (as in Figure 1).

A detailed description of the implementation is presented in Raghunath (2015)with an experiment that recreates the designs reported in Mulani, Slemp, and Kapania (2013).

4.2 Experiment

In this section, a simply supported flat rectangular panel is optimized for minimum mass using VTDIRECT95 and QNSTOP. The performance results for the optimization of curvilinear blade-stiffened panels containing two stiffeners (Case 1) and four stiffeners (Case 2), using the subroutines VTdirect, pVTdirect, QNSTOPS, and QNSTOPP with and without SORCER are presented below. The design variables for both problems and the exact settings used with the subroutines VTdirect, pVTdirect, QNSTOPS, and QNSTOPP are presented in Raghunath (2015).

For optimization with (the serial subroutine) VTdirect and (the parallel subroutine) pVTdirect, the stopping condition is a limit of 1000 on the number of objective function evaluations. For pVTdirect, the number of processes is set to 4. Since pVTdirect is incompatible with SORCER/JavaSpace/table model query (as described in Section 3.3.2), results for these cases are omitted.

For optimization with (the serial subroutine) QNSTOPS, deterministic mode is used with 5 start points and a budget of of 200 evaluations each (1000 total). Additionally, for JNI runs without SORCER, (the parallel subroutine) QNSTOPP is parallelized over just the sampling point objective function evaluations with 4 OpenMP threads. Recall from section 3.3.2, QNSTOPP for SORCER does not use OpenMP to achieve parallelism, but instead interacts with the model provider via a table model query. For all QNSTOPP runs, the table is configured with four rows, resulting in 4 concurrent objective function evaluations.

To calculate the parallel efficiency of QNSTOPP with and without SORCER, the parallel efficiency measure E_p was used. Note that the modified QNSTOPP described in Section 3.3.3 is used for runs with SORCER:

$$E_p = \frac{\big((\text{QNSTOPS time})/(\text{QNSTOPP time})\big)}{(\text{total number of threads})}$$

In Tables 1, 2, and 3 "script robustness" refers to a Java utility `GenericUtil` separate from SORCER, recommended for use of scripts in production distributed computing, that increases the robustness of scripts and communication links across different operating systems.

4.2.1 Performance Results

The execution times for pVTdirect, VTdirect, QNSTOPS, and QNSTOPP, with and without SORCER, are listed in Tables 1 (Case 1) and 2 (Case 2). The last column in the table represents the parallel efficiency with QNSTOPP.

Table 1. Execution time in seconds for pylon wing panel optimization with 2 stiffeners.

	VTdir	pVTdir	QNSTOPS	QNSTOPP	E_p
SORCER and script robustness	13,009	N/A	11,388	3,545	0.80
SORCER w/o script robustness	8,957	N/A	7,994	2,542	0.79
SORCER/Catalog w/o script robust.	8,487	N/A	7,597	2,458	0.77
W/o SORCER, w/o script robust.	8,460	2,924	7,560	2,309	0.82

Table 2. Execution time in seconds for pylon wing panel optimization with 4 stiffeners.

	VTdir	pVTdir	QNSTOPS	QNSTOPP	E_p
SORCER w/ script robustness	14,450	N/A	10,370	3,676	0.71
SORCER w/o script robustness	10,384	N/A	7,451	2,697	0.69
SORCER/Catalog w/o script robust.	9,815	N/A	7,088	2,615	0.68
W/o SORCER, w/o script robust.	9,786	3,789	7,052	2,408	0.73

In Case 2, note that the numbers of function evaluations for VTdirect and QNSTOP* (1165 and 825 respectively) are different than those for Case 1 (1131 and 950 respectively). Even though the problem size doubled (from 13 to 25), the parallel efficiencies decreased because the number of function evaluations by QNSTOPP was less for Case 2 than for Case 1.

For 100 runs of VTdirect, the average computational expense of each objective function evaluation is listed in Table 3, where n is the problem dimension. The SORCER with script robustness overhead per function evaluation ($\approx 4\,s$) is about the same for both problem sizes, and without script robustness the SORCER overhead is negligible. Robustness and portability do not come cheap.

Table 3. Objective function evaluation times in seconds for pylon wing panel (2 & 4 stiffeners).

	$n = 13$	$n = 25$
With SORCER and script robustness	11.13	12.90
With SORCER, without script robustness	7.36	9.14
Without SORCER and script robustness	7.32	9.10

5 DISCUSSION

VTDIRECT95 and QNSTOP were implemented as services on a SORCER grid, facilitating the optimization of curvilinearly stiffened panels in a truly distributed manner. Source code for the service wrappers is in Raghunath (2015). From Table 3, the SORCER with script robustness overhead is about four seconds (essentially all due to script robustness). Tables 1 and 2 show that the other SORCER overhead is not significant for expensive function evaluations. The conclusion is that the right SORCER paradigms (JavaSpaces, though the most general approach, is but one of several) must be used in the right contexts, and no single SORCER paradigm is a general purpose solution to parallel and distributed computing for MDO. On the continuum of distributed computing technology (MPI, Globus, Legion, Condor, SORCER), SORCER is at the heavyweight end.

The use of JNI to wrap the corresponding native code provided a clean and elegant interface between the optimization algorithm and the Java block that evaluates the objective for a design point. However, the JNI development overhead is relatively high. It should also be mentioned that the installation of SORCER is far from routine—SORCER is currently a research code with limited documentation and requires considerable knowledge of network computing to install and utilize.

ACKNOWLEDGEMENTS

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8650-09-2-3938. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government. The EBF3PanelOpt code was developed under a research contract from NASA Fundamental Aeronautics Program to Virginia Polytechnic Institute and State University with Karen M. B. Taminger as the Program Manager.

REFERENCES

Amos, B. D., D. R. Easterling, L. T. Watson, B.S. Castle, M. W. Trosset, and W. I. Thacker. 2014a. "Fortran 95 Implementation of QNSTOP for Global and Stochastic Optimization". In *22nd High Performance Computing Symposium (HPC 2014)*. Tampa, Florida, Society for Computer Simulation International.

Amos, B. D., D. R. Easterling, L. T. Watson, W. I. Thacker, B. S. Castle, and M. W. Trosset. 2014b. "Algorithm XXX: QNSTOP: Quasi-Newton Algorithm for Stochastic Optimization". Technical Report 2014-07, Virginia Polytechnic Institute and State University, Blacksburg, VA.

Boisvert, R. F., J. Moreira, M. Philippsen, and R. Pozo. 2001. "Java and Numerical Computing", *IEEE Computing in Science and Engineering* vol 3(2), pp. 18–24.

Burton, S. A., E. J. Alyanak, and R. M. Kolonay. 2012. "Efficient Supersonic Air Vehicle Analysis and Optimization Implementation using SORCER". In *AIAA 2012- 5520, 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. Indianapolis, Indiana, American Institute of Aeronautics and Astronautics and International Society for Structural and Multidisciplinary Optimization.

Castle, B. S.. 2012. *Quasi-Newton Methods for Stochastic Optimization and Proximity-based Methods for Disparate Information Fusion*. Ph.D. thesis, Indiana University, Bloomington, IN.

Freeman, E., S. Hupfer, and K. Arnold. 1999. *JavaSpaces Principles, Patterns, and Practice*. Boston, MA, Addison Wesley Longman, Inc.

Gao, D. Y., L. T. Watson, D. R. Easterling, W. I. Thacker, and S. C. Billups. 2013. "Solving the Canonical Dual of Box- and Integer-constrained Nonconvex Quadratic Programs via a Deterministic Direct Search Algorithm", *Optimization Methods and Software* vol. 28(2), pp. 313–326.

Georgakopoulous, D. and M. P. Papazoglou. 2008. *Service-Oriented Computing*. Cambridge, The MIT Press.

Ghommem, M., M. R. Hajj, B. K. Stanford, L. T. Watson, and P. S. Beran. 2012. "Global and Local Optimization of Flapping Kinematics". In *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. Honolulu, Hawaii, American Institute of Aeronautics and Astronautics.

He, J., A. Verstak, M. Sosonkina, and L. T. Watson. 2009a. "Performance Modeling and Analysis of a Massively Parallel DIRECT: Part 2", *International Journal of High Performance Computing Applications* vol. 23(1), pp. 29–41.

He, J., A. Verstak, L. T. Watson, and M. Sosonkina. 2009b. "Performance Modeling and Analysis of a Massively Parallel DIRECT: Part 1", *International Journal of High Performance Computing Applications* vol. 23(1), pp. 14–28.

He, J., L. T. Watson, and M. Sosonkina. 2009. "Algorithm 897: VTDIRECT95: Serial and Parallel Codes for the Global Optimization Algorithm DIRECT", *ACM Transactions on Mathematical Software (TOMS)* vol. 36(3), Article No. 17.

Jones, D. R., C. D. Perttunen, and B. E. Stuckman. 1993. "Lipschitzian Optimization without the Lipschitz Constant", *Journal of Optimization Theory and Application* vol. 79(1), pp. 157–181.

Kodiyalam, S., R. J. Yang, L. Gu, and C. H.. 2004. "Multidisciplinary Design Optimization of a Vehicle System in a Scalable, High Performance Computing Environment", *Structural and Multidisciplinary Optimization* vol. 26(3/4), pp. 256–263.

Liang, S.. 1999. *The Java Native Interface: Programmer's Guide and Specification*. MA, Addison Wesley Longman Inc.

Lindsey, C. S., J. S. Tolliver, and T. Lindblad. 2010. *JavaTech, an Introduction to Scientific and Technical Computing with Java*. Cambridge, Cambridge University Press.

Mehmood, A., I. Akhtar, M. Ghommem, M. R. Hajj, and L. T. Watson. 2011. "Optimization of Drag Reduction on a Cylinder Undergoing Rotary Oscillations". In *52nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*. Denver, Colorado, American Institute of Aeronautics and Astronautics.

Mulani, S. B., W. C. H. Slemp, and R. K. Kapania. 2013. "EBF3PanelOpt: an Optimization Framework for Curvilinear Blade-stiffened Panels", *Thin Walled Structures* vol. 63, pp. 13–26.

Papazoglou, M. P., P. Traverso, S. Dustdar, and F. Leymann. 2007. "Service-Oriented Computing: State of the Art and Research Challenges", *Computer* vol. 40(11), pp. 38–45.

Raghunath, C.. 2015. *Service Oriented Computing Environment (SORCER) for Deterministic Global and Stochastic Optimization*. M.S. thesis, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA.

Raymer, D. P.. 2006. *Aircraft Design: A Conceptual Approach*. New York, American Institute of Aeronautics and Astronautics Education Series.

AUTHOR BIOGRAPHIES

CHAITRA RAGHUNATH (M.S., Virginia Tech) is a Network Engineer at Hughes Network Systems. Her interests include distributed and high performance computing.

TYLER H. CHANG is a first-year Ph.D. student advised by Dr. Layne T. Watson. He is a Cunningham and Davenport Fellow with interest in numerical analysis and parallel algorithms.

LAYNE T. WATSON (Ph.D., Michigan, 1974) has interests in numerical analysis, mathematical programming, bioinformatics, and data science. He has been involved with the organization of HPCS since 2000.

MOHAMED JRAD (Ph.D., Virginia Tech) is currently a postdoctoral fellow in the department of Aerospace and Ocean Engineering at Virginia Tech. His interests include computational structural mechanics, aeroelasticity, and multidisciplinary design optimization.

RAKESH K. KAPANIA (Ph.D., School of Aeronautics and Astronautics, Purdue University, 1985) is the Mitchell Professor of Aerospace & Ocean Engineering at Virginia Tech. His research interests are: computational structural mechanics, aeroelasticity and aero-servo-elasticity, artificial neural networks, and multidisciplinary design optimization. He is the winner of the 2016 AIAA/ISSMO Multidisciplinary Design Optimization Award.

RAYMOND M. KOLONAY (Ph.D.) is the Director of Multidisciplinary Science & Technology Center within AFRL's Aerospace Systems Directorate. Dr. Kolonay is an AFRL Fellow, ASME Fellow, Fellow of the RAeS, and an Associate Fellow of the AIAA.

IMPLICANT BASED SOLVER FOR XOR BOOLEAN LINEAR SYSTEMS

Jayashree Katti
Department of Information Technology
Pimpri Chinchwad College of Engineering
Pune,India
jayashree.katti@gmail.com

Virendra Sule
Department of Electrical Engineering
Indian Institute of Technology
Bombay, India
vrs@ee.iitb.ac.in

B.K.Lande
Vasantdada Patil College of Engineering
Bombay,India
bklande@gmail.com

ABSTRACT

An approach is presented for solving linear systems of equations over the Boolean algebra $B_0 = \{0,1\}$ based on implicants of Boolean functions. The approach solves for all implicant terms which represent all solutions of the system. Traditional approach to solving such linear systems is to consider them over the field $GF(2)$ and solve either by Gaussian elimination or Lanczos methods. One of the unfinished problems in Computer Science is that of developing scalable parallel solvers for such systems. The proposed approach based on implicants has inherent parallel structure for computation in terms of independent threads. We show that for sparse systems with a fixed bound on number of variables in any equation and using sufficient parallel resource, this approach requires $O(n)$ time where n is the number of variables. Hence this approach is expected to provide a scalable solution to the problem of solving large Boolean linear systems over large number of processors.

Keywords: XORSAT, Implicants.

1 INTRODUCTION AND MOTIVATION

Boolean Satisfiability problems arise in many applications such as cryptology, hardware and software verification, reliability, artificial intelligence, decision under logic constraints, computational studies of Biological networks (Crama and Hammer 2011), (Bolouri 2008) ,(Alan Veliz-Cuba and Laubenbacher 2014). In Computer Science, the problem of deciding satisfiability of Boolean formulas in Conjunctive Normal Form (CNF) known as CNF-SAT has been of central importance (Schoning and Toran 2013). These problems (known broadly as Boolean SAT problems) are concerned with deciding consistency (or existence of solutions) of Boolean equations in several variables. XOR-SAT is one special case of Boolean SAT where each equation is an exclusive OR (XOR) combination of variables.

Such linear XOR systems naturally appear in problems such as quadratic sieve method for prime factorization of numbers (Das 2016). Also in decoding of linear error correction coding, linear XOR systems

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

need to be solved while optimizing the weight of solutions. Problems of finding all solution assignments with minimum Hamming weight, with maximum weight and of fixed weight are of different nature than the traditional problems of deciding satisfiability. All of these problems are addressed if the approach is aimed at finding all satisfying solutions. While solving general non linear polynomial equation systems, the XOR linear systems in terms of monomials can be solved as an intermediate problem (this is known as the XL approach to solving multivariate systems (Bard 2009)). Solving XOR Boolean systems stands out as a problem on its own and performances of algorithms aimed at solving such problems are relevant for understanding performances of solvers for general problems (Sule 2014). Although the XOR linear system problem is known to be of class P (as compared 3-CNF SAT of class NP complete), search for algorithms which scale to solving XOR problem of large sizes is as much important as scalability and performance of solvers for NP complete problems arising in practice by parallel algorithms.

The aim of this paper is to develop an approach for solving XOR linear systems over the Boolean algebra B_0 with a view to address following two objectives. This approach is based on implicant computation of Boolean formulas recently announced in (Sule 2016). In this paper we present the application of the ideas for the XOR linear case.

1. Finding all solutions of the system. This is not addressed by the known SAT approaches which are concerned with deciding satisfiability (or the existence of a solution). This problem of representing all solutions is of higher complexity than the satisfiability problem (Crama and Hammer 2011),(Desai and Sule 2014) in case of 2-CNF SAT problems. Moreover in applications such as Cryptography or Biological networks, satisfiability (or consistency) of the system is already known and it is required to find all solutions of the system. We shall follow the approach to represent all solutions in terms of implicants of equations as proposed in (Sule 2016).
2. Developing an approach which has inherent parallelism and can scaleup for solving large size problems over large number of processors. An important unresolved issue with solving Boolean equations is developing a solver which can scaleup with good efficiency for solving large systems arising in applications by parallel computation. Scalability of parallel solvers also depends on the algorithm and is affected by the number of processors. Our approach gives a method of computation in multiple parallel threads and is expected to have good scalability even over large number of processors. In fact its parallel performance improves with increased parallel resource.

Most systems arising from real life applications are sparse, i.e. each clause has only a small fraction of the complete set (large number) of variables. Such equations may have randomly distributed variables in each equation or in certain situation such as factorization of numbers dominant variables for small prime factors. While only local variables may be present in an equation when the variables have space dependent features. Gaussian elimination based algorithms cause loss of sparsity as computation progresses. On the other hand the implicant based approch increases sparsity due to substitutions.

Performances of other algorithms such as Grobner basis algorithm concerning scalability leave much desired as pointed out in (Bard 2009, Sule 2013). A general survey of parallel SAT solvers (Hammadi and Wintersteiger 2012) discusses many issues of scalability which are yet to be resolved. A limitation of SAT solvers is also that these are mainly designed for deciding satisfiability of CNF formulas and unless general systems are transformed to this form these methods are not applicable. The problem of solving Boolean equations is of considerable interest to Biological regulatory networks and is being studied from both theoretical and applied angle (Alan Veliz-Cuba and Laubenbacher 2014, Zou 2014). These references show that this problem is of current interest and hence it is important to continue search of new methods for solving Boolean equations which can scale up over large sizes of systems as well as large number of parallel processing elements. In short it is desirable to develop solvers for Boolean systems which provide inherent parallelism in computation. In this paper we propose such a method for solving such problems associated with XOR systems.

1.1 Notations and background

The Boolean algebra referred in this paper is the two element algebra $B_0 = \{0,1,+,.,'\}$ with binary operations $+,.$ denoting the well known *OR* (disjunction) and *AND* (conjunction) while $'$ denotes the complement operation. The Boolean ring $\{0,1,\oplus,.\}$ with $\oplus$ denoting the well known XOR shall also be denoted by B_0. The Boolean ring under $\oplus$ is equivalent to the binary field $GF(2)$. Two element Boolean algebra or ring are very well known and apart from the change of notation for operations we shall refer (Brown 2003) for their theory. Boolean functions $f : B_0^n \to B_0$ of n variables (denoted X) are equivalence classes of formal conjunctions and disjunctions of n-variables $x_i, i = 1,\dots n$ and their complements x_i'. Such formal expressions when evaluated by assigning values of arguments from B_0 define Boolean functions. Boolean functions themselves form a Boolean algebra denoted $B_0(n)$. For a Boolean function f the set of all *satisfying assignments* is the set of points a in B_0^n such that $f(a) = 1$. This set is denoted by $S(f)$. A term in X is a function

$$t(X) = \prod_{1 \le i \le n} x_i^{\alpha_i} \ \alpha_i \in \{0,1\}$$

where for a variable x, $x^\alpha = x$ when $\alpha = 1$ and $x^\alpha = x'$ when $\alpha = 0$. The set of indices i in a term t shall be called its *support* and denoted $\sup(t)$. Clearly

$$S(t) = \{x_i = \alpha_i \forall i \in \sup(t), x_i = D \forall i \ni \sup(t)\}$$

where D denotes an arbitrary assignment. Hence we represent the set $S(t)$ by the compact notation (t) which denotes the partial assignments for $x_i, i \in \sup(t)$ in $S(t)$.

1.2 Implicants and representation of set of all satisfying assignments

An implicant of a Boolean function $f(X)$ is a term $t(X)$ such that $t \le f$ in the Boolean algebra of functions $B_0(n)$. The substitution of partial assignments (t) in f is denoted as f/t and is known as the *ratio* or *cofactor* of f by t. We observe the obvious result,

Proposition 1. If $f(X)$ is a Boolean function and t a term in X then following statements follow the implication $1) \Rightarrow 2) \Rightarrow 3) \Rightarrow 1)$.

1. t is an implicant of f.
2. $f/t = 1$.
3. $(t) \subset S(f)$.

A set of implicants $I(f)$ of f is said to be *complete* if $f(a) = 1$ for some a then there exists a t in $I(f)$ such that $t(a) = 1$. Hence when $I(f)$ is complete for f we have the equivalent expressions as given in the following,

Proposition 2. Following statements are equivalent

1. $I(f)$ is a complete set of implicants.
2. $S(f) = \bigcup_{t \in I(f)} (t)$
3. $f = \sum_{t \in I(f)} t$

where the sum in the third expression is an OR sum of implicants t in I.

Proof. 1) $\Leftrightarrow$ 2). The inclusion $\bigcup_{t\in I(f)}(t) \subset S(f)$ follows from the definition of implicant. Conversely, let $a \in S(f)$, then since $I(f)$ is complete there exists a $t \in I(f)$ such that $t(a) = 1$. Hence

$$S(f) \subset \bigcup_{t\in I}(t)$$

1) $\Leftrightarrow$ 3). By definition

$$\sum_{t\in I(f)} t \leq f$$

But since $I(f)$ is complete, if $f(a) = 1$ there is an implicant t such that $t(a) = 1$. Hence

$$\sum_{t\in I(f)} t(a) = f(a)$$

from which the equivalence follows. □

Above proposition is a basis of our algorithm for computing all solutions of XOR systems.

1.3 XOR-SAT and associated problems

The linear XOR-SAT problem without constraints on solutions, in n variables over B_0 is defined by a system of m equations of the form

$$\bigoplus_{j=1}^{n} a_{ij}x_j = b_i, i = 1,\ldots,m \tag{1}$$

where a_{ij}, b_i are elements of B_0. The problem is to find all n-tuples $a = (a_1,\ldots,a_n)$ in B_0^n such that each a gives a solution assignment $x_i = a_i$. The basic XOR-SAT problem we consider is 1) to find all assignments for $X = \{x_1,\ldots,x_n\}$ in B_0^n which satisfy equations (1). Such assignments when they exist (i.e. when the system is satisfiable) are finite in number. There are important associated problems. If $w : B_0^n \to \mathbb{W}$ is a non-negative integer valued function (representing weight of an assignment) then we have the associated problems 2) to find all solutions a of the system such that $w(a) < q$ where q is a specified non-negative number and 3) to find all solutions a of the system such that $w(a)$ is minimum. Clearly if we solve the problem 1) then the associated problems are solvable by search over the solution set. Our approach to represent the solutions in terms of implicants makes such a search feasible.

One of the central issues with these problems is that although the number of solutions are always finite, the number grows exponentially in the number of free assignments of variables in each solution. Hence solutions of the above problems need to be compactly represented rather than just enumerated. In fact simply enumerating the finite set of assignments is not practically feasible in large sized problems. Clearly the best way to represent such solution sets are by collecting the terms corresponding to variables which have fixed assignments since variables with free assignments need not be explicitly shown in a solution. This way the exponential number of solutions can be represented compactly by fixed variable assignments. The representation of $S(f)$ as in Proposition 2 provides such a compact representation for satisfying assignments of a Boolean function f. We extend this representation for the solution set of the systems of XOR linear equations. Next, the problem of scalability can also be addressed by this compact way to represent assignments. For instance the implicants representing any solution of the system must necessarily also be implicants for solving a single equation. Hence an algorithm which at each step restricts search of assignments over small number of variables involved in a single equation and carries out independent search along parallel threads can provide scalability of computation. Our approach to solving the XOR linear problem is based on these ideas.

2 IMPLICANT BASED APPROACH FOR SOLUTION OF BOOLEAN SYSTEMS

Our approach to the problem of representing all solutions of the system (1) depends on constructing a complete set $I(E)$ of all implicants corresponding to an equation E of the system. Let E denote an equation in the linear system of the form

$$a_{j1}x_{j1}+a_{j2}x_{j2}+\ldots+a_{jk}x_{jk}=b_j$$

where $1 \leq j1 \leq j2 \leq \ldots \leq jk \leq n$. Consider a Boolean function $f(x_{ji})$ in variables x_{ji} whose set of satisfying assignments is

$$S(f)=\{x_{ji}\in B_0|E \text{ is satisfied }\}$$

Then it follows that a complete set of implicants $I(f)$ characterizes the set of all satisfying assignments for the equation E. We shall thus denote this set of implicants as $I(E)$ and call it a *complete set of implicants* of equation E and call this function as the *true value function* of equation E. Thus the set $I(E)$ represents all solutions of the equation E denoted as $S(E)$. Then for each t in $I(E)$, (t) denotes a set of partial assignments satisfying E, hence

$$S(E)=\{\bigcup_{t\in I(E)}(t)\}$$

this is then a compact way to represent all solutions (or the satisfying assignments) of an equation E.

2.1 Satisfying assignments of simultaneous equations

Now consider the problem of representing satisfying assignments of two simultaneous equations in variables X. So let E_1 and E_2 be these two equations. If we compute a complete set of implicants $I(E_1)$ then the two simultaneous equations are consistent iff for all t in $I(E_1)$ the substitution E_2/t equivalently f_2/t does not result into a contradiction $f_2/t=0$ (or E_2 not satisfied). We can formally write the following.

Proposition 3. Two simultaneous equations E_1 and E_2 are consistent iff $f_2/t \neq 0$ (i.e. E_2 is satisfied) for some t in a complete set of implicants $I(E_1)$. A complete set of implicants $I(E_1,E_2)$ of the simultaneous system of equations is given by either of the sets

$$\begin{aligned} I(E_1,E_2) &= \bigcup_{t\in I(E_1),s\in I(E_2/t)}\{ts\} \\ &= \bigcup_{s\in I(E_2),t\in I(E_1/s)}\{ts\} \end{aligned}$$

Proof. Let $I(E_1)$ be a complete set of implicants of equation E_1. Then each t in $I(E_1)$ represents partial assignment (t) in the set of all solutions of E_1. Hence the simultaneous equations have a solution iff there is a partial assignment (t) whose satisfying set $S(t)$ intersects the solution set of E_2. This is true iff when (t) is substituted in E_2 does not lead to contradiction. This is the condition $f_2/t \neq 0$. When there is no contradiction the resulting equation is E_2/t. If this equation has a satisfying partial assignment s in the remaining variables then ts is a simultaneous implicant of both equations representing a partial assignment $(t)(s)$. Taking union over all implicants t in $I(E_1)$ for which E/t is not a contradiction thus gives the formula $I(E_1,E_2)$. Since the order of equations leaves the solutions invariant the formula is symmetric. □

This proposition is the basis of our algorithm to compactly represent all satisfying assignments of simultaneous equations. We first consider few examples.

2.2 Examples of representing solutions by implicants

Example 1. Consider the two equation system

$$\begin{aligned} w\oplus x\oplus y &= 1\\ x\oplus y\oplus z &= 0 \end{aligned}$$

For first equation E_1 we have

$$I(E_1)=\{wx'y',w'xy',w'x'y,wxy\}$$

This gives

$$\begin{aligned} E_2/wx'y' &\Leftrightarrow \{z=0\}\\ E_2/w'xy' &\Leftrightarrow \{z=1\}\\ E_2/w'x'y &\Leftrightarrow \{z=1\}\\ E_2/wxy &\Leftrightarrow \{z=0\} \end{aligned}$$

Since no contradiction took place the system is consistent and every implicant of E_1 leads to a solution. Hence we have

$$\begin{aligned} I(E_1,E_2) &= \{wx'y'z',w'xy'z,\\ &\quad w'x'yz,wxyz'\} \end{aligned}$$

which gives the set of all solution assignments satisfying the simultaneous equations.

Consider another example of a consistent system.

Example 2.

$$\begin{aligned} w\oplus x\oplus y &= 1\\ w\oplus y &= 1 \end{aligned}$$

The first equations is same as above hence has same $I(E_1)$. We compute the substitutions in the second equation

$$\begin{aligned} E_2/wx'y' &\Leftrightarrow \{1=1\}\\ E_2/w'xy' &\Leftrightarrow \{0=1\}\\ E_2/w'x'y &\Leftrightarrow \{1=1\}\\ E_2/wxy &\Leftrightarrow \{0=1\} \end{aligned}$$

Two of these are contradictions. For the other two $E_2/t=1$ hence t is an implicant of E_2 also. Hence we have

$$I(E_1,E_2)=\{wx'y',w'x'y\}$$

2.3 Notation for larger examples

We shall now introduce a notation to represent larger systems and implicants. Consider the system of linear equations.

$$\begin{aligned} x_3\oplus x_5\oplus x_7 &= 1\\ x_1\oplus x_4\oplus x_5 &= 0\\ x_2\oplus x_5\oplus x_6 &= 1\\ x_3\oplus x_7 &= 0\\ x_5\oplus x_7 &= 0 \end{aligned}$$

We represent the above system denoted S as the set

$$\{[3,5,7,0,1],[1,4,5,0,0],[2,5,6,0,1],[3,7,0,0],[5,7,0,0]\}$$

(This notation is inspired by notation for CNF clauses well known in DIMACS notation but is not the same). We also denote an implicant term of the type $t = x'_3 x_5 x'_6 x_7$ as well the partial assignment (t) by the notation $(-3,5,-6,7)$. Also the partial assignment of a product of two implicants ts is denoted as $(t)(s)$. In the above system S, consider first an equation with minimum number of variables

$$[3,7,0,0]$$

A complete implicant set for this equation is

$$\{(-3,-7),(3,7)\}$$

Let S/t denote the system obtained by substitution of assignment $t = 1$ in all equations of S. Then (equations which are trivially satisfied and result in tautologies $0 = 0$ or $1 = 1$ are dropped from the new system)

$$\begin{array}{l} S/(-3,-7) = \\ \{[5,0,1],[1,4,5,0,1],[2,5,6,0,1],[5,0,0]\} \\ S/(3,7) = \\ \{[5,0,1],[1,4,5,0,1],[2,5,6,0,1],[5,0,1]\} \end{array}$$

In both the resulting reduced systems we have implicant of first equation as $\{(5)\}$. Substitution of this implicant gives rise to two new reduced systems

$$\begin{array}{c} \{[1,4,0,0],[2,6,0,0],[1,0,0]\} \\ \{[1,4,0,0],[2,6,0,0],[1,0,1]\} \end{array}$$

The first system has an inconsistent equation $1 = 0$ while the second system has no contradiction. Hence the complete set of satisfying assignments are represented by the implicant set by taking the product of the previous implicant with that of the second system

$$\begin{array}{l} \{(3,7)(5)(1,4),(3,7)(5)(-1,-4), \\ (3,7)(5)(2,6),(3,7)(5)(-2,-6)\} \end{array}$$

3 PROPOSED ALGORITHM TO FIND ALL SOLUTIONS

We now propose the algorithm for representing all solutions of a system of equations S in terms of an implicant set or return an empty set if the system is inconsistent. The algorithm starts with selecting an equation E called a pivot equation. (A suitable choice of a pivot is an equation with minimum number variables). A complete set of implicants denoted $I(E)$ is then computed. All satisfying assignments of E are represented by all partial assignments satisfying these implicants. The system of equations is then reduced to S/t for a selected implicant t. The processes is repeated until a contradiction is reached when an equation in the reduced system is contradicted when evaluated at an implicant or else an augmented implicant set is returned. The final sets of implicants returned in each thread determine the partial assignments of all satisfying assignments of the system. This is described in the following pseudocode of the algorithm 1.

4 TIME COMPLEXITY AND RESULTS ON RANDOM CASES

Performance of the above XOR system solver has been evaluated on systems $Ax = b$ where matrices A are over the Boolean ring B_0 and nonsingular of size n. The vector x is an n tuple of variables to be solved and b is a known n tuple. The operation Ax uses the Boolean ring operations of product and XOR sum. Systems of different sizes varying from $n = 40$ to $n = 500$ are selelcted in two different sets in which A is chosen non-singular. Such random non-singular matrices are chosen by randomly transforming a matrix in Hessenberg form as described below while b is a random vector. All these systems have a unique solution since A is non singular by design.

Algorithm 1: All Solutions: (Reduction of a system by Implicants)

```
   Input  : The linear system of equations $S : f_k(X) = b_k$ ($k$-th equation denoted as $E_k$), $k = 1, 2, \dots n$, in
            variables $X = \{x_1, \dots, x_n\}$.
   Output: set of all solution assignments.
 1 Choose an equation $E$ in $S$ with least number of variables;
 2 Compute a complete set of implicants $I(E)$ of $E$ and order it as $\{t_1, \dots, t_{N_k}\}$;
 3 Select lowest order implicant $t$ in $I(E)$;
 4 Start thread:
 5 Reduce the system: pick equations from $S$ of indices $j$ and compute $f_j/t$.
 6 if $f_j/t = 0$ for some $j$ then
 7 |   End thread;
 8 |   Discard the implicant;
 9 |   Select next implicant in the order in 3 and restart thread;
10 else
11 |   Compute the reduced equations $f_j/t$ for all $j$;
12 |   Denote the reduced system as $S/t$;
13 |   Delete the equations for $j$ such that $f_j/t = 1$;
14 end
15 if all $f_j/t = 1$ then
16 |   Return $(I)(t)$;
17 else
18 |   set $S \leftarrow S/t$ $I \leftarrow (I)(t)$.
19 end
20 Repeat 6 with this $I$ untill $S = \emptyset$;
   /* The set $I$ is the set of implicants denoting partial assignment of
      solutions of $S$ in the current thread started by the implicant.     */
21 Collect union of partial assignments given by $I$ in all threads;
22 This gives the set of all solution assignments;
```

4.1 Procedure for generating matrix A

Matrix A is generated by making random elementary transformations on a non singular Boolean matrix in Hessenberg form. For instance a 5×5 non-singular matrix in Hessenberg form is

$$H = \begin{bmatrix} * & 1 & 0 & 0 & 0 \\ * & * & 1 & 0 & 0 \\ * & * & * & 1 & 0 \\ * & * & * & * & 1 \\ 1 & * & * & * & * \end{bmatrix}$$

where $*$ denotes an arbitrary 0 or 1 entry. Similar construction is considered to generate a general non-singular matrix in Hessenberg form. Then the A matrix is obtained by random elementary row operations on H. This procedure is carried out in following steps.

1. set counter $cnt = 0$.
2. Randomly generate i from the range 1 to n.
3. Randomly generate j from the range 1 to n.
4. If $i \neq j$ then
   (a) add elements of row i to row j i.e.
   $Row(j) = Row(j) + Row(i)$. Elementary addition are over binary field $\mathbb{F}_2$.
   (b) make $cnt = cnt + 1$
5. If $i = j$ then Go To step 2.
6. steps 2 to 5 were followed until $cnt \geq (n * 10/100)$

4.1.1 Analysis of complexity of solving the system and experimental test cases

Algorithm 1 essentially involves two computational steps at every stage when a pivot equation is chosen. These are as follows.

1. Generation of a complete set of implicants $I(E)$ for the pivot equation.
2. Substitution of the partial assignment (t) of an implicant $t \in I(E)$ in the rest of the equations to get reduced system S/t.

These two steps define a thread segment. The starting point of this thread is an implicant t which is to be used for reduction of the system. At the end of the thread implicant t is either discarded due to a contradiction or is qualified as an implicant of the system or leaves a reduced system. All these thread segments are independent computations. Hence the reduction of the original system progresses along parallel threads. If we assume that there is sufficient parallel resource available then at each stage of start of a thread segment the computations can be carried out on independent processors. Hence the time required for solving the XOR system with sufficient parallel resource is equal to the time required for the thread requiring longest time consisting of a sequence of thread segments. We assume that the time for implicant computation in each thread segment is constant (this is justified for sparse systems in which number of nonzero entries in any equation is small and bounded from above). Similarly the time taken for substitution of a partial assignment in equations is also considered constant (upper bound). In each thread segment at least one variable assignment is discovered. Hence a longest thread has atmost n steps. Hence the time taken for longest thread is of order $O(n)$ asssuming all thread segments are exeuted independendly (or in parallel).

4.2 Experimental test case results

Experimental cases of computation with this algorithm are documented in the following table.In these random samples of systems, for each n, 10 binary random matrices A of size $n \times n$ and 10 n-tuple vectors b of were selected. The systems $Ax = b$ were solved sequentially by the above algorithm and time taken for each thread segment were measured. From these records of thread segment time, the time required for solving the system when all threads could be executed in parallel was calculated. This is the time taken for the sequence of thread segments which takes the longest time, while the time required to solve each system sequentially is the sum of times taken for all thread segments. From these measurements the averages of time taken for the longest threads and total were calculated for the random samples. These are plotted in the following table shown. The graph shows approximately linear $O(n)$ trend as expected. The slight trend visible of the type $O(n^{1+\alpha})$ in the average time for longest threads is due to the fact that the experimental test cases had nonzero terms in any equation equal to half of n. Hence the time taken for computing implicant sets in the thread segments was not constant as assumed but was actually weakly $O(n^{\alpha})$ for $\alpha < 1$ with a small constant. Hence the longest thread time appears to tend towards $O(n^{1+\alpha})$.

As described above our computations are completely sequential. However due to the independence of thread segments through which the algorithm works it is theoretically possible to make estimates of parallel efficiency. If it is assumed that an infinite parallel resource is available for implementing the parallel threads, then the maximum speedup possible is the ratio

$$\begin{aligned}\text{speedup} &= \frac{\text{Time Taken for Sequential Solution}}{\text{Time Taken for longest Thread}} \\ &= \frac{\text{Sum of Time taken for all threads}}{\text{Time taken for the longest thread}}\end{aligned}$$

The table also document this maximum speedup in terms of average time taken for longest thread and the sequential solution. The table also show that speedup is higher if number of threads are larger. Although this means larger requirement of memory and parallel processors, this also gives a positive indication that this algorithm is expected to be scalable for large data and large number of processors.

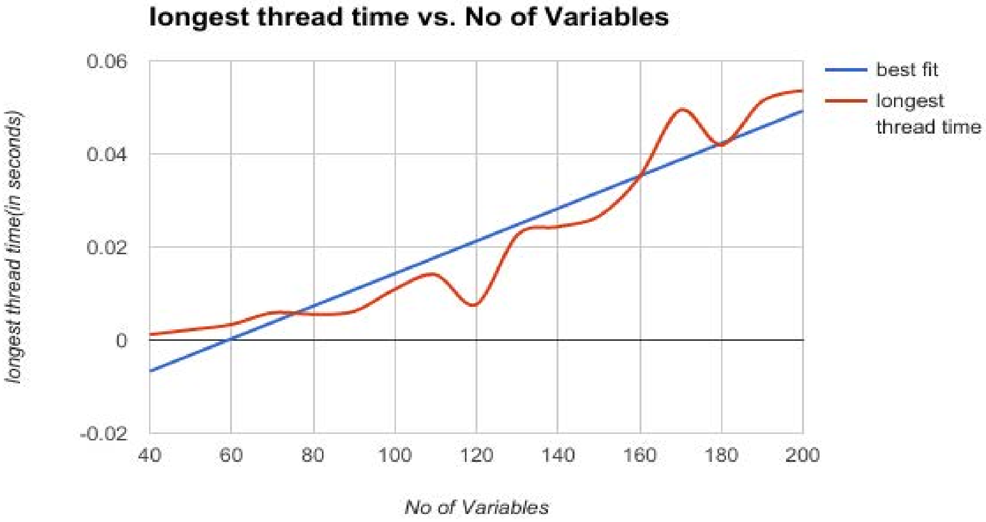


Figure 1: Longest Thread time Vs No of Variables.

Table 1: Experimental Results for all solutions

Number of Variables	Average Number of Threads	Average Time Taken for longest Thread	Average Time Taken for Sequential Solution	Maximum speedup predicted by parallel computation
40	5	1.18E-03	5.64E-03	8
50	9	2.22E-03	1.64E-02	8
60	9	3.35E-03	2.41E-02	11
70	13	5.85E-03	5.83E-02	11
80	15	5.55E-03	9.16E-02	17
90	30	6.14E-03	1.85E-01	35
100	19	1.09E-02	1.74E-01	22
110	51	1.41E-02	3.76E-01	53
120	41	7.64E-03	3.86E-01	76
130	122	2.26E-02	1.57E+00	150
140	233	2.43E-02	2.96E+00	137
150	153	2.66E-02	2.82E+00	130
160	176	3.51E-02	3.72E+00	139
170	203	4.94E-02	7.89E+00	152
180	190	4.19E-02	6.02E+00	160
190	393	5.13E-02	1.18E+01	230
200	274	5.36E-02	1.87E+01	350

5 CONCLUSION

A parallel solver for special Boolean systems called XOR linear systems is presented in this paper which represents all solutions of the system. Such systems are traditionally solved by Gaussian elimination over the binary fields or Lanczos methods. The proposed solver treats these as Boolean systems and all operations performed are Boolean. The algorithm splits the computation in terms of independent threads and hence when sufficient parallel resource is available, the algorithm gives an idea of the maximum speedup achievable by parallel computation. Parallel performance speed ups are tabulated for random systems and show promising speed ups. Although systems required to be solved in applications are non linear, the linear system speed up performance is relevant since the basic of operation of substitution of assignments does not get affected by linear nature of functions. Hence the results of this paper are of interest even for general non linear problems for solving Boolean systems.

REFERENCES

Alan Veliz-Cuba, Boris Aguilar, F. H., and R. Laubenbacher. 2014. *Steady state analysis of Boolean molecular network models via model reduction and computational algebra*. 221 ed. BMC Bioinformatics.

Bard, G. 2009. *Algebric Cryptanalysis*. Springer.

Bolouri, H. 2008. *Computational modeling of gene regulatory networks.* Imperial College Press.

Brown, F. M. 2003. *Boolean analysis: The logic of Boolean equations*. Dover.

Crama, Y., and P. Hammer. 2011. *Boolean functions. Theory, algorithms and applications Encyclopedia of Mathematics and its applications*, Volume 142. Cambridge.

Das, A. 2016. *Computational number theory*. CRC Press.Prentice-Hall, Inc.

Desai, M. P., and V. Sule. 2014. “Generalized cofactors and decomposition of Boolean satisfiability problems”. *arXiv.org/cs.DS/1412.2341v1*.

Hammadi, Y., and C. M. Wintersteiger. 2012. “Seven Challenges in Parallel SAT Solving”. *Challenge paper AAAI 2012 Sub-Area spotlights track. Association of Advancement of Artificial Intelligence.*.

Schoning, U., and J. Toran. 2013. *The satisfiability problem, algorithms and analyses*. Lehmanns media.

Sule, V. 2013. “Generalization of Boole-Shannon expansion, consistency of Boolean equations and elimination by orthonormal expansion”. *arXiv.org/cs.CC/1306.2484v3*.

Sule, V. 2014. “An algorithm for Boolean satisfiability based on generalized orthonormal expansion”. *arXiv.org/cs.DS/1406.4712v3*.

Sule, V. 2016. “Implicant based parallel all solution solver for Boolean satisfiability”. *arxiv.org/cs.Ds/1611.09590v3*.

Zou, Y. M. 2014. “An algorithm for detecting fixed points of Boolean networks”. *arXiv.org:1404.5515v1[q-bio.QM]*.

AUTHOR BIOGRAPHIES

JAYASHREE V. KATTI is an Associate Professor at Pimpri Chinchwad College of Engineering, Savitribai Phule Pune University. She is currently pursuing PhD in Computer Science and Engineering. Her research interests are Cryptography, Algebraic Cryptanalysis, Visual Cryptography, High performance computing. Her email address is jayashree.katti@gmail.com.

VIRENDRA SULE has been teaching at Indian Institutes of Technology for over 25 years and has published research in Control and Systems theory. Since recent times he has been working in Cryptography and computational algorithms for Boolean systems. For a brief period he worked with the TATA startup venture Computational Research Laboratories on parallel computing. His email address is viren.sule@gmail.com.

B.K.LANDE currently working as Professor in Narsee Monjee Institute of Management Studies, Mumbai. His research interests are Controls and Communication Engineering. His email address is bklande@gmail.com.

DESIGNING LARGE HYBRID CACHE FOR FUTURE HPC SYSTEMS

Jiacong He
Department of Electrical Engineering
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX, USA
Email: jiacong.he@utdallas.edu

Joseph Callenes-Sloan
Department of Electrical Engineering
The University of Texas at Dallas
800 W Campbell Rd, Richardson, TX, USA
Email: jcallenes.sloan@utdallas.edu

ABSTRACT

DRAM cache is a large cache stacked on the processors die using 3D-stacking technology, which may be used in the future High-Performance Computing (HPC) systems to reduce latency and increase bandwidth. However, the energy becomes an inevitable challenge with the increasing cache capacity. In this paper, we first propose a large hybrid cache for future HPC systems, which can effectively reduce the static energy compared with the DRAM cache. Further, we apply volatile STT-RAM as part of the hybrid cache to reduce both the static and dynamic energy of the DRAM cache. Finally, we propose to maintain the cache tag array in the region of the hybrid cache with less read latency to improve performance. Experimental results show our hybrid cache reduces energy by 31.6% and improves performance by 18.8% on average.

Keywords: DRAM cache, STT-RAM, performance, energy, HPC.

1 INTRODUCTION

Future high-performance computing demands large cache capacity and high memory bandwidth. However, the existing SRAM cache with low density hinders the increment of cache capacity, and the limited pin count leads to the memory bandwidth wall. Recently, the 2.5D/3D die-stacking technologies are widely used in the major processor vendors (e.g., Intel Xeon Phi Processor includes up to 16GB 2.5D-stacking DRAM memory). Also, DRAM has already been used as a large cache in commercial supercomputer (e.g., IBM POWER8 uses up to 128MB eDRAM as L4 cache per socket). Thus, many researchers (Loh and Hill 2011, Huang and Nagarajan 2014, Zhao et al. 2007) proposed to use 3D die-stacking DRAM as a last-level cache to increase the cache capacity and off-chip memory bandwidth. The DRAM cache consists of multiple layers of DRAM stacked on the processor die using Through-Silicon via (TSV). It is potential to meet the workloads demand of future HPC systems by increasing the on-chip cache capacity up to gigabytes of storage and providing orders of magnitude higher bandwidth.

However, die-stacking DRAM cache with large capacity suffers from high leakage power and becomes increasingly susceptible to error due to the process scaling. Also, the 3D design has more challenge in power and thermal management because multiple stacking layers result in higher power densities. Recently, Spin-Transfer Torque RAM (STT-RAM), as an emerging non-volatile memory technology, is potential to be used as a large cache due to its near-zero leakage and high density. However, STT-RAM has the disadvantage of high write energy and high write latency, so it can not directly substitute for DRAM cache without any optimizations. Thus, STT-RAM is commonly used in a hybrid cache to utilize the advantage of different memory technologies. For example, recent works (Li et al. 2011, Wu et al. 2009) leveraged non-volatile

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

STT-RAM to build a hybrid cache with SRAM. However, there are fabrication challenges of the hybrid cache in the conventional 2D design. We observe the emerging 3D stacking technology provides an excellent opportunity to build a large hybrid cache by integrating different wafers with different memory techniques.

While today's servers need tens to hundreds of gigabytes of DRAM each, the corresponding demand for die-stacked cache capacity varies between hundreds of megabytes to several gigabytes (Jevdjic et al. 2013). Thus, our proposed large hybrid cache also requires large tag storage (64MB tag array for 1GB cache) considering to use conventional 64B cache block. Ideally, the tag array should be stored in the SRAM cache to make the tag access latency as small as possible, while it is impractical due to the precious SRAM cache capacity. Some researchers proposed to store the tag array within the large on-die cache, which needs to optimize the latency of tag access by adding extra design complexity. We notice that the tag management policy in the large hybrid cache is also important, but previous works (Cong et al. 2011, Li et al. 2011, Wu et al. 2009) rarely considered this issue.

In this paper, we propose a large last-level hybrid cache for HPC systems, which consists of DRAM and STT-RAM regions. Each region can be composed of several layers stacked upon each other. The DRAM layers are used as the main component of the hybrid cache due to their high endurance. The STT-RAM layers with small leakage are used to reduce the static energy consumption of the hybrid cache. Also, we notice the STT-RAM can be relaxed (Smullen et al. 2011) to reduce its high write energy and latency by sacrificing its non-volatility. Thus, the non-volatile STT-RAM in the hybrid cache is replaced by volatile STT-RAM to further reduce both static and dynamic energy of the hybrid cache. Finally, we observe that there are two different tag array in the DRAM region and STT-RAM region of the hybrid cache respectively. And the read latency in these two regions is unbalanced due to the disparate memory technologies, thus we propose to move all tag array to the cache region (DRAM or STT-RAM) with lower read latency to improve the performance.

Overall, our contributions are as follows.

- We propose a large hybrid cache for future HPC systems to reduce static energy.
- We use volatile STT-RAM as part of the hybrid cache to reduce both static and dynamic energy.
- We optimize the tag management of the proposed hybrid caches to improve performance.

2 MOTIVATION

With the increasing frequency of CPU, more and more programs will be limited in the performance by the systems' memory bandwidth, rather than by the computational performance of the CPU. Also, the high-end computer spends over 90% of their time idle waiting for cache misses and fetching data from off-chip memory. Thus, the conventional DRAM memory with low bandwidth and high latency leads to the memory wall problem in the current shared-memory HPC systems. To handle the memory wall problem in HPC systems, die-stacking technologies have recently drawn much attention from the research community as a viable solution. Through die-stacking technology, DRAM can be stacked on top of the processor die (3D) or on a separate die connected through-silicon interposer (2.5D), providing an additional cache capacity to the traditional SRAM cache, much higher bandwidth and lower interconnect latency compared to off-chip DRAM memory.

Further, a typical supercomputer consumes prohibitively large amounts of electrical power for computing, while the demand for computing is increasing exponentially as a consequence of data explosion in the scientific computing. It is observed that Last Level Cache (LLC) has become a significant source of both static and dynamic energy consumption in modern processors, consuming up to 17% of total core energy. Thus, although DRAM cache as LLC can be used to alleviate bandwidth and latency pressure in the HPC systems, the large on-die cache exacerbates the energy challenge. To handle the power wall challenge in the

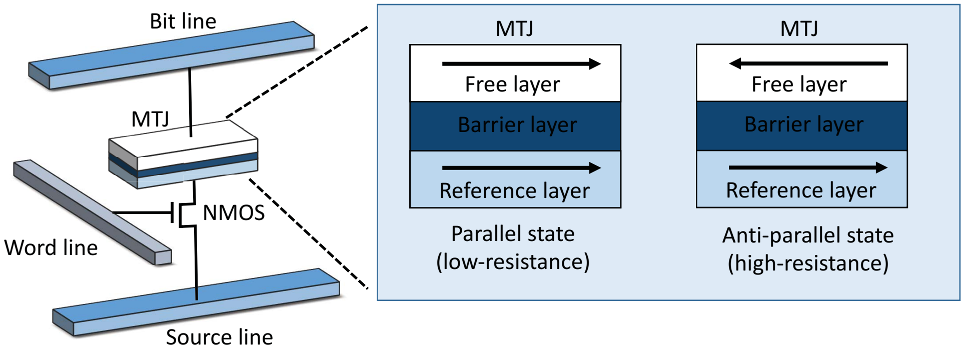


Figure 1: The structure of STT-RAM cell with 1MTJ 1T.

conventional cache, emerging non-volatile STT-RAM is being explored as potential alternatives of SRAM and eDRAM cache. A typical cell size of STT-RAM is 40F^2 compared with 146F^2 SRAM cell size, so STT-RAM is attractive to be used as a large LLC cache for area savings. Also, The read latency and energy of STT-RAM are comparable to SRAM and DRAM, and STT-RAM has near-zero leakage power and zero refresh energy. However, it is also noticed the STT-RAM is 2x worse in write latency and 10x worse in write energy compared with DRAM. Thus, STT-RAM can not be directly used to replace DRAM as a large cache based on the write-intensive nature of many scientific workloads.

To fully utilize the benefit of different memory techniques, we observe the large hybrid cache consisting of STT-RAM and DRAM can effectively reduce overall energy while maintaining performance at an efficient level. However, one key challenge in designing a large on-die cache is the cache tag management. We observe that the read latency is unbalanced in the hybrid cache, and the tag array access is actually a read operation, which can be utilized to optimize the hybrid cache performance by moving tag array to the cache region (DRAM or STT-RAM) with lower read latency.

3 BACKGROUND

STT-RAM. As shown in the Figure 1, the STT-RAM cell has an access transistor that connects the storage device and the bitline. It also has a Magnetic Tunnel Junction (MTJ) to store binary data, and the MTJ consists of two ferromagnetic layers and one tunnel barrier layer. The resistance of the MTJ is used to represent the binary data stored in the cell, which is determined by the relative magnetization direction of these two layers (Kultursay et al. 2013). And the low and high resistance are used to represent logical 0 and 1 respectively. Further, the data retention time of STT-RAM could be relaxed to reduce its high write energy by shrinking the planar area of the MTJ or decreasing the thickness of the free layer.

DRAM Cache. Previous researchers divide the DRAM cache into two categories. One is the block-based DRAM cache, which is architected as a large, software-transparent last-level cache (Loh and Hill 2011). It uses the 64B block size of conventional SRAM cache to optimize temporal locality. Thus, it requires a large amount of space for tag storage (16MB tag storage for 256MB DRAM cache). Another is the page-based DRAM cache design using a much larger cache block size of 2KB to 4KB (Jevdjic et al. 2014, Jevdjic et al. 2013). The tag overhead in block-based design is reduced to a few megabytes. However, a large cache line may fetch many unused data on a DRAM cache miss from the low-bandwidth off-chip memory.

Hybrid Cache. Different memory technologies have different characteristics of power, performance, and density. Hybrid cache integrates different memory technologies and achieves the overall optimal design. There are two types of hybrid cache architectures (Wu et al. 2009), one is the inter-cache design (every cache level of the cache hierarchy has disparate memory technologies) and the other is intra-cache design (single

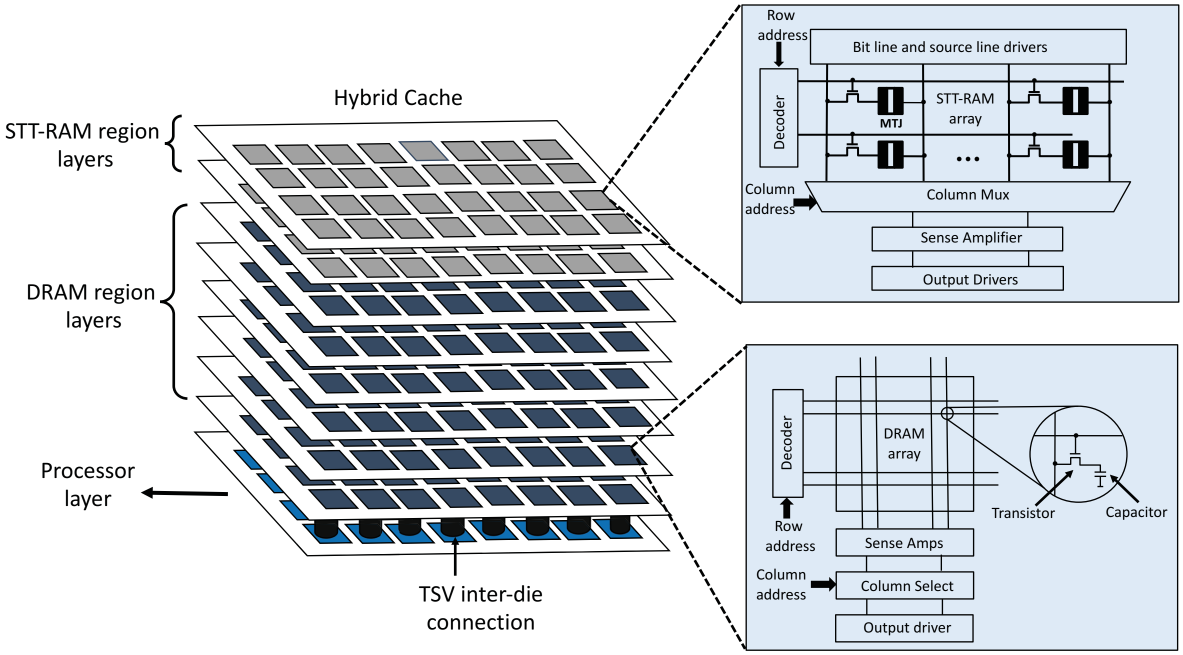


Figure 2: Proposed 3D-stacking hybrid cache architecture overview, with DRAM and STT-RAM cell array.

cache level consists of different memory technologies, and this is what our paper focuses on). Previously, many researchers proposed hybrid SRAM/STT-RAM cache and hybrid SRAM/DRAM cache (Wang et al. 2014, Cong et al. 2011), but little work discussed hybrid DRAM/STT-RAM cache. Different from prior work (He and Callenes-Sloan 2016), we focus on the evaluation of hybrid cache for the scientific and large-scale applications, especially based on the HPC systems.

4 HYBRID CACHE FOR FUTURE HPC SYSTEMS

4.1 Architectural Design Overview

We propose a large die-stacking hybrid cache consisting of DRAM region and STT-RAM region. The total hybrid cache capacity is 1GB, where DRAM region capacity is 768MB and STT-RAM region capacity is 256MB. The DRAM region is divided into 6 layers with 128MB per layer, and the STT-RAM region is divided into 2 layers with 128MB per layer. Every layer is stacked upon each other as illustrated in the Figure 2. The lowest die is the processors die containing 32 cores, each core has private L1 cache (64KB per core), L2 cache (4MB per core) cache and shared L3 cache (8MB). The DRAM region with 6 layers is stacked on the processors die, providing cache access performance due to shorter vertical wire connection. The STT-RAM region with 2 layers is stacked on the DRAM region providing static energy reduction. Once there is a data request from the CPU, the request is transmitted from the processors' layer to the last layer of the hybrid cache using low-latency TSV inter-die connection. The DRAM region is first accessed to buffer a large amount write request due to its large capacity, which can help to alleviate the high write energy pressure in the STT-RAM region.

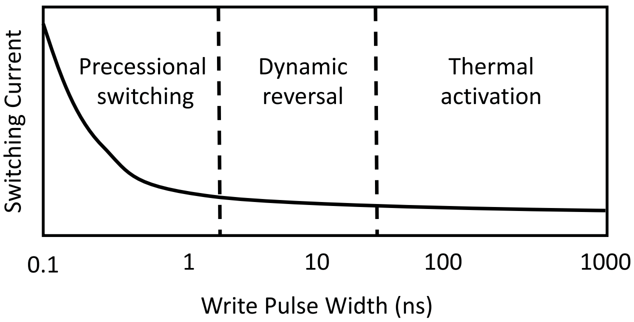


Figure 3: Three different working regions of MTJ.

Table 1: Relation between retention time and thermal factor at 278K.

Retention time	10 years	1 year	1 month	1 week	1 day	1 hour	1 min	1 s	10 ms
Δ	40.12	37.56	35.34	33.96	32.02	38.65	24.47	20.32	15.78

4.2 STT-RAM Optimization

Although emerging STT-RAM can efficiently reduce the static energy compared with conventional DRAM, its high write energy and latency still pose a large challenge in using STT-RAM in the large cache. In this paper, we propose to use disparate memory technologies (DRAM and STT-RAM) to build a large hybrid cache based on the 3D-stacking technique, which can effectively reduce both the static and dynamic energy in the large cache, and keep the original performance benefit. However, the HPC workloads and scientific applications usually are write-intensive with large datasets, so the STT-RAM used in the hybrid cache still need to be optimized to accommodate data-intensive workloads. It is noticed that the non-volatility of STT-RAM can be sacrificed to reduce its high write energy and latency. Thus, we optimize our hybrid cache by replacing non-volatile STT-RAM with volatile STT-RAM to achieve better energy and performance efficiency. In the following parts, we first analyze the non-volatility characteristics of STT-RAM, and then we show how to optimize hybrid cache using volatile STT-RAM in terms of energy and performance.

4.2.1 STT-RAM Non-volatility

Figure 1 depicts the structure of the STT-RAM cell array, where the STT-RAM cell is connected to word line (WL), bit line (BL) and source line (SL). The WL is used to select the specific row, and the voltage difference between SL and BL is used to complete write and read operation. When executing a read operation, a negative voltage is applied between SL and BL, and the current flowing through the free layer of the MTJ is sensed by the sense amplifier. To write data to an MTJ, a large current must be pushed through the MTJ to change the magnetic orientation of the free layer. Depending on the direction of the current, the free layer becomes parallel or anti-parallel to the fixed layer. The amount of current required for writing into an MTJ should be larger than a critical current.

MTJ has three regions, including the thermal activation region, dynamic reverse region and processional switching region (Diao et al. 2007). Their distribution is shown in the Figure 3, and the required switching current in each working region can be calculated by:

$$J_C^{THM}(T_{sw}) = J_{C0}(1 - \frac{1}{\Delta} ln(\frac{T_{sw}}{\tau_0})) \qquad (T_{sw} > 10ns) \tag{1}$$

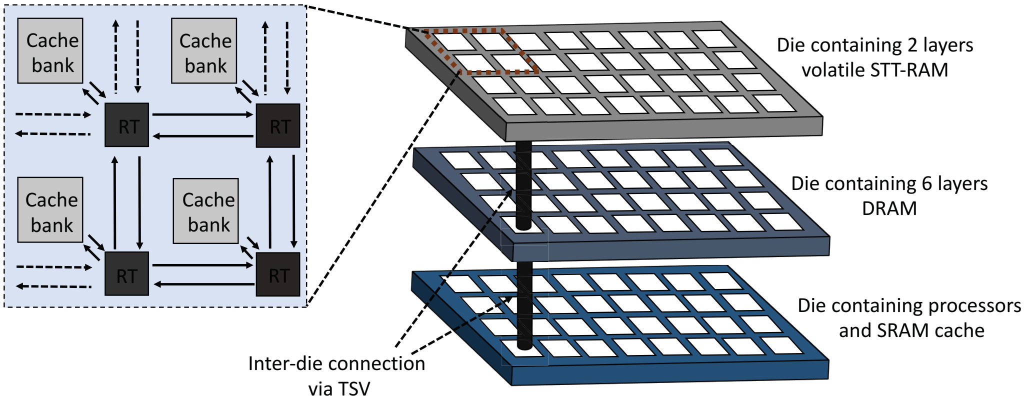

Cache bank
Cache bank
Cache bank
Cache bank
RT
RT
RT
RT
Die containing 2 layers volatile STT-RAM
Die containing 6 layers DRAM
Die containing processors and SRAM cache
Inter-die connection via TSV

Figure 4: 3D Hybrid cache including 1 volatile STT-RAM die, 1 DRAM die and 1 processors die. There are 32 processing cores, 32 cache banks per cache layer. Cache banks are connected through NoC routers.

$$J_C^{DYN}(T_{sw}) = \frac{J_C^{THM}(T_{sw}) + J_C^{PREC}(T_{sw})e^{-A(T_w - T_{PIV})}}{1 + e^{-A(T_w - T_{PIV})}} \qquad (10ns \geq T_{sw} > 3ns) \tag{2}$$

$$J_C^{PREC}(T_{sw}) = J_{C0} + \frac{ln(\frac{\pi}{2\theta})}{T_{sw}} \qquad (T_{sw} \leq 3ns) \tag{3}$$

where $J_C(T_w)$ is the required switching current density, J_{CO} is the threshold of the switching current density, T_{sw} is the switching pulse width, τ_0 is the relaxation time, Δ is the thermal stability of MTJ. The thermal stability of MTJ determines the retention time T_{ret} of STT-RAM (Diao et al. 2007), which can be modeled as: $T_{ret} = \frac{1}{f_0}e^{\Delta}$. Based on the analysis above we estimate the average time for MTJ bits flip. In the Table 1, we show the retention time changing with thermal factors at temperature 278K. Reducing the size of MTJ leads to shorter retention time, which provides larger storage density and less write energy of STT-RAM.

4.2.2 Hybrid Cache with Volatile STT-RAM

To design a hybrid cache with volatile STT-RAM, we need to first determine a suitable data retention time for the STT-RAM. As shown in Table 1, the retention time varies from year to millisecond, and lower retention time needs extra refresh operation like DRAM to keep data valid. On the one hand, if the retention time is selected too long, the high write energy and latency of STT-RAM can not be effectively reduced. On the other hand, if the retention time is selected too short, the refresh operation of volatile STT-RAM will lead to high refresh energy consumption. In order to choose an appropriate retention time that can balance the high write energy and extra refresh energy of the STT-RAM, we observe that the data refresh period of on-die DRAM can be used as a good reference for determining the retention time of STT-RAM.

It is known the refresh period of commodity DRAM is 64ms, which means the DRAM restores the degraded voltage stored in the DRAM cell capacitors for every 64ms due to DRAM volatile nature. For a detailed description of DRAM refresh, we refer the reader to (Liu et al. 2012). However, the refresh period of on-die DRAM cache is smaller than 64ms due to the usage of fast logic transistors, which have higher leakage than the DRAM memory. To reduce the design complexity of refresh circuit, the retention time of volatile STT-RAM should be close to the refresh rate of DRAM cache. Therefore, we conduct an application-driven study to analyze the refresh times of the DRAM cache blocks to determine a suitable data retention time. An extensive analysis of emerging workloads indicates that the average retention times for the DRAM cache

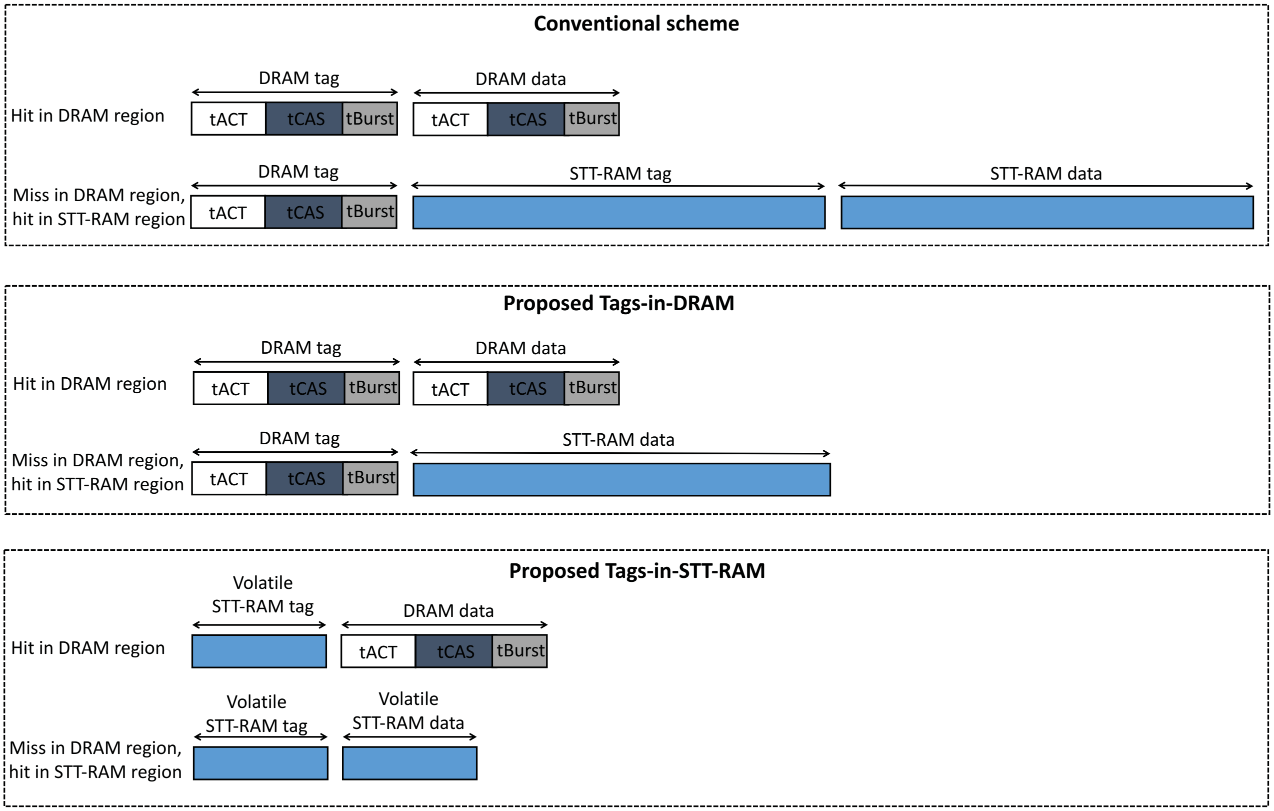


Figure 5: Access latency of different types of hybrid cache architecture.

blocks is close to 100μs. Although this aggressive refresh rate used in the STT-RAM will lead to over 95% write energy reduction, while the refresh energy is increased over 10x compared with 64ms retention time. To balance the write energy reduction and the overhead of refresh energy increment, we advocate the retention time of STT-RAM to be 0.5ms, which has about 90% write energy reduction and 5x refresh energy increment.

As shown in the Figure 4, our hybrid cache uses TSV as the vertical interconnection between hybrid cache and processor cores. In each core, there is a hybrid cache controller connected to the hybrid cache, from which data and request are moved through layers between processing cores and caches. Each core within the 2D processor layer is communicated through Network-on-Chip (NoC) routers. Also, the latency for traversing each layer is negligible compared to that between two NoC routers. Each layer of the hybrid cache is divided into 32 banks, and several cache banks in each layer which are connected with NoC routers.

4.3 Tag Management for the Large Hybrid Cache

Base on the hybrid cache architecture, we observe that the conventional tag array of the large hybrid cache actually consists of two parts, one is the tag array in the DRAM region and the other is the tag array in the STT-RAM region. Due to the unbalanced read latency of the disparate memory technologies, the latency to access tag array is also unbalanced. Different from conventional design based on NUCA (Das et al. 2015) and NUMA (Li et al. 2013), we consider the unbalanced read latency of the two proposed hybrid cache architectures, which are the hybrid cache with non-volatile STT-RAM and with volatile STT-RAM respectively.

Table 2: System Configurations.

CPU core	32-core OoO, 3.2GHz, 16-core/socket
SRAM Cache	L1: 64KB, 8-way, 4-cycle load-to-use, 64B linesize L2: 4MB, 16-way, 15-cycle hit latency, sequential tag/data access L3: 8MB, 32-way, 64B linesize, LRU, write-back, write-allocate policy
Hybrid LLC	(H1) STT-RAM: 256MB, 29-way, 1.13/21.35 pJ/bit for R/W energy (H2) Volatile STT-RAM: 256MB, 29-way, 1.06/1.18 pJ/bit for R/W energy DRAM: 768MB, 29-way, 1.25/1.31 pJ/bit for R/W energy
Off-Chip DRAM	2 channels, 2 ranks per channel, 8 banks per rank, DDR3-1600 (12.8GB/s)
Network Parameter	9 layers, 32 TSVs, 2-cycle router latency

For the hybrid cache with non-volatile STT-RAM, it is noticed the read latency of STT-RAM is 2x higher than the DRAM. Thus, we propose to move the tag array of STT-RAM region to the DRAM region (Tags-in-DRAM). Similarly, for the hybrid cache with volatile STT-RAM, the volatile STT-RAM is optimized to have lower read latency than the DRAM (over 20%) by shrinking the thickness of MTJ. Thus, we propose to move the tag array of DRAM region to the STT-RAM region (Tags-in-STT-RAM). The reason behind these designs is that the tag access actually is a read operation (to determine cache hit or miss), and we always move the tag array to the region of the hybrid cache with lower read latency to improve overall performance. In the conventional tag management shown in the Figure 5, high latency is wasted in the tag access in the separate hybrid cache regions. Compared to the conventional scheme, Tags-in-DRAM and Tags-in-STT-RAM design can reduce the tag access latency by obviating the need to access the high-latency tag array.

5 EXPERIMENTAL METHODOLOGY

We extend gem5 simulator (Binkert et al. 2011) to simulate a 32-core system with 3-level SRAM cache and a last-level hybrid cache. The STT-RAM region and DRAM region are modeled similar to (Kultursay et al. 2013) and (Loh and Hill 2011) respectively. The major system parameters are listed in the Table 2. All the experimental parameters of 3D volatile/non-volatile STT-RAM and DRAM cache are obtained from the modified version of DESTINY (Poremba et al. 2015). McPAT (Li et al. 2009) is used to get the power values of the hybrid cache. The state-of-the-art DRAM cache (Loh and Hill 2011) is used as our baseline. The 3D TSV model parameter is based on (Sun et al. 2009). The simulations were done for 2 configurations: **(H1)** The hybrid cache with non-volatile STT-RAM; **(H2)** The hybrid cache with volatile STT-RAM.

We analyze a set of large-scale HPC applications from the PARSEC (Bienia et al. 2008) and scientific applications from the SPEC CPU2006 (Henning 2006) to evaluate the energy and performance of the hybrid cache. For each of the workloads, we warmed up the simulation for one billion cycles and collected results for one billion cycles. For the evaluation metrics, we use energy savings and IPC speedups to show how much energy and performance efficiency can be achieved.

PARSEC benchmark focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors. It consists of computationally intensive and parallel programs that are very common in the domain of HPC.

SPEC CPU2006 benchmark is comprised of various scientific and real-life applications, which are used to measure the computer performance stressing on the system's processor and memory subsystem. It evaluates performance by measuring how fast the computer completes a single task.

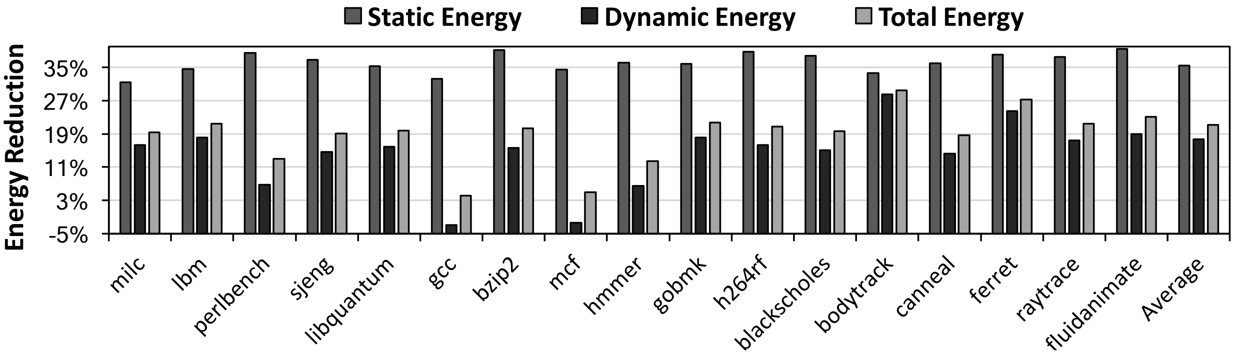


Figure 6: Energy reduction of hybrid cache with non-volatile STT-RAM (H1).

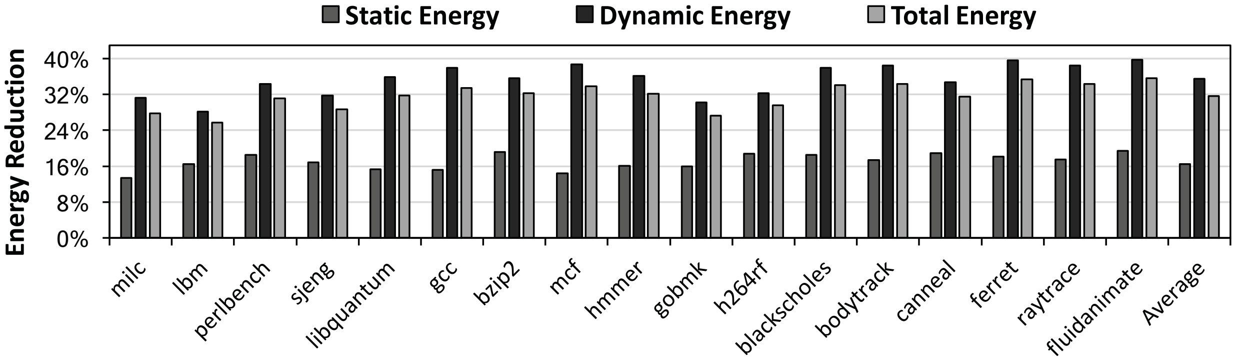


Figure 7: Energy reduction of hybrid cache with volatile STT-RAM (H2).

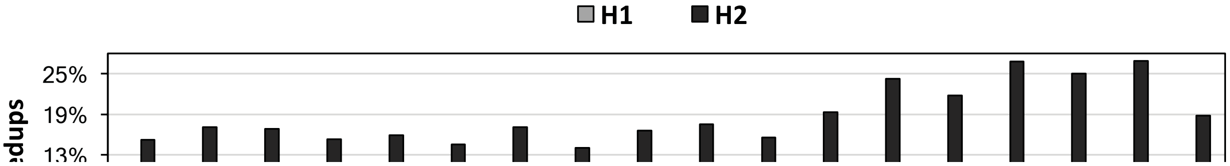
6 RESULTS AND ANALYSIS

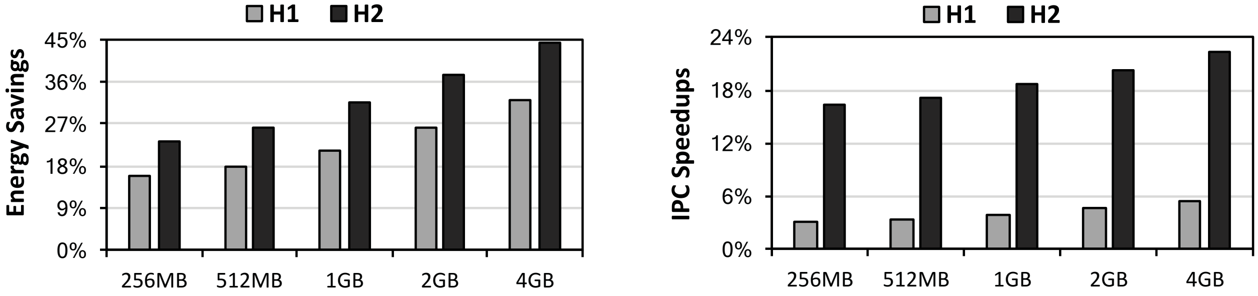
6.1 Energy Analysis

Static Energy: As shown in the Figure 6, the static energy savings of our proposed H1 architecture is about 35.4% on average compared with the baseline DRAM cache. The reason behind the savings is that STT-RAM with small leakage can effectively reduce the static energy. Similarly, the proposed H2 architecture results in 16.4% static energy reduction shown in the Figure 7. This is because STT-RAM is relaxed to reduce write energy by incurring some leakage power increment, but it is still smaller than the leakage power of DRAM.

Dynamic Energy: As presented in the Figure 6, We also observe there is average 17.6% dynamic energy reduction in H1 architecture compared with the baseline. Even for the write-intensive benchmark (e.g. *hmmer* and *perlbench*), there is average 6.5% dynamic energy saving. Because the DRAM region of our hybrid cache is configured large enough to buffer write-intensive request without accessing STT-RAM region, and the STT-RAM eliminates significant refresh energy compared with DRAM. However, there are negative energy savings for the benchmark with large datasets (e.g. mcf and *gcc*), where the input data can be the 0.5x$\sim$1x size of DRAM cache, hence STT-RAM region needs to be frequently accessed incurring high write energy. Considering the result of H2 architecture in the Figure 7, the dynamic energy saving is more balanced in the different benchmarks with 35.5% on average. Because STT-RAM is optimized to have small write energy, and also we select an optimal retention time to balance the refresh energy and write energy.

Total Energy: The total cache energy is reduced by 21.2% and 31.6% in H1 and H2 respectively, compared to the DRAM cache baseline. This energy saving can be attributed to the following reasons: 1) the static





(a)

(b)

Figure 9: Sensitivity analysis of different hybrid cache size.

energy occupies up to 40% of the total cache energy; 2) the volatile STT-RAM with an optimal retention time reduces both the static energy and dynamic energy.

6.2 Performance Analysis

Figure 8 shows the IPC speedups of the proposed hybrid caches compared with the baseline. For H1 architecture, the performance is improved by 4.4% on average. The performance speedup comes from the proposed tag management policy that avoids the STT-RAM tag long-latency access. However, there is some negative improvement for the benchmarks with large input, which results from 1) frequently writing into conventional STT-RAM with long latency; 2) DRAM cache miss when the input datasets are larger than the DRAM cache capacity; 3) 64B random data access in the 4KB memory page caused by the DRAM cache miss.

For H2 architecture, there are 18.8% performance speedups, and we notice the results are more balanced than the H1. This is because STT-RAM is optimized to have comparable latency to the DRAM. Further, the proposed Tags-in-STT-RAM policy replaces DRAM tag access with the low-latency STT-RAM tag. Also, it is noticed that PARSEC benchmarks have more performance improvement than SPEC benchmarks. This is because the parallel characteristics of PARSEC benchmarks can meet more processing demand of CMP systems.

6.3 Sensitivity Analysis

To better understand the effect of hybrid cache capacity on the energy and performance for the HPC systems, we change the capacity to 256MB, 512MB, 1GB, 2GB and 4GB respectively. As Figure 9(a) shows, we observe the total cache energy saving increases with the larger cache capacity. This is because: (1) larger caches have more static energy consumption, which can be effectively removed by the non-volatile STT-RAM; (2) hybrid cache with larger capacity can buffer more data written in the DRAM region, which can reduce write operation in the STT-RAM compared with small cache capacity; (3) dynamic energy is proportional to the size of the cache, and the larger volatile STT-RAM has smaller dynamic energy due to the optimal retention time in our design.

As shown in the Figure 9(b), the performance also increases with larger cache capacity. It can be attributed to the following reasons: 1) the DRAM cache with larger capacity has higher hit rate, which can reduce off-chip memory access; 2) larger capacity increases hit rate in the low-latency region of hybrid cache, e.g. the DRAM region in the H1 architecture and STT-RAM region in the H2 architecture. However, it is noticed that the average tag access time depends on the cache storage size, and larger tag array requires more time for tag access and comparison, which may also impact the performance.

7 CONCLUSIONS

In this paper, we identify that DRAM cache can be used in the future HPC systems to improve performance, but it has the disadvantage of high power consumption. Thus, we present hybrid cache design for the future HPC systems to improve both energy and performance efficiency. First, we observe the DRAM cache with a large capacity has high leakage power, and the large hybrid cache using non-volatile STT-RAM is proposed to reduce static energy. Second, we propose to use volatile STT-RAM as a part of hybrid cache to reduce both dynamic and static energy of the DRAM cache. Finally, we propose tag management policy based on our hybrid cache to improve performance. The results show that energy is reduced by 31.6% and performance is improved by 18.8% on average.

REFERENCES

Bienia, C., S. Kumar, J. P. Singh, and K. Li. 2008. “The PARSEC benchmark suite: characterization and architectural implications”. In *Proceedings of PACT*.

Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, J. Saidi, D. R. Hower, T. Krishna, S. Sardashti et al. 2011. “The gem5 simulator”. *ACM SIGARCH Computer Architecture News*.

Cong, J., G. Gururaj, and Y. Zou. 2011. “An energy-efficient adaptive hybrid cache”. In *Proceedings of ISLPED*.

Das, S., T. M. Aamodt, and W. J. Dally. 2015. “SLIP: reducing wire energy in the memory hierarchy”. In *Proceedings of ISCA*.

Diao, Z., Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai. 2007. “Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory”. *Journal of Physics: Condensed Matter*.

He, J., and J. Callenes-Sloan. 2016. “Reducing the energy of a large hybrid cache”. In *Proceedings of ICECS*.

Henning, J. L. 2006. “SPEC CPU2006 benchmark descriptions”. *ACM SIGARCH Computer Architecture News*.

Huang, C.-C., and V. Nagarajan. 2014. “ATCache: reducing DRAM cache latency via a small SRAM tag cache”. In *Proceedings of PACT*.

Jevdjic, D., G. H. Loh, C. Kaynak, and B. Falsafi. 2014. “Unison cache: A scalable and effective die-stacked DRAM cache”. In *Proceedings of MICRO*.

Jevdjic, D., S. Volos, and B. Falsafi. 2013. “Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache”. *Proceedings of ISCA*.

Kultursay, E., M. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. “Evaluating STT-RAM as an energy-efficient main memory alternative”. In *Proceedings of ISPASS*.

Li, J., C. J. Xue, and Y. Xu. 2011. “STT-RAM based energy-efficiency hybrid cache for CMPs”. In *Proceedings of VLSI-SoC*.

Li, S., J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”. In *Proceedings of MICRO*.

Li, T., Y. Ren, D. Yu, S. Jin, and T. Robertazzi. 2013. “Characterization of input/output bandwidth performance models in NUMA architecture for data intensive applications”. In *Proceedings of ICPP*.

Liu, J., B. Jaiyen, R. Veras, and O. Mutlu. 2012. “RAIDR: Retention-aware intelligent DRAM refresh”. In *Proceedings of ISCA*.

Loh, G. H., and M. D. Hill. 2011. “Efficiently enabling conventional block sizes for very large die-stacked DRAM caches”. In *Proceedings of MICRO*.

Poremba, M., S. Mittal, D. Li, J. S. Vetter, and Y. Xie. 2015. “DESTINY: A Tool for Modeling Emerging 3D NVM and eDRAM caches”. In *Proceedings of DATE*.

Smullen, C. W., V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. 2011. “Relaxing non-volatility for fast and energy-efficient STT-RAM caches”. In *Proceedings of HPCA*, pp. 50–61.

Sun, G., X. Dong, Y. Xie, J. Li, and Y. Chen. 2009. “A novel architecture of the 3D stacked MRAM L2 cache for CMPs”. In *Proceedings of HPCA*.

Wang, Z., D. A. Jiménez, C. Xu, G. Sun, and Y. Xie. 2014. “Adaptive placement and migration policy for an STT-RAM-based hybrid cache”. In *Proceedings of HPCA*.

Wu, X., J. Li, L. Zhang, R. Speight, and Y. Xie. 2009. “Hybrid cache architecture with disparate memory technologies”. In *Proceedings of ISCA*.

Zhao, L., R. Iyer, R. Illikkal, and D. Newell. 2007. “Exploring DRAM cache architectures for CMP server platforms”. In *Proceedings of ICCD*.

AUTHOR BIOGRAPHIES

JIACONG HE received M.S. degree in Computer Engineering from the Illinois Institute of Technology. He currently is a Ph.D. student in the Electrical Engineering Department at the University of Texas at Dallas. His research interests include memory systems, high performance computing and dependable systems. His email address is jiacong.he@utdallas.edu.

JOSEPH CALLENES-SLOAN is an Assistant Professor in the Erik Jonsson School of Engineering & Computer Science at the University of Texas at Dallas. He holds a Ph.D. in Electrical and Computer Engineering at the University of Illinois in Urbana-Champaign. His research interests include computer architecture, scientific and high performance computing, low power design and fault tolerance. His email address is jcallenes.sloan@utdallas.edu.

ADAPTIVE PARTICLE ROUTING IN PARALLEL/DISTRIBUTED PARTICLE FILTERS

Xudong Zhang
Department of Computer Science
Graduate Center of The City University of New York
365 5th Avenue
New York, NY 10016
xzhang5@gradcenter.cuny.edu

Lixin Huang
Department of Electrical Engineering and Computer Science
Syracuse University
4-206 Center for Science and Technology
Syracuse, NY 13244
lhuang24@syr.edu

Evan Ferguson-Hull
Department of Mathematics
Bates College
2 Andrews Road
Lewiston, ME 04240
eferguso@bates.edu

Feng Gu
Department of Computer Science
College of Staten Island
2800 Victory Boulevard
Staten Island, NY 10314
Feng.Gu@csi.cuny.edu

ABSTRACT

Particle filters estimate the state of dynamic systems through Bayesian interference and stochastic sampling techniques. Parallel/distributed particle filters aim to improve the performance by deploying all particles on different processing units. However, the communication cost of transferring particles is high due to the centralized processing in resampling step. To reduce the communication cost without loss of accuracy, the hybrid particle routing policy is designed for the resampling step, which mainly executes particles resampling and exchanges locally and routes them globally every specific number of calculation steps. However, the global particle routing is more necessary when the convergence of particles is low. In this paper, we propose the adaptive particle routing algorithm, in which the local resampling and particle exchange are used, and the planned global particle routing is adopted only when the measured convergence is below the set threshold. The experimental results show the improved performance.

Keywords: particle filters, parallel/distributed computing, adaptive particle routing.

1 INTRODUCTION

Particle filters, also called sequential Monte Carlo (SMC) methods, provide a numerical approximation to the nonlinear filtering problem. Particle filters use Bayesian inference and stochastic sampling techniques to recursively estimate the states of dynamic systems from some given observations (Smith, Schmidt, and McGee 1962; Kailath, Sayed, and Hassibi 2000; Gu 2010; Gustafsson 2010; Helmke and Moore 2012) with little or without assumptions of the system model's properties. Therefore, particle filters have been used in many non-linear and/or non-Gaussian applications, such as positioning, navigation, visual tracking, and wildfire spread systems (Freeman 1987; Ikeda and Matsumoto 1987; Kocarev and Parlitz 1995; Gustafsson et al. 2002; van Leeuwen 2003). In the applications of particle filters, sequential

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

importance sampling and resampling (SISR) is one of the widely used particle filtering algorithms. The SISR algorithm has two main stages, sampling and resampling. In the sampling stage, a set of particles representing the belief of the system is used to generate a new set of particles to represent the system model. These new particles represent the posterior belief according to the prior distribution. An observation measures the particles by calculating and normalizing the weights of all the particles. In the resampling stage, offspring particles are obtained according to the normalized weights. At each time step, sampling and resampling are executed and the resampled particles will be the input of the sampling of next time step. This procedure continues until the observation is unavailable.

One of the challenges to apply particle filters is the performance due to the used large number of particles, especially for large-scale dynamic systems. To improve the performance, parallel/distributed particle filters are introduced (Bolic, Djuric, and Hong 2005; Sheng et al. 2005; Bai et al. 2016). There are no communications between processing units in the sampling stage. Therefore, the main difference for these algorithms lies in how to route the resampled particles to other processing units in the resampling stage due to its centralized processing. Different particle routing policies define how the processing units with extra particles send particles to those with shortage of particles to achieve the load balance. Although efficient particle routing policies can achieve speedups to some extent, they still suffer from high communication costs. To further enhance the performance, decentralized resampling algorithms are designed (Bolic, Djuric, and Hong 2005), in which the global resampling is removed and only a small percentage of particles are exchanged between processing units after the local resampling on each processing unit. However, it may decrease the accuracy of state estimation due to lack of the global resampling.

To improve the performance without loss of accuracy, a hybrid particle routing policy is adopted (Bai et al. 2016). The hybrid routing policy is mainly based on the decentralized resampling and invokes the centralized resampling every a certain number of calculation steps. Therefore, it combines both the decentralized resampling and the centralized resampling to achieve better speedups and accuracy of the estimated states. However, in many scenarios, the particles are well converged, therefore, the centralized resampling (scheduled at every k time steps) may not be needed. To further improve the performance, we propose the adaptive particle routing policy, in which the decentralized resampling is adopted and the centralized resampling every a certain number of steps is invoked only when the convergence of particles is low. We measure the convergence of particles to decide if the centralized resampling is needed at those scheduled centralized resampling steps. It avoids unnecessary centralized particle routing steps and reduces their incurred extra communication costs.

The rest of the paper is organized as follows. Section 2 introduces the related work in particle filters and parallel/distributed particle filters. Section 3 presents the hybrid particle routing policy in parallel/distributed particle filters. Section 4 describes the proposed adaptive particle routing policy and its algorithm in parallel/distributed particle filters. Section 5 provides the experiments and achieved results. Section 6 concludes the paper and points out the future work.

2 RELATED WORK

The applications of particle filters can be found in a variety of domains, including epidemic predictions, geophysical systems, geosciences and remote sensing, transportation systems, and wildfire spread simulations. Dawson, Gailis, and Meehan (2015) consistently analyzed the probability that a disease happened in a population based on the medical records of the individual of the target popular using particle filters. The results showed the improvement of detection times for outbreaks in populations with electronic medical records available. Mattern, Dowd, and Fennel (2013) assimilated satellite observations of surface chlorophyll into a 3-D biological ocean model to improve its state estimation using particle filters. They tested the feasibility of biological state estimation with particle filters for realistic models. Yan, DeChant, and Moradkhani (2015) estimated soil moisture and soil hydraulic parameters using particle filters. The proposed approach corrected the soil moisture state and estimated the soil hydraulic parameters. Yan, Gu, and Hu (2013) applied particle filters to reconstruct the event like a traffic jam by the collected information of deployed cameras. They detected the slow moving vehicle in the road network to cause the traffic jam. Xue, Gu, and Hu (2012) assimilated temperature data from deployed fire sensors into a wildfire spread simulation model to estimate the fire fronts and the related experimental results verified the improved state estimation.

In many other applications, the parallel/distributed particle filters are adopted to address the performance issue. Ing and Coates (2005) implemented a distributed particle filters algorithm for object tracking in wireless sensor networks. The designed scheme significantly reduced the energy cost of communication. Hong et al. (2006) designed and implemented a flexible resampling mechanism for parallel particle filters in a CMOS process, and then analyzed its complexity and performance. Sutharsan et al. (2012) presented an optimization-based scheduling algorithm for parallel implementation of particle filters and evaluated the effectiveness of the proposed algorithm by the application of multi-target tracking. Hegyi et al. (2007) described two different parallel particle filter algorithms for the state estimation of freeway traffic network. Their accuracy, performance, and communication costs are analyzed and compared. Rosencrantz et al. (2002) developed a decentralized parallel particle filters algorithm to exchange information between nearby platforms in robotic systems. They illustrated the scaling capability to a large team of vehicles. Liu et al. (2009) used parallel particle filters algorithm in face tracking and it worked robustly for cluttered backgrounds and different illuminations. The multi-core parallel computing achieved a good linear speedup compared to its sequential implementation.

From the above applications, there are two main categories of resampling algorithms in parallel/distributed particle filters algorithms, including the centralized resampling algorithm and the decentralized resampling algorithm. Teuilere and Brun (2003) used a centralized approach to parallelize the resampling step and applied it to Doppler-hearing tracking of maneuvering sources, in which a central unit collected the weights from each processing unit, did the resampling, and returned replication factors to each processing unit. Bolic et al. (2005) proposed the decentralized resampling strategies and implemented four versions of parallel/distributed particle filters algorithms. They removed the centralized resampling and utilized the local weight information to decide the exchange of particles between processing units. The centralized resampling and the distributed resampling have their own disadvantages, either achieving low speedups or losing accuracy. Bai et al. (2016) systemically analyzed various centralized resampling and decentralized resampling routing policies and proposed a novel approach to combine both to achieve better speedups without loss of accuracy. This proposed hybrid particle routing policy was based on the decentralized resampling schema and invoked the centralized resampling every k time steps. It was examined by an application of large-scale spatial temporal system, wildfire spread simulation, and exhibited its effectiveness. However, in some cases, the particles distribution is "good" and the scheduled centralized resampling is not needed. Therefore, it should be called as needed for those time steps to reduce the communication cost. Based on this idea, we develop the adaptive particle routing policy and provide details in the following sections.

3 ADAPTIVE PARTICLE ROUTING ALGORITHM

3.1 Parallel/Distributed Particle Filters

There are three main steps in the general particle filters algorithm (SISR algorithm), including sampling, weight computation, and resampling step. Since resampling needs the global information of all particles, it is the main obstacle to parallelize particle filters algorithms. In general, two primary categories of resampling in parallel/distributed particle filters are developed, including centralized resampling and decentralized resampling. In the centralized resampling, there are two types of nodes, the central unit and the processing unit. Sampling and weight computation are independently executed on each processing unit due to no data dependency, and resampling is conducted on the central unit because of the demand of global information. The central unit collects the weights of all particles from all the processing units, performs particle resampling, and transfers the particles between the central unit and the processing units according to different particle routing policies. Figure 1 shows the procedure of the centralized resampling. In the figure, fours processing units (PU1, PU2, PU3, and PU4) send their weights to the central unit (CU), and CU serves as the hub for four processing units to exchange particles after resampling. Different centralized resampling routing policies and their corresponding analysis could be found in (Bai et al. 2016). Decentralized resampling removes the central unit to reduce the communication cost. Sampling, weight computation, and resampling are executed on each processing unit separately. To make "good" particles propagate to other processing units, a specific percentage of particles on each processing unit are sent to its neighboring processing unit at each time step. Figure 2 displays the decentralized resampling schema. In the figure, four processing units (PU1, PU2, PU3, and PU4) perform independent particle filters steps and forward some number of particles to their neighboring processing units in the clockwise order. More decentralized particle routing policies were discussed in (Bolic et al. 2005).

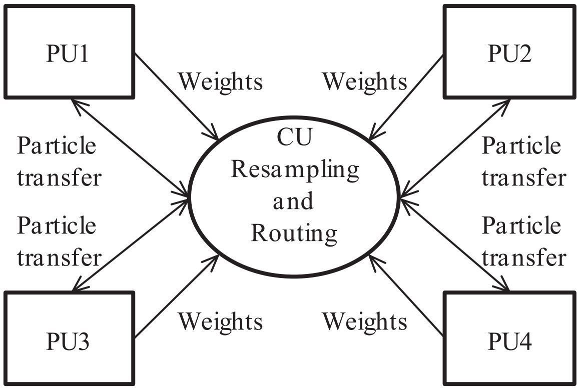


Figure 1: Centralized resampling.

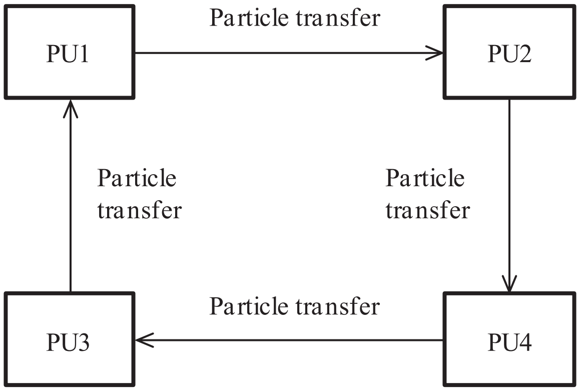


Figure 2: Decentralized resampling.

The centralized resampling schema precisely implements the particle filters algorithm, but suffers from the scalability due to the central unit. The distributed resampling schema improves the scalability, but may need a large number of iterations for fully resampling because of its local nature and limited particle exchanges between processing units. A hybrid routing policy (Bai et al. 2016) was proposed, in which the decentralized resampling was mainly adopted to achieve a large degree parallelism. Processing units performed local resampling and exchanged particles between neighboring processing units. To overcome the limitation of local particle exchanges in the decentralized resampling, the centralized resampling was occasionally invoked to utilize the full knowledge of weights of all particles. It helped quickly and efficiently route "good" particles to all the processing units. This hybrid particle routing policy has been applied in large-scale spatial temporal systems, such as wildfire spread simulation. Through the simulation results, the hybrid particle routing policy greatly improved the performance of the data assimilation of wildfire spread simulation without loss of the state estimation accuracy. More details can be referred to the work in (Bai et al. 2016).

3.2 Adaptive Particle Routing Algorithm

In the adaptive particle routing, both of the decentralized resampling and the centralized resampling are used to achieve both of performance and accuracy of particle filters algorithms. The centralized resampling is invoked every a certain number of steps in the hybrid particle routing policy. However, in many those steps, the particles have good convergence. Therefore, the centralized resampling is not necessary. To more efficiently utilize the centralized resampling, we need to evaluate the convergence of particles to decide its necessity. Towards this objective, we propose the adaptive particle routing algorithm to adaptively invoke the scheduled centralized resampling every k steps when needed. To measure the convergence, we adopt the effective sample size $\widehat{N}_{eff}$, as defined in Equation (1).

$$\widehat{N}_{eff} = \frac{1}{\sum_{i=1}^{N}(\tilde{q}_t{}^{(i)})^2}\,, \qquad (1)$$

where $\tilde{q}_t{}^{(i)}$ is the normalized weight of particle i at time step t, and N is the number of particles. A threshold is used and the centralized resampling scheduled at every specific number of times steps is invoked if the effective sample size of particles is smaller than the predefined threshold. Table 1 lists the adaptive particle routing algorithm.

Table 1: Adaptive Particle Routing Algorithm.

```
Processing unit side:
  for all the parallel processing units at time step $t$
    1. Run the sampling step.
    2. Calculate the importance weight of each particle.
    3. Send all weights to the central unit.
    4. Receive information from the central unit. If the centralized resampling needs to be performed,
       go to step 5, otherwise go to step 9.
    5. Receive routing information from the central unit.
    6. If having surplus of particles, send the selected particles (based on the received routing
       information from the central unit) to the central unit.
    7. If having shortage of particles, receive particles from the central unit.
    8. End.
    9. Normalize, perform resampling locally, and send partial particles to the neighboring processing
       units in the clockwise order .
    10. End.
Central unit side:
  At every $k$ time step
    1. Predefine the threshold $TD$.
    2. Receive the weight of each particle from all processing units.
    3. Calculate the normalized importance weights of all particles and the effective particle size $\widehat{N}_{eff}$
           $\widehat{N}_{eff} = \frac{1}{\sum_{i=1}^{N}\left(\tilde{q}_t{}^{(i)}\right)^2}$
    4. If $\widehat{N}_{eff} < TD$, go to step 5 (activate the centralized resampling), otherwise skip the following and
       inform all the processing units whether global resampling is needed.
    5. Exert the centralized resampling and compute routing information.
    6. Send the routing information to processing units.
    7. Receive particles from processing units that have surplus of particles.
    8. Send particles according to the routing information to the processing units that have shortage of
       particles.
    9. End.
```

4 EXPERIMENTS AND RESULTS

In order to evaluate the performance of the adaptive particle routing algorithm, we implemented the sequential particle filters algorithm, the parallel particle filters algorithm with the hybrid particle routing policy and the parallel particle filters algorithm with the adaptive particle routing policy. The SISR algorithm was applied to the following system with the system equation in Equation (2) and the measurement equation in Equation (3). In the equations, x_{t+1} and x_t are the system state at time step t+1 and time step t respectively; y_t is the measurement variable at time step t; ω_t and e_t are the system noise and measurement noise at time step t. In the above system, the associated configurations are: $x_0{\sim}N(0,5)$, $\omega_t{\sim}N(0,10)$, and $e_t{\sim}N(0,1)$. This system has been analyzed in many particle filters publications (Gordon et al. 1993; Kitagawa 1996; Doucet 1998; Arulampalam et al. 2002). We will also use this system to evaluate our proposed adaptive particle routing policy in parallel particle filters algorithms. We compare the accuracy among the sequential implementation, the parallel implementation with the hybrid particle routing policy, and the parallel implementation with the adaptive particle routing policy, and also compare the performance of these two parallel implementations. We run the all the experiments with 15,000 time steps using 10,000 particles. We present the results below.

$$x_{t+1} = \frac{x_t}{2} + \frac{25x_t}{1+x_t^2} + 8\,cos(1.2t) + \omega_t \tag{2}$$

$$y_t = \frac{x_t^2}{20} + e_t \tag{3}$$

Figure 3, Figure 4, and Figure 5 show the plots for the sequential particle filters implementation, the parallel particle filters implementation with the hybrid particle routing policy, and the parallel particle filters implementation with the adaptive particle routing policy respectively. In the figures, the horizontal axis and the vertical axis refer to the time step and the state respectively, and the blue line and the red line represent the true states and the estimated states respectively. From the figures we know that the estimated states are close to the true states by applying the observations into the system model for all the three implementations. To compare the accuracy, we calculate the time-averaged root mean square error (RMSE) as defined in Equation (4) for the three cases, where *R* is the calculated time-averaged RMSE, $\hat{x}_t$is the estimated state at time step *t*, x_t is the true state at time step *t*, and *T* is the total number of time steps. The calculated time averaged RMSEs for the sequential particle filters implementation, the parallel particle filters implementation with the hybrid particle routing policy, and the parallel particle filters implementation with the adaptive particle routing policy are 0.103,00, 0.091,98, and 0.092,00 respectively. They are small and very close, which further indicates all of the three implementations are able to estimate the system states with high accuracy and their estimated accuracies are similar.

$$R = \frac{1}{T}\sqrt{\sum_{t=1}^{T}(\hat{x}_t - x_t)^2} \tag{4}$$

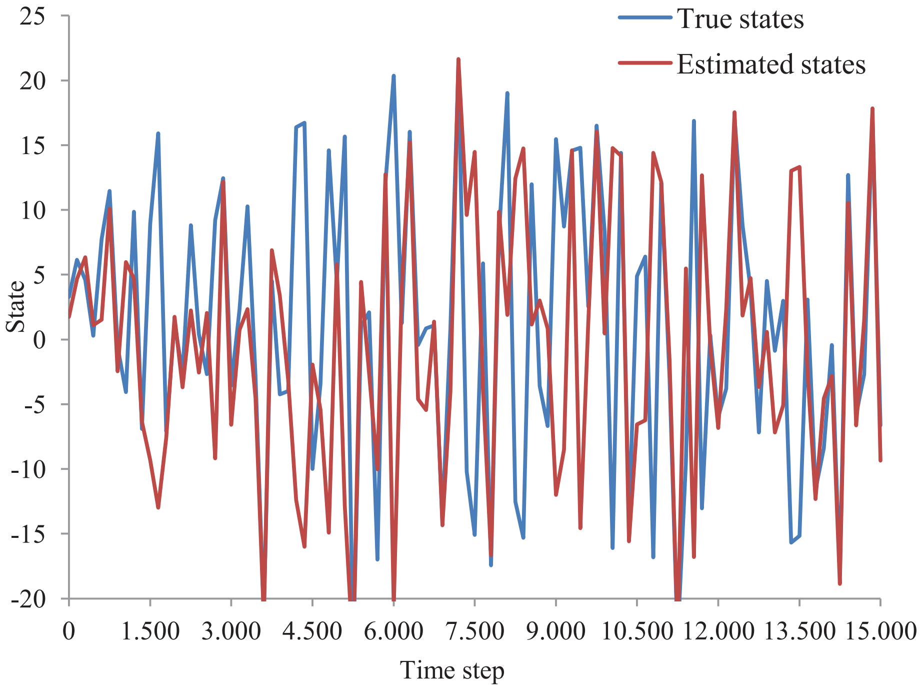


Figure 3: Sequential particle filters.

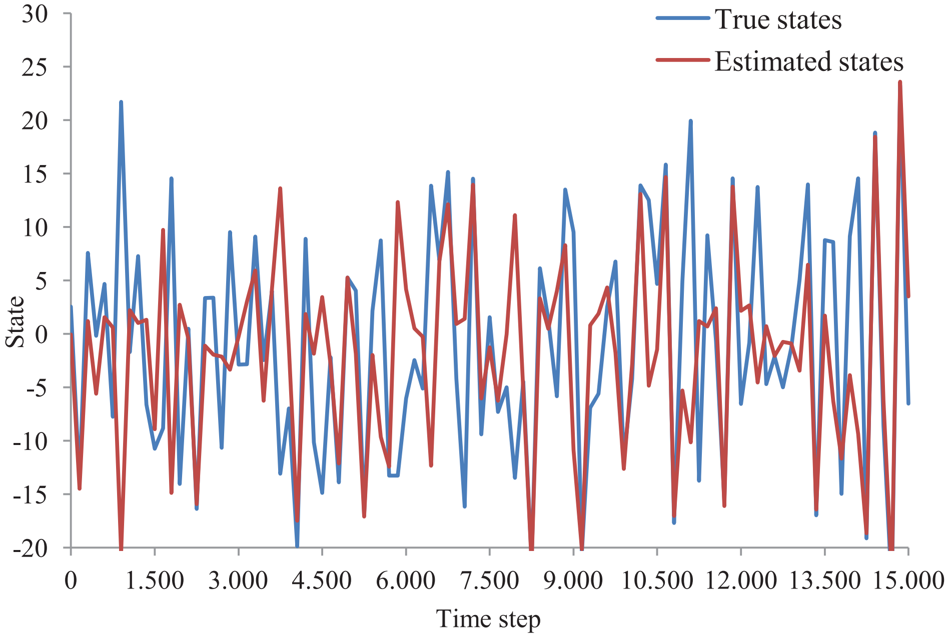


Figure 4: Parallel/distributed particle filters using hybrid particle routing policy.

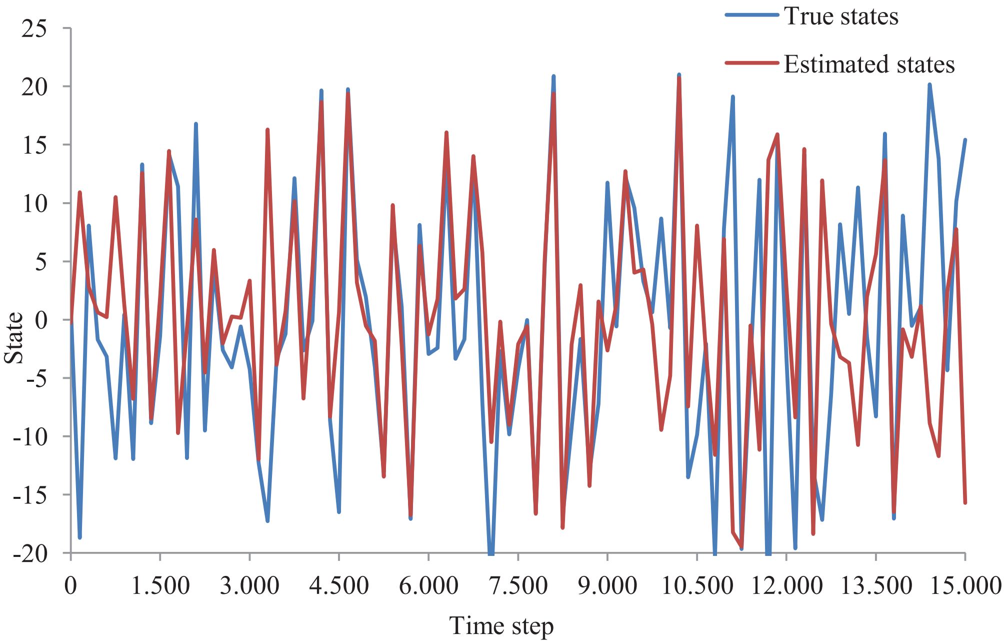


Figure 5: Parallel/distributed particle filters using adaptive particle routing policy.

We also compare the performance of the two parallel particle filters implementations using the hybrid particle routing policy and the adaptive particle routing policy. Firstly, we calculate the number of transferred particles for both of the algorithms during the execution. Figure 6 display the numbers of transferred particles for the parallel particle filters implementations with both of the particle routing policies. In Figure 6, the horizontal axis and the vertical axis represent the time step and the number of transferred particles (in thousand) respectively, and the green line and blue lines represent the numbers of transferred particles for the parallel implementations with the hybrid particle routing policy and the adaptive particle routing policy respectively. It indicates that the number of transferred particles for the parallel particle filters with the adaptive particle routing policy is smaller than that for the parallel particle filters with the hybrid particle routing policy, because the former policy avoids the unnecessary global resampling. Therefore, the communication cost of the former is less than that of the latter. The time consumptions for the parallel particle filters with the adaptive particle routing policy and the parallel particle filters with the hybrid particle routing policy are 56.9 seconds and 68.1 seconds respectively, which is consistent with the results of the number of transferred particles in Figure 6.

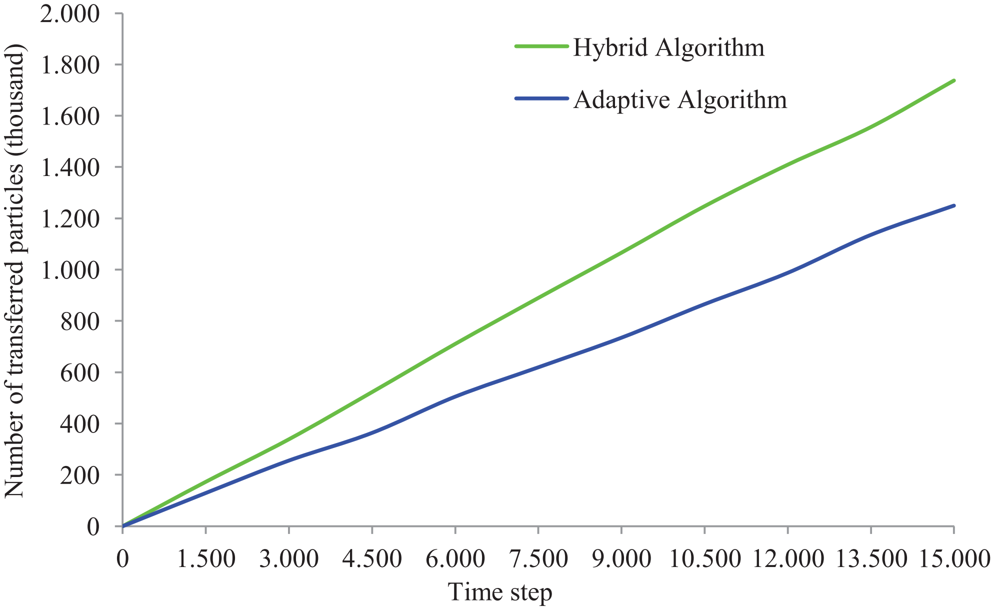


Figure 6: Number of transferred particles for parallel particle filters with the hybrid particle routing policy and the adaptive particle routing policy.

5 CONCLUSIONS AND FUTURE WORK

Parallel/distributed particle filters are able to improve the performance by deploying the particles on multiple processing units. However, the high communication costs among multiple processing units for particles transfer in the resampling step decrease the entire performance. Although the decentralized particle routing policy can address this issue, the accuracy may be affected due to the local resampling on processing units and limited particle exchanges between processing units. The hybrid particle routing policy is based on the decentralized resampling schema and occasionally invokes the centralized resampling to achieve the speedups with the similar accuracy. The adaptive particle routing policy is able to avoid the unnecessary centralized resampling steps by measuring the convergence of particles in order to further improve the performance. The designed experiments show the parallel particle filters with the adaptive particle routing policy achieves better speedups without loss of accuracy. This will have an important impact on performance improvement for parallel particle filters applications, especially those large-scale dynamic systems due to their high dimensions and large system states. Our future work will focus on the following directions. Firstly, we will systematically analyze the theoretical communication and computation cost for different particle routing policies and measure their performances using the defined metrics. Secondly, we will apply the proposed adaptive particle routing algorithm in the large-scale spatial temporal systems to achieve better performance.

ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation under Grant 1356977 and 1359266.

REFERENCES

Arulampalam, M. S., S. Maskell, N. Gordon and T. Clapp. 2002. "A Tutorial on Particle Filters for Online Nonlinear/non-Gaussian Bayesian Tracking". *IEEE Transactions on Signal Processing* vol. 50, pp. 174-188.

Bai, F., F. Gu, X. Hu, and S. Guo. 2016. "Particle Routing in Distributed Particle Filters for Large-Scale Spatial Temporal Systems". *IEEE Transactions on Parallel and Distributed Systems* vol. 27, pp. 481-493.

Bolic, M., P. M. Djuric, and S. Hong. 2005. "Resampling Algorithms and Architectures for Distributed Particle Filters". *IEEE Transactions on Signal Processing* vol. 53, pp. 2442-2450.

Dawson, P., R. Gailis, and A. Meehan. 2015. "Detecting Disease Outbreaks Using a Combined Bayesian Network and Particle Filter Approach". *Journal of Theoretical Biology* vol. 370, pp. 171-183.

Doucet, A. 1998. "On Sequential Simulation-based Methods for Bayesian Filtering". Technical Report CUED/F-INFENG/TR.310, Signal Processing Group, Department of Engineering, University of Cambridge, UK

Freeman, W. J. 1987. "Simulation of Chaotic EEG Patterns with a Dynamic Model of the Olfactory System". *Biological Cybernetics* vol. 56, pp. 139-150.

Gordon, N. J., D. J. Salmond, and A. F. Smith. 1993. "Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation". In *IEE Proceedings F-Radar and Signal Processing* vol. 140, pp. 107-113.

Gu, F. 2010. *Dynamic Data Driven Application System for Wildfire Spread Simulation*. Ph.D. thesis, Department of Computer Science, Georgia State University, Atlanta, Georgia. Available via http://scholarworks.gsu.edu/cs_diss/57. Accessed Dec. 14, 2010.

Gustafsson, F. 2010. "Particle Filter Theory and Practice with Positioning Applications". *IEEE Aerospace and Electronic Systems Magazine* vol. 25, pp. 53-82.

Gustafsson, F., F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P. J. Nordlund, 2002. "Particle Filters for Positioning, Navigation, and Tracking". *IEEE Transactions on Signal Processing* vol. 50, pp. 425-437.

Hegyi, A., L. Mihaylova, R. Boel, and Z. Lendek. 2007. "Parallelized Particle Filtering for Freeway Traffic State Tracking". In *Control Conference (ECC), 2007 European* pp. 2442-2449.

Helmke, U., and J. B. Moore. 2012. *Optimization and Dynamical Systems*. Springer Science & Business Media.

Hong, S., S. S. Chin, P. M. Djurić, and M. Bolić. 2006. "Design and Implementation of Flexible Resampling Mechanism for High-speed Parallel Particle Filters". *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* vol. 44, pp. 47-62.

Ikeda, K., and K. Matsumoto. 1987. "High-dimensional Chaotic Behavior in Systems with Time-delayed Feedback". *Physica D: Nonlinear Phenomena* vol. 29, pp. 223-235.

Ing, G., and M. J. Coates. 2005. "Parallel Particle Filters for Tracking in Wireless Sensor Networks". In *IEEE 6th Workshop on Signal Processing Advances in Wireless Communications,* pp. 935-939.

Kailath, T., A. H. Sayed, and B. Hassibi. 2000. *Linear Estimation*. Upper Saddle River, New Jersey, Prentice Hall, Inc.

Kitagawa, G. 1996. "Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models". *Journal of Computational and Graphical Statistics* vol. 5, pp. 1-25.

Kocarev, L., and U. Parlitz. 1995. "General Approach for Chaotic Synchronization with Applications to Communication". *Physical Review Letters* vol. 74, p. 5028-5031.

Liu, K. Y., S. Q. Li, L. Tang, L. Wang, and W. Liu. 2009. "Fast Face Tracking Using Parallel Particle Filter Algorithm". In *2009 IEEE International Conference on Multimedia and Expo*, pp. 1302-1305.

Mattern, J. P., M. Dowd, and K. Fennel. 2013. "Particle Filter‐Based Data Assimilation for a Three‐Dimensional Biological Ocean Model and Satellite Observations". *Journal of Geophysical Research: Oceans* vol. 118, pp. 2746-2760.

Rosencrantz, M., G. Gordon, and S. Thrun. 2002. August. "Decentralized Sensor Fusion with Distributed Particle Filters". In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence,* pp. 493-500. Morgan Kaufmann Publishers Inc..

Sheng, X., Y. H. Hu, and P. Ramanathan. 2005. "Distributed Particle Filter with GMM Approximation for Multiple Targets Localization and Tracking in Wireless Sensor Network". In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. IEEE Press.

Smith, G. L., S. F. Schmidt, and L. A. McGee. 1962. *Application of Statistical Filter Theory to the Optimal Estimation of Position and Velocity on Board a Circumlunar Vehicle*. National Aeronautics and Space Administration.

Sutharsan, S., T. Kirubarajan, T. Lang, and M. McDonald. 2012. "An Optimization-based Parallel Particle Filter for Multitarget Tracking". *IEEE Transactions on Aerospace and Electronic Systems* vol. 48, pp. 1601-1618.

Teulière, V., and O. Brun. 2003. "Parallelisation of the Particle Filtering Technique and Application to Doppler-bearing Tracking of Maneuvering Sources". *Parallel Computing* vol. 29, pp. 1069-1090.

van Leeuwen, P. J. 2003. "A Variance-minimizing Filter for Large-scale Applications". *Monthly Weather Review* vol. 131, pp. 2071-2084.

Xue, H., F. Gu, and X. Hu. 2012. "Data Assimilation Using Sequential Monte Carlo Methods in Wildfire Spread Simulation". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* vol. 22, p.23-26.

Yan, H., C. M. DeChant, and H. Moradkhani. 2015. "Improving Soil Moisture Profile Prediction with the Particle Filter-Markov Chain Monte Carlo Method". *IEEE Transactions on Geoscience and Remote Sensing* vol. 53, pp. 6134-6147.

Yan, X., F. Gu, X. Hu, and C. Engstrom. 2013. "Dynamic Data Driven Event Reconstruction for Traffic Simulation Using Sequential Monte Carlo Methods". In *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, pp. 2042-2053. IEEE Press.

AUTHOR BIOGRAPHIES

XUDONG ZHANG is a PhD student in the Department of Computer Science at the Graduate Center of The City University of New York. His research interests include modeling and simulation, and high performance computing. His email address is xzhang5@gradcenter.cuny.edu.

LIXIN HUANG is an undergraduate student majoring in computer engineering at Syracuse University. Her research interests include high performance computing, social computing and digital humanities. Her email address is lhuang24@syr.edu.

EVAN FERGUSON-HULL is an undergraduate student majoring in mathematics at Bates College. His email is eferguso@bates.edu.

FENG GU is an Assistant Professor in the Department of Computer Science at College of Staten Island, The City University of New York. He holds a Ph.D. in Computer Science from Georgia State University. His research interests include modeling and simulation, high performance computing, and bioinformatics. His email address is Feng.Gu@csi.cuny.edu.

IMPROVING THE PERFORMANCE OF OPTIMISTIC TIME MANAGEMENT MECHANISM WITH SUB-STATE SAVING

B. Kaan Görür
Department of Computer Engineering,
Hacettepe University
Ankara, Turkey
bkaangorur@gmail.com

Kayhan İmre
Department of Computer Engineering,
Hacettepe University
Ankara, Turkey
ki@hacettepe.edu.tr

Halit Oğuztüzün
Department of Computer Engineering,
Middle East Technical University
Ankara, Turkey
oguztuzn@ceng.metu.edu.tr

Levent Yilmaz
Department of Computer Science and Software
Engineering, Auburn University
Auburn, Alabama 36849, USA
yilmaz@auburn.edu

ABSTRACT

Optimistic approaches are scalable methods for time management in parallel and distributed simulations. In optimistic time management, logical processes advance their local time without constrained by the others in the simulation. If a logical process receives a message from its past, it goes back to a previously saved state, which is called a rollback. Even though the received message from the past would not cause any problem, rollback is performed in any case. In this paper, we presented a method to reduce the number of rollbacks, without sacrificing the accuracy of simulation. We propose to save a relatively small subset of the full simulation state to allow the logical processes to make a decision whether a rollback is really needed or not. Our technique is demonstrated in an agent-based simulation using the Time Warp algorithm adapted for optimistic time management for Repast HPC.

Keywords: parallel and distributed simulation, optimistic time management, Time Warp algorithm, state saving.

1 INTRODUCTION

Parallel and distributed processing techniques are used extensively to simulate models that require high computational power. In the last three decades, a lot of algorithms have been proposed to overcome performance issues in parallel and distributed simulations (PADS), for example Jefferson (1985), Leong, Agrawal, Agre (1993), Fleischmann and Wilsey (1995) and Collier and North (2012). Many studies show that improvements in the time management mechanism can bring about remarkable performance gains, see, for example Jafer and Wainer (2010) and Barnes et al. (2013). Two main approaches are popular in PADS community: conservative and optimistic time management. Conservative time management makes sure to keep logical processes (LP) safe against timing errors. LPs in conservative approach should advance their local time together, because they have to be in a correctly computed state. Therefore, once their local time is advanced, LPs are guaranteed not to receive any message from their past. Although the tight synchronization mechanism makes conservative time management more cautious, it causes the simulation

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

to slow down in some cases. Therefore, several algorithms that loosen the synchronization have been offered. Optimistic time management is a strategy that does not strive for tight synchronization as much as conservative time management. Optimistic LPs are prone to receive messages from their past, known as straggler messages. Ignoring straggler messages may cause the simulation results to be incorrect, so LPs have to perform a rollback and take care of repairing their state when a straggler message is received. The main advantage of optimistic approaches is avoiding the overhead of synchronization operations. On the other hand, possibility of frequent rollbacks can be evaluated as the weakest point of optimistic approaches, because the cost of a rollback is considerable. Therefore, the performance of optimistic algorithms that have not been designed with special care to minimize the cost of rollbacks or that do not fit the application at hand may perform worse than conservative ones (Fujimoto 1999).

In optimistic algorithms, one of the two basic rollback methods can be employed to manage straggler messages. The first one is the incremental state saving method that makes LPs save state transitions. In this method, the LP that receives a straggler message makes reverse computations of the saved operations. The second method is copy-state saving that makes LPs to save their exact state. When a straggler message is received, the LP goes back to the saved state. In copy-state saving method, one of the difficulties is deciding on when checkpoints are taken. Taking a checkpoint at each time step is usually prohibitive in terms of both memory/disk and processors, although it provides LPs to directly restore their state to a past state. We will refer to this method as perpetual checkpointing, in the rest of this paper. To overcome this overhead, checkpoints are taken only at specified time ticks, known as periodic checkpointing. In this case, the LP restores the simulation state to the last checkpoint when a straggler message is received. Then, the events between the checkpoint and straggler's time are re-executed. This step is called coasting-forward. The coasting-forward step is the same as the normal execution of the simulation except that the send/receive operations are not repeated. After coasting forward is completed, the straggler message is processed and rest of the events are executed in order. Many studies showed that periodic checkpointing techniques can provide remarkable performance improvements (Lin et al. 1993; Fleischmann and Wilsey 1995). However LPs have to go back to the checkpoint that has been taken much earlier than time of the straggler message, in some cases. This means that coasting-forward step can be very expensive (Fujimoto 1999). In this paper, we are interested in only copy-state saving methods.

Both periodic and perpetual checkpointing have some pros and cons. We aim to leverage the advantages of those two approaches. In this paper, we propose a sub-state saving method to improve the performance of optimistic approach by reducing the number of rollbacks. The main contribution of this study is saving partial state of LPs to decide whether a rollback is really needed or not. As far as we know, no study in the literature uses state saving for this purpose. Instead, the other studies save states to create a checkpoint in case a straggler message is received in the future and LPs have to be restored (Fujimoto 1999; Fleischmann and Wilsey 1995). We also present an implementation strategy for this method on spatial agent-based models. The proposed method has been compared with conventional Time Warp algorithm for an agent-based model in a high performance computing (HPC) system. We experimented the proposed method in Repast HPC with Time Warp (RHPC_TW) (Görür et al. 2016) that is the earlier part of the present study.

2 RELATED WORK

Time Warp algorithm was proposed by Jefferson (1985) and became the most well-known optimistic time management algorithm. Performance evaluation studies for the Time Warp algorithm can be found in the literature. For example, Presley, Reiher and Bellenot (1990) observed a speedup of 29.5% for Sharks World with Time Warp. Perumalla (2007) and Bauer, Carothers and Holder (2009) evaluated the performance of Time Warp on more modern HPC systems. Barnes et al. (2013) broke simulation speed record in 2013 by employing Time Warp on the IBM's Sequoia supercomputer. Our previous study compared conservative and optimistic time management for agent-based simulations (Görür et al. 2016). In that study we showed that more scalable and faster results can be obtained by employing Time Warp.

Although Time Warp can reduce the execution time of many simulations, it is still open to improvements. Some performance optimizations were achieved by using adaptive checkpointing techniques. The goal of those techniques is finding the best checkpoint interval and dealing with the trade-off between the costs of state saving and coasting forward steps (Lin et al. 1993; Palaniswamy and Wilsey 1993; Rönngren and Ayani 1994; Fleischmann and Wilsey 1995). There are also some studies that relax causal constraints of LPs by ignoring some of the straggler messages and allowing incorrect computations (Thondugulam 1999). The challenge that arises from those studies is the trade-off between accuracy and speed. Another line of approach is examining the semantics of straggler messages. Leong, Agrawal and Agre (1993) considered the messages that can be run in any order. There are also approaches that employ Time Warp algorithm for various reasons, such as balancing the computational load among LPs (Glazer and Tropper 1993).

Spatial models are widely used in many agent-based simulations, such as the study of Epstein (2002). In that study, rebellion against authority has been modeled on a 2 dimensional grid. The usage of spatial-agent based models for socio-ecological systems was discussed by Filatova et al. (2013). Parallelization strategies for agent based models are examined by Fachada et al. (2016).

From the viewpoint of rollbacks and checkpointing, there are some overlaps between Time Warp and algorithm-based fault tolerant protocols. The main difference between them is that the main concern of algorithm-based fault tolerant protocols is detecting faults in processors and recover the system to improve the reliability of a multi-processor system (Yajnik and Jha 1997); while Time Warp deals with loosening the synchronization among LPs to improve the performance of a parallel or distributed simulation (Fujimoto 1999). On the other hand, a fault tolerant protocol for distributed systems that employs Time Warp was studied by Agrawal and Agre (1992).

In this study, we aim to improve the performance of Time Warp by not performing unnecessary rollbacks that are triggered as soon as a straggler message is received. The LPs that receive a straggler message has to start a rollback operation in conventional Time Warp algorithm to fix incorrect computations. Although some of the straggler messages do not contain any real interaction among agents, LPs perform a rollback in vain. Our method prescribes how to discover and ignore those messages to reduce the number of rollbacks without disturbing the accuracy of the simulation. Differently from the studies cited above, we saved partial states of the simulation to detect whether a rollback is needed or not.

3 IMPROVING THE PERFORMANCE OF TIME WARP

Figure 1 shows how the conventional Time Warp algorithm with periodic checkpointing works. In this example, two LPs, LP_A and LP_B, run in parallel. LP_B schedules an event to LP_A at time 13, but LP_A is notified at 20 about that event. By that time, LP_A has already processed the events between 13 and 20. LP_A is supposed to handle the event at 14 that was scheduled by LP_B, so LP_A should rollback. In the first step, LP_A restores its local state to the last saved state, in other words, the last checkpoint, and the local time of LP_A is set to the last checkpoint's time, which is 10 in this example. In the second step, the coasting-forward step is performed and the events between the checkpoint and the straggler message are processed. During the coasting forward, no communication operations are performed, because the messages have been already sent/received to/from other LPs. In the third step, the straggler message is processed in the correct order. In the fourth and last step, the events after the straggler message are re-executed and LP_A continues to its usual execution from time 14. Since this step is the usual execution of LPs, communication operations are allowed again. Before the rollback, LP_A might have sent some incorrect messages to some other LPs. During the fourth step, LP_A sends a special message, known as an anti-message, to these LPs so that they can undo the effect of the erroneous messages.

The total cost of a rollback in the conventional Time Warp algorithm is given in (1), where C_{SR}, C_{CF}, C_{RE}, and C_{AM} refer to the costs of state restoration, coasting forward, re-execution and anti-message handling steps, respectively.

$$TotalCost = C_{SR} + C_{CF} + C_{RE} + C_{AM} \tag{1}$$

To improve the performance of Time Warp, we propose extending it with saving sub-states in order to reduce the number of rollbacks. Before we introduce this method, as a preliminary, we discuss another one which we call Time Warp with state difference. Time Warp with state difference method inspects the straggler message and decides whether that message would affect the simulation or not. However, this method has a considerable overhead and does not bring a significant performance improvement. In the rest of this paper, we will refer to Time Warp with state difference method as state difference; and Time Warp with sub-state saving method as sub-state saving.

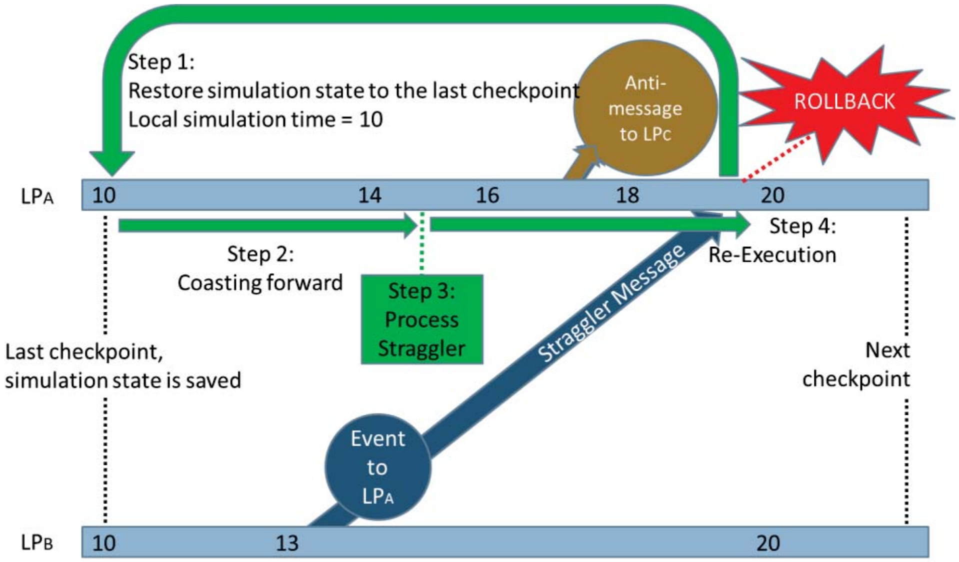


Figure 1: Flow of the Time Warp algorithm.

3.1 Time Warp with State Difference

An example flow of state difference extension of Time Warp is given in Figure 2, step by step. In this method, the goal is detecting if the straggler message would influence the simulation or not. When a straggler message is received, the LP saves its current state (step 1), in case the LP decides to ignore the straggler messages after the state difference. Then, state restoration and coasting forward steps are performed just like conventional Time Warp (steps 2 and 3). Differently from conventional Time Warp algorithm, the LP compares the "computed state" and the "must state" in the step 4. The computed state is the state that has already been computed before the straggler message is received; while the must state is the correct simulation state if the straggler message would have been processed at the right time. In order to access those states, the LP has to perform steps 2 and 3, because the state at straggler message's time had not been saved. Depending on the difference between the computed state and the must state, the LP decides whether a rollback is necessary or not. If a rollback is needed, then the usual flow of Time Warp is performed (steps 5 and 6). Otherwise, the LP ignores the straggler message and takes up where it left off before the straggler (alternative step 5 in Figure 2). That is why the LP saved its current state in step 1. Although the extra state saving (step 1) and state restoring (alternative step 5) operations are performed only once, they are still expensive, because all the agents in the LP have to be handled.

The main drawback of this method is extra state saving (step 1) and restoring operations (alternative step 4) in addition to restoration to the checkpoint and coasting forward which are still expensive. In other words, if a rollback is not necessary, step 2 and 3 are processed in vain. If the LP decides to avoid rollback, a state restoration will be required. The total cost of a rollback in the state difference method is given in (2), while C_{SS} and C_{SD} refer to the costs of state saving and state difference calculation steps, respectively.

$$TotalCost = C_{SS} + C_{SR} + C_{CF} + C_{SD} + \begin{cases} C_{RE} + C_{AM}, & if\ rollback\ is\ necessary \\ C_{SR}, & if\ rollback\ is\ not\ necessary \end{cases} \quad (2)$$

A rough comparison of the total costs of the conventional Time Warp algorithm and the state difference algorithm is as follows:

- If a rollback is needed, state difference method is more expensive than conventional Time Warp with the difference $C_{SS} + C_{SD}$.
- If a rollback is not needed, $C_{RE} + C_{AM}$ (in conventional Time Warp) and $C_{SS} + C_{SD} + C_{SR}$ (in state difference) should be compared. Since state saving and restoration operations are very expensive, state difference method will be worse than conventional Time Warp unless re-execution step is really long and expensive. Furthermore, to make state difference method perform better than conventional Time Warp, the difference between the must and computed states should be calculated by a light-weight method.

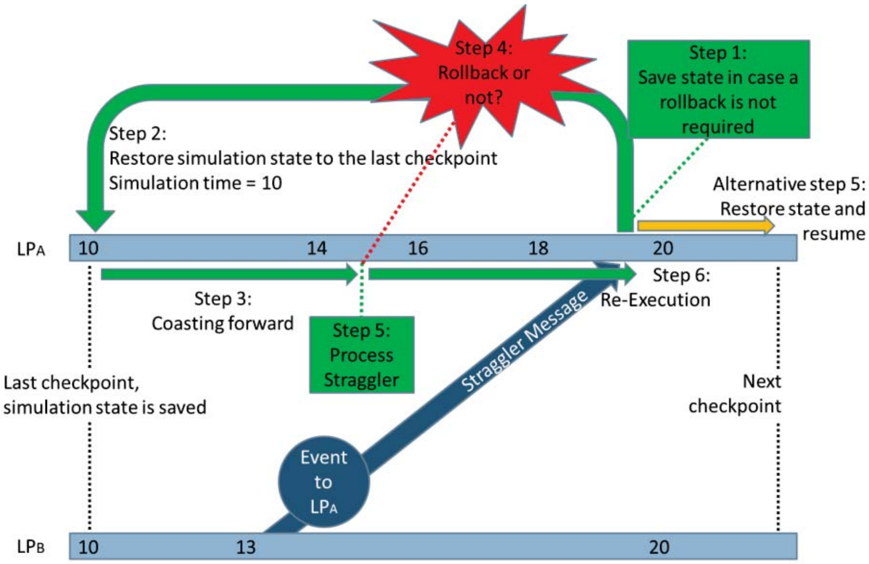


Figure 2: State difference algorithm.

In brief, the state difference extension may not worth in many cases; this is what led us into looking for a better alternative, the sub-state saving method.

3.2 Time Warp with Sub-state Saving

To overcome the drawbacks of the state difference method, straggler messages should be examined as they are received, instead of automatically going back to a past state. However, it is not always possible, because the LP does not know the earlier state of the simulation when it should have received the message. Therefore, the LP cannot find out whether the straggler message would contribute to simulation or not. To this end, we propose extending the Time Warp algorithm by sub-state saving method. Sub-state saving method prescribes all LPs to save a subset of their simulation state at every time step. These subsets are used to determine if a rollback is needed or not, when a straggler message is received. Therefore, sub-state saving method prevents an LP to start an immediate rollback. Instead, it postpones the rollback until straggler message inspection is completed. An example flow of this method is given in Figure 3.

Different from previous methods, the straggler message is inspected in step 1, as we see in Figure 1. At the end of step 1, LP_A makes a decision on performing a rollback or not. Thanks to sub-state saving method, LPs eliminate unnecessary rollbacks. The overhead of this method is saving sub-states at every time tick. Since sub-states are much smaller than full state of the simulation, the size of them is not so large. Therefore, saving sub-states is not as expensive as saving full states in terms of processors and disk. A detailed comparison between the sizes of sub-states and full states is given in the next section with an example. The total cost of a rollback in sub-state saving method is given in (3), while C_{SMI} refers to the cost of straggler message inspection in step 1.

$$TotalCost = C_{SMI} + \begin{cases} C_{SR} + C_{CF} + C_{RE} + C_{AM}, & if\ rollback\ is\ necessary \\ 0, & if\ rollback\ is\ not\ necessary \end{cases} \tag{3}$$

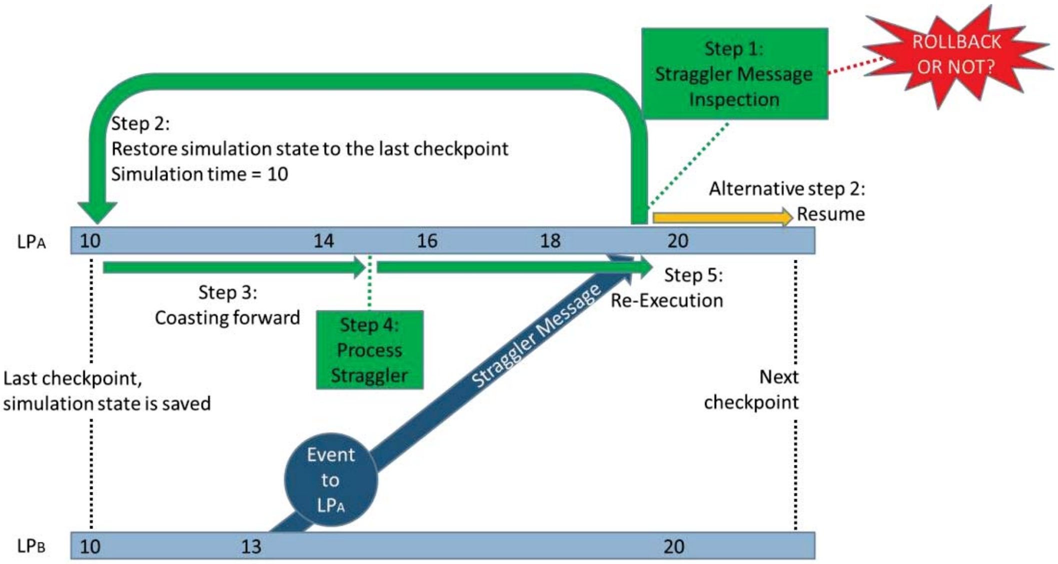


Figure 3: Sub-state saving algorithm.

In other words:

- If a rollback is required, sub-state saving method is C_{SMI} more expensive than conventional Time Warp.
- If a rollback is not required, sub-state saving method is cheaper than conventional Time Warp with the difference C_{SR} + C_{CF} + C_{RE} + C_{AM} - C_{SMI}. If straggler message inspection can be implemented as a light-weight method, sub-state saving method will worth employing.

4 IMPLEMENTATION STRATEGY FOR SPATIAL MODELS

We have implemented our method for agent-based models on a 2-dimensional spatial context. Before we present the implementation strategy, we briefly explain how RHPC_TW manages agents in space. RHPC_TW is the Time Warp extension of Repast HPC that is an open source, MPI based, distributed agent-based modeling and simulation tool (Collier and North, 2012). Repast HPC provides three types of context for developers: 2-dimensional grid, 2-dimensional continuous space and network. In Repast HPC, the workload is distributed to the LPs so that each is responsible for the same size of sub-grid as in Figure 4. In this example, there are nine LPs each managing a sub-grid that has 5x5 cells where agents live. An agent senses the other agents in its neighborhood and acts. If an agent is located in the buffer area, it can be a neighbor of an agent who belongs to another LP. Therefore, the agents in buffer areas should be visible to neighbor LPs at the right time. We have used RHPC_TW to simulate our spatial agent-based models. In the Time Warp algorithm, a rollback is performed if an LP receives the agents from a neighbor LP's buffer area at a later simulation time than it must receive. Perhaps, received agents would not cause an interaction even though they were received at the right time. Since they were not a part of any interaction, we call those agents as asocial agents. The sub-state saving method identifies the asocial agents. In spatial models, a sub-state refers to the states of the agents in the buffer. Since the set of those agents is a subset of the LP's full state, we call it a sub-state. Besides, a sub-state does not have to contain all attributes of agents, because some of them cannot be useful to determine whether a rollback is needed or not.

In our method, the interaction model should be provided by the user. Then, the simulation engine can find asocial agents and identify the straggler messages that can be ignored. To be more clear, we explain our method with an example. We have implemented Civil Violence model (Epstein 2002) in our experiments. In that model, there are four kind of agents:

- Cops who try to arrest activists,
- Activists who run away from cops,
- Quiets who wander around and calculate if they should rebel or not by considering activists and cops around,
- Jailed agents who cannot move for a specified time period.

An activist agent transforms to a jailed agent if it is arrested by a cop. After a while the jailed agent transforms to a quiet agent. If a quiet agent decides to rebel, it transforms to an activist agent. According to these principles of the Civil Violence model, the interaction model of agents is as follows:

- A cop has to see activists in its neighborhood.
- A quiet agent has to see cops and activists in its neighborhood.
- An activist has to see cops in its neighborhood.

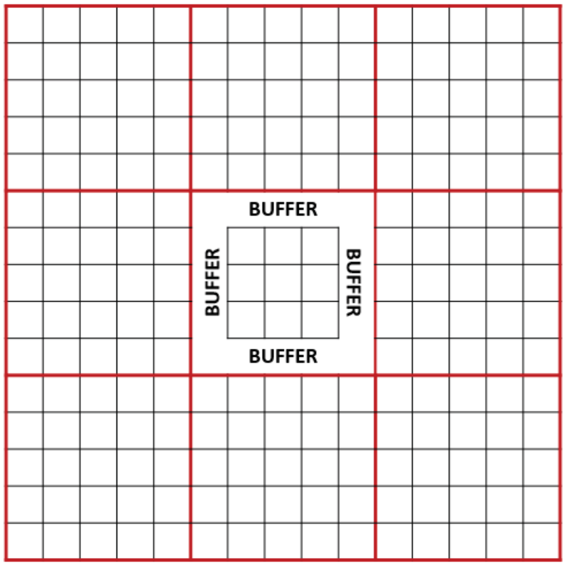


Figure 4: An example 2-dimensional grid in Repast HPC.

Therefore, LPs do not have to rollback when one of the interaction below appears:

- An activist did not see a quiet or a jailed agent at right time.
- A quiet agent did not see a jailed agent at right time.
- A jailed agent did not see any kind of agents at right time.
- A cop did not see a quiet or a jailed agent at right time.

Let’s assume that LP_A and LP_B at time tick 10 is as in Figure 5 and the buffer size is 1, it means that an LP can see one row and one column from its neighbors. LP_B is notified by LP_A about Q (a quiet agent) and J (a jailed agent) when LP_B’s local time is 15. In conventional Time Warp algorithm, LP_B does not know its past state at time tick 10, because that state had not been saved previously. So, LP_B has to perform a rollback operation by going back to the last checkpoint (assume it was at time tick 5). However, the straggler message would not cause an interaction at time tick 10, according to the interaction model above. Sub-state saving method handles this kind of cases. In sub-state saving method, LPs save some subsets that refer to the agents in buffer area, at every time tick. Saved sub-states allow LPs to examine straggler message and make a decision whether the received agents are asocial or not. Now, LP_B can realize that it had only a C (a cop agent) agent in the buffer at time tick 10. So, LP_B identifies Q and J agents as asocial agents and avoid an expensive and unnecessary rollback.

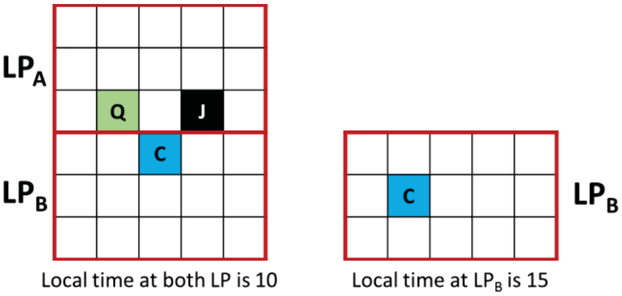


Figure 5: An example view of Civil Violence model.

The sub-state saving method has an overhead, but it is much smaller compared to the cost of saving full state. For instance, for an $N \times N$ sub-grid with the number of A agents and the buffer size of b;

- In full state saving, A agents should be saved with all attributes. Even the agents in a simple model may have many attributes. For instance, in the Civil Violence model, we have used 10 attributes (6 integer and 4 double) that include the location of the agent, agent type and other attributes specific to the model. In the GNU Compiler Collection (gcc) which we used, the sizes of integer and double variables are 4 bytes and 8 bytes, respectively. Therefore, the total size of a full-state is at least $A \times (6 \times 4 + 4 \times 8) = 56 \times A$ bytes.
- Through sub-state saving, a smaller number of agents will be saved. The number of buffer cells is $4 \times b \times (N\text{-}b)$. Therefore, there are $A \times (4 \times b \times (N\text{-}b) / N^2)$ agents in the buffer area on the average at any time. It means that $A \times (4 \times b \times (N\text{-}b) / N^2) \times (6 \times 4 + 4 \times 8) = 224 \times A \times b \times (N\text{-}b\,) / N^2$ bytes are saved at every time step on the average. Moreover, since only position (two variables of double) and type (one variable of integer) attributes of agents are saved in our case, size of the saved data will be $A \times (4 \times b \times (N\text{-}b) / N^2) \times (1 \times 4 + 2 \times 8) = 80 \times A \times b \times (N\text{-}b\,) / N^2$.

If the number of agents is 1000, the grid size is 1000 × 1000 and the buffer size is 1, then the full state saving method requires an LP to save 56000 bytes of data; while sub-state saving method requires an LP to save about 80 bytes of data on the average. This amount is the size of saved data at only one time step, but sub-states are saved at every time step. So, the proposed method actually requires $80 \times \mathit{CheckpointInterval}$ bytes of data to be saved between two checkpoints. The largest checkpoint interval size that we have seen in our experiments was 12. Therefore, total size of saved data with sub-state saving method is still much less than that with the full state saving method. This small size of data is sufficient to determine if a rollback is needed or not. Moreover, searching for which agents should be saved into a sub-state is not an overhead to the sub-state saving algorithm, because they are discovered during the buffer synchronization.

5 EXPERIMENTAL RESULTS

In the HPC system that we have used, there are 189 GB of memory and 4 AMD Opteron 6376 processors each of which has 16 cores and 2.3 GHz of clock frequency. The 40 of those cores are allowed to be employed at the same time. We have compared sub-state saving method with conventional Time Warp for the Civil Violence model on a 2-dimensional grid. We have experimented with that model with varying numbers of agents (12960 and 25920), buffer sizes (1, 2 and 3) and LPs (9, 16, 25 and 36) to observe which algorithm is better in which case. Finally, we adjusted checkpointing intervals dynamically with the additive increase multiplicative decrease method. The details of this checkpointing technique can be found in our earlier study (Görür et al. 2016).

Our method showed better performance than Time Warp in some cases. Figure 6 summarizes the results of our experiments. All of the experiments have been performed at least four times and the average of them is given here. We do not present the results for the state difference method, because they are clearly dominated by those for both conventional Time Warp and sub-state saving algorithm, as we explained in section 3.1. According to the results in Figure 6:

- When the simulation is performed with less number of LPs, conventional Time Warp algorithm shows better performance, because the overhead of sub-state saving method is considerable.
- A significant performance improvement in sub-state saving method was observed when larger number of LPs are used. Time Warp with sub-state saving catches up with the conventional Time Warp algorithm and beats it.
- Conventional Time Warp algorithm is hit by Amdahl's Law (Amdahl 1967), especially cores in different CPUs are employed. In all experiments with conventional Time Warp algorithm, using 25 LPs showed worse performance than using 16 LPs. However, employing sub-state saving method was not effected by Amdahl's Law in our experiments. Besides, the trend on the performance based on the number of LPs shows that using sub-state saving method can make conventional Time Warp algorithm more scalable when multiple CPUs are used.
- The buffer size is the same with the neighborhood range that is used in the model. Therefore, larger buffer size means more interaction potential among agents. Correspondingly, the execution times for both algorithms increase while the buffer size is increasing. Moreover, the rollback probability is directly proportional to the buffer size and number of agents. As the rollback probability increases, sub-state saving method shows better performance, because it can detect unnecessary rollbacks and avoid them. From this aspect, sub-state saving method can make Time Warp algorithm more scalable.

6 CONCLUSION

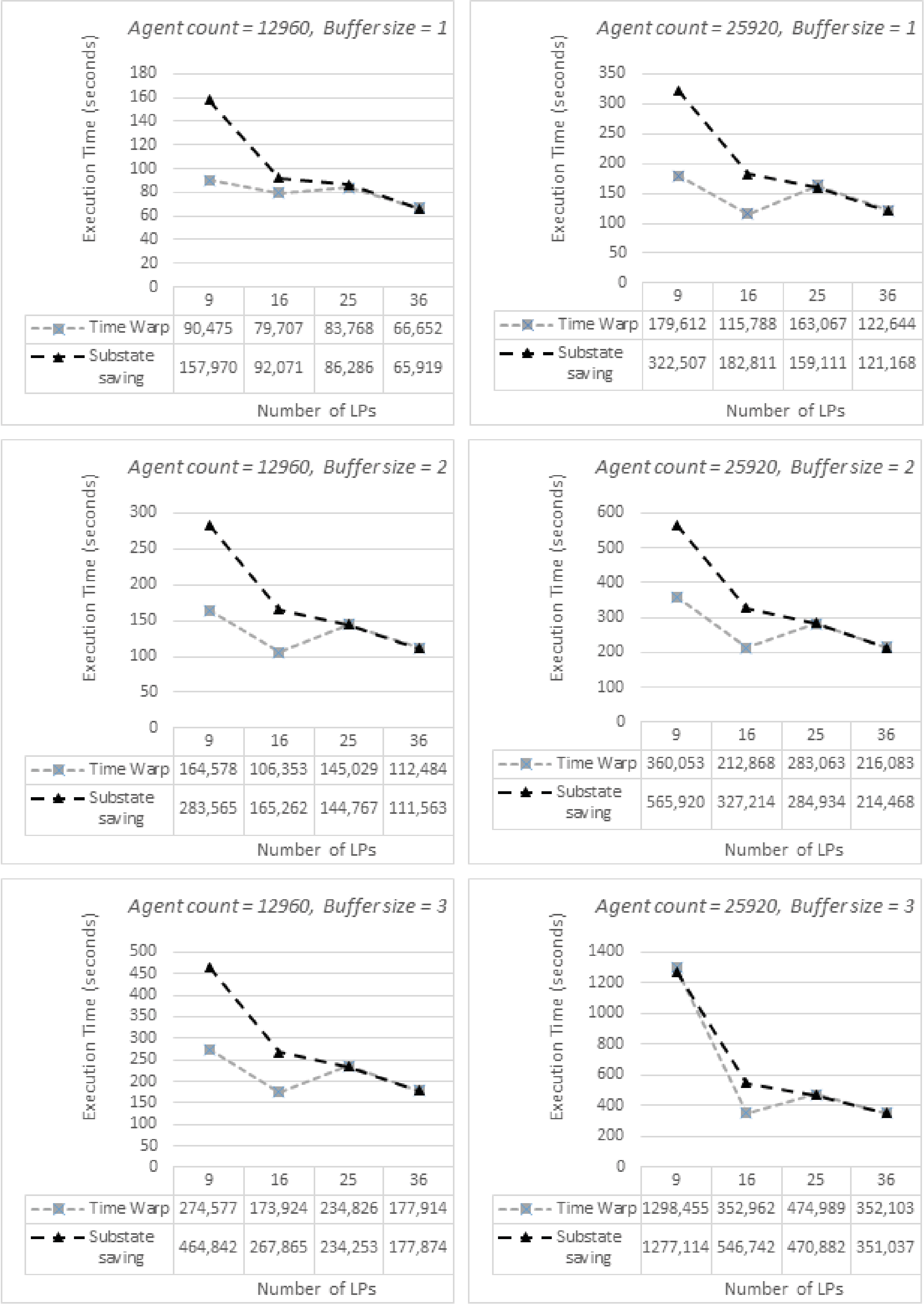
In this paper, we extended the conventional Time Warp algorithm with the sub-state saving method and compared it with the conventional Time Warp algorithm on a spatial model. An implementation strategy for agent-based models where agents live in a 2-dimensional grid is presented, as well. We introduced asocial agents that cause an LP to rollback in conventional Time Warp algorithm, although they do not interact with other agents. Through saving sub-states, we showed that getting rid of unnecessary rollbacks is possible by handling asocial agents.

Our experiments showed that sub-state saving method improves the performance of Time Warp, especially if the simulation is performed by large number of LPs. As the number of LPs increase, the execution time of simulation that employs sub-state saving method reduces much more than conventional Time Warp algorithm. Therefore, Time Warp algorithm can become more scalable by using sub-state saving method. Reduced number of rollbacks also brings an implicit advantage that improves the performance of Time Warp algorithm. Since our checkpointing algorithm adjusts the checkpoint interval considering rollbacks, the checkpoint intervals become larger when sub-state saving algorithm is employed comparing to conventional Time Warp algorithm.

Although sub-state saving method improves the performance of Time Warp algorithm, it comes with an overhead that cannot be ignored. This overhead is completely dependent on the buffer size and the number of agents in buffer area. As a future work, we are planning to reduce the cost of saving sub-states. We are also planning to experiment our method in other HPC systems that have more resources. We are going to compare the performance of our method with different models to investigate its generality.

ACKNOWLEDGEMENTS

We would like to thank the Department of Computer Engineering, Middle East Technical University, Ankara, Turkey, for the access provided to their parallel computing resources.



Agent count = 12960, Buffer size = 1

	9	16	25	36
Time Warp	90,475	79,707	83,768	66,652
Substate saving	157,970	92,071	86,286	65,919

Agent count = 25920, Buffer size = 1

	9	16	25	36
Time Warp	179,612	115,788	163,067	122,644
Substate saving	322,507	182,811	159,111	121,168

Agent count = 12960, Buffer size = 2

	9	16	25	36
Time Warp	164,578	106,353	145,029	112,484
Substate saving	283,565	165,262	144,767	111,563

Agent count = 25920, Buffer size = 2

	9	16	25	36
Time Warp	360,053	212,868	283,063	216,083
Substate saving	565,920	327,214	284,934	214,468

Agent count = 12960, Buffer size = 3

	9	16	25	36
Time Warp	274,577	173,924	234,826	177,914
Substate saving	464,842	267,865	234,253	177,874

Agent count = 25920, Buffer size = 3

	9	16	25	36
Time Warp	1298,455	352,962	474,989	352,103
Substate saving	1277,114	546,742	470,882	351,037

Figure 6: Performance comparison of conventional Time Warp algorithm and Time Warp algorithm with Substate Saving**.**

REFERENCES

Agrawal, D., and J. R. Agre. 1992. “Recovering from multiple process failures in the time warp mechanism”. *IEEE Transactions on Computers* vol 41, pp. 1504–1514.

Amdahl, G. M. 1967. “Validity of the single processor approach to achieving large scale computing capabilities”. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pp. 483–485.

Barnes Jr, P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. “Warp Speed: Executing Time Warp on 1,966,080 Cores”. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 327–336. Montreal, Quebec, Canada, ACM.

Bauer Jr, D. W., C. D. Carothers, and A. Holder. 2009. “Scalable Time Warp on Blue Gene Supercomputers”. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation,* pp. 35–44. IEEE Computer Society.

Collier, N., and North, M. 2012. “Parallel Agent-Based Simulation with Repast for High Performance Computing”. *Simulation: Transactions of the Society for Modeling and Simulation International* vol 89, pp. 1215–1235.

Epstein, J. M. 2002. “Modeling Civil Violence: An Agent-Based Computational Approach”. In *Proceedings of the National Academy of Sciences of the United States of America (PNAS)* vol 99, pp. 7243–7250.

Fachada, N., V. V. Lopes, R. C. Martins, and A. C. Rosa. 2016. “Parallelization Strategies for Spatial Agent-Based Models”. *International Journal of Parallel Programming*, pp. 1–33.

Filatova, T., P. H. Verburg, D. C. Parker, and C. A. Stannard. 2013. “Spatial Agent-Based Models for Socio-Ecological Systems: Challenges and prospects”. *Environmental Modelling & Software* vol. 45, pp. 1–7.

Fleischmann, J., and P. A. Wilsey. 1995. “Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators”. *ACM SIGSIM Simulation Digest* vol 25, pp. 50–58.

Fujimoto, R. M. 1999. *Parallel and Distributed Simulation Systems*. New York, USA, John Wiley & Sons, Inc.

Glazer, D. W., and C. Tropper. 1993. “On Process Migration and Load Balancing in Time Warp”. *IEEE Transactions on Parallel and Distributed Systems* vol 4, pp. 318–327.

Görür, B. K., K. İmre, H. Oğuztüzün, and L. Yilmaz. 2016. “Repast HPC with Optimistic Time Management”. In *Proceedings of the 24th High Performance Computing Symposium (HPC’ 16),* pp. 23–31. Pasadena, California, Society for Computer Simulation International.

Jafer, S., G. A. Wainer. 2010. “Conservative vs. Optimistic Parallel Simulation of DEVS and Cell-DEVS: A Comparative Study”. In *Proceedings of the 2010 Summer Computer Simulation Conference (SCSC ’10)*, pp. 342–349. Ottawa, Ontario, Canada, Society for Computer Simulation International.

Jefferson, D. R. 1985. “Virtual Time”. *ACM Transactions on Programming Languages and Systems* vol 7, pp. 404–425.

Leong, H. V., D. Agrawal, and J. R. Agre. 1993. “Using Message Semantics to Reduce Rollback in the Time Warp Mechanism”. In *Distributed Algorithms: 7th International Workshop, WDAG'93 Lausanne, Switzerland, September 27--29, 1993 Proceedings*, edited by Andrê Schiper, pp. 309–323. Berlin, Springer Berlin Heidelberg.

Lin, Y-B., B. R. Preiss, W. M. Loucks, and E. D. Lazowska. 1993. “Selecting the Checkpoint Interval in Time Warp Simulation”. *SIGSIM Simulation Digest* vol 23, pp. 3–10.

Palaniswamy, A. C., and P. A. Wilsey. 1993. “An Analytical Comparison of Periodic Checkpointing and Incremental State Saving”. *SIGSIM Simulation Digest* vol 23, pp. 127–134.

Perumalla, K. S. 2007. “Scaling Time Warp-based Discrete Event Execution to 10^4 Processors on a Blue Gene Supercomputer”. In *Proceedings of the 4th International Conference on Computing Frontiers*, pp. 69–76. Ischia, Italy, ACM.

Presley, M. T., P. L. Reiher, and S. F. Bellenot. 1990. “A Time Warp Implementation of Sharks World”. In *Proceedings of the 22nd Conference on Winter Simulation (WSC’ 90)*, pp. 199–203.

Rönngren, R., and R. Ayani. 1994. “Adaptive Checkpointing in Time Warp”. *ACM SIGSIM Simulation Digest* vol 24, pp. 110–117.

Thondugulam, N. V., D. M. Rao, R. Radhakrishnan, and P. A. Wilsey. 1999. “Relaxing Causal Constraints in PDES”. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pp. 696–700.

Yajnik, S., and N. K. Jha. 1997. “Graceful Degradation In Algorithm-Based Fault Tolerant Multiprocessor Systems”. *IEEE Transactions on Parallel and Distributed Systems* vol 8, pp. 137–153.

AUTHOR BIOGRAPHIES

B. KAAN GÖRÜR is a specialist computer engineer at Roketsan A.S. in Ankara, Turkey. He is also a PhD student in Department of Computer Engineering in Hacettepe University. His research interests are parallel and distributed simulation, agent based modeling and simulation, model driven engineering, and simulation visualization. His email address is bkaangorur@gmail.com.

KAYHAN İMRE is an assistant professor at the Department of Computer Engineering, Hacettepe University, Ankara, Turkey. He received his Ph.D. degree in Computer Science from University of Edinburgh, Scotland in 1993. His research interests are parallel computing, parallel and distributed simulation, real-time systems and photonic networks. His email address is ki@hacettepe.edu.tr

HALİT OĞUZTÜZÜN is a professor at the Department of Computer Engineering, Middle East Technical University (METU), Ankara, Turkey. He got his Ph.D. in Computer Science from University of Iowa, Iowa City, IA in 1992. His current research interests are model-driven engineering and distributed simulation. His email address is oguztuzn@ceng.metu.edu.tr

LEVENT YILMAZ is Professor of Computer Science and Software Engineering at Auburn University with a joint appointment in Industrial and Systems Engineering. He holds M.S. and Ph.D. degrees in Computer Science from Virginia Tech. His research interests are in agent-directed simulation, cognitive computing, and model-driven science and engineering for complex adaptive systems. He is the former Editor-in-Chief of Simulation: Transactions of the SCS and the founding organizer of the Agent–Directed Simulation Conference series. His email address is yilmaz@auburn.edu.

OPTIMIZING ENERGY CONSUMPTION IN GPUS THROUGH FEEDBACK-DRIVEN CTA SCHEDULING

Amin Jadidi
Mohammad Arjomand
Mahmut Taylan Kandemir
Chita R. Das
Department of Electrical Engineering and Computer Science
The Pennsylvania State University
342 Information Sciences and Technology Building
University Park, PA, USA
{axj945,mxa51,kandemir,das}@cse.psu.edu

ABSTRACT

Emerging GPU architectures offer a cost-effective computing platform by providing thousands of energy-efficient compute cores and high bandwidth memory that facilitate the execution of highly parallel applications. In this paper, we show that different applications, and in fact different kernels from the same application might exhibit significantly varying utilizations of compute and memory resources. In order to improve the energy efficiency of the GPU system, we propose a run-time characterization strategy that classifies kernels as compute- or memory-intensive based on their resource utilizations. Using this knowledge, our proposed mechanism employs core shut-down technique for memory-intensive kernels in order to manage energy in a more efficient way. This strategy uses performance and memory bandwidth utilization information to determine the ideal hardware configuration at run-time. The proposed technique saves on average 21% of total chip energy for memory-intensive applications, which is within 8% of the optimal saving that can be obtained from an oracle scheme.

1 INTRODUCTION

GPUs are being increasingly employed to accelerate different types of computing platforms ranging from embedded devices to supercomputers. As a result, today's GPUs are not running only graphics applications, but also, applications from database domain and high-performance computing domain, among others. This makes GPU a general-purpose computing platform or shortly GPGPU. To cope with contrasting demands of these different types of applications, GPU architects keep increasing on-chip resources such as cores, caches, software-managed memories and memory controllers (MCs); and projections include even more powerful resource-rich GPU systems in future. An important issue at this juncture is whether current applications effectively utilize available on-chip resources in GPUs and, if not, *what are the reasons behind it?* and more importantly, *which classes of design optimizations can be done in GPGPUs to improve their performance/energy per cost?* Recent studies (Hong and Kim 2010, Lee et al. 2011, Leng et al. 2013, Rogers et al. 2012) have focused on this resource utilization problem and proposed techniques that handle underutilized hardware components in GPUs. Here, we look at resource utilization and the promising energy optimization based on run-time characteristics of the kernels.

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

Our experimental study shows that both compute cores and memory bandwidth are underutilized in many GPU applications. In highly memory-intensive applications, cores get stalled frequently because all the threads are waiting for the memory requests. On the other hand, in compute-intensive applications, memory experiences long idle periods. Moreover, as different kernels execute different parts of the same application, they may exhibit significant variations regarding utilizations of compute cores and memory bandwidth, making a universal solution that works across different applications highly unlikely. Motivated by this observation, this paper proposes an energy-saving strategy that exploits resource underutilization in GPUs. More precisely, in this work we focus on core underutilization in GPUs. With the knowledge that increasing the degree of parallelism does not necessarily improve the performance (due to congestion in the interconnection network, contention in the last level cache, and memory bandwidth saturation (Hong and Kim 2010, Guz et al. 2009)), we propose a strategy to regulate the number of active cores when they are over provisioned for a particular memory-intensive kernel. Our proposed mechanism manages the degree of parallelism through a feedback-driven CTA scheduling and adopts a core shut-down strategy to manage energy consumption. Although a warp-throttling technique (Rogers et al. 2012) could be employed to resolve the existing problem on memory side by reducing the number of concurrently running threads, our feedback-driven CTA scheduler is compatible with core shut-down techniques. Note that, a core shut down mechanism also reduces the static leakage power which contributes to a considerable portion of the total chip power consumption (Leng et al. 2013). We also demonstrate that our mechanism outperforms a typical core-side DVFS technique.

The rest of the paper is organized as follows. Section 2 provides background on our target GPU architecture. In Section 3, we describe the resource underutilization problem in GPUs. In Section 4, we present our runtime technique for regulating number of active cores through feedback-driven CTA scheduling. Section 5 presents an experimental evaluation of the proposed strategy. Section 6 reviews related works and Section 7 concludes the paper.

2 BACKGROUND

In this section, we provide a brief background on the GPU architecture targeted by our work.

GPU Architecture: Our target GPU consists of multiple streaming multiprocessors (SMs) (Terms "core" and "SM" are used interchangeably in this paper.), each containing 32 CUDA cores (Fermi 2009). Each CUDA core can execute a thread, in a "Single-Instruction, Multiple-Threads" fashion. This architecture is supported by a large register file that hides the latency of memory accesses. The memory requests generated by multiple concurrently executing threads in an SM are coalesced into fewer cache lines and sent to L1 data cache, shared by all CUDA cores in the SM. Memory requests are injected into the network, which connects the SMs to 6 memory partitions through a crossbar. Each memory partition includes a slice of shared L2 cache, and a memory controller. Figure 1 shows this baseline architecture. We further assume that the system supports per-SM power-gating (Hong and Kim 2010).

GPGPU Applications: A typical GPGPU application consists of one or multiple kernels each of which is launched once or multiple times during the entire execution of the application. These kernels implement specific modules of an application. Each kernel consists of a set of parallel threads. As shown in Figure 1, these threads are divided into groups of threads, called Cooperative Thread Arrays (CTAs). The underlying architecture further divides each CTA into groups of threads, called warps, that is transparent to the programmer. The execution on GPU starts with performing memory allocation in GPU memory. Then, CPU copies the required data into the allocated memory, and a kernel is launched on GPU. After a kernel is launched, the CTA scheduler schedules available CTAs associated with the kernel on all the available cores. The maximum number of CTAs per core is limited by core resources (i.e., number of threads, size of shared memory, register file size, and etc). In a finer granularity, the CTA assignment policy is followed by per-core warp scheduling. Warps associated with CTAs are scheduled on the assigned cores and get equal priority. Once a core finishes executing of a CTA, the CTA scheduler assigns another CTA to that core to execute. In such scheduling mechanism, there is no priority among CTAs and the process continues until all the CTAs are executed. When all the threads finish their computation, the results are copied to the CPU memory and the GPU memory is freed. At this point the CPU can launch the next kernel on the GPU. In Section 4, we explain how our sampling and reconfiguration mechanism cooperates with the CTA scheduler at run-time.

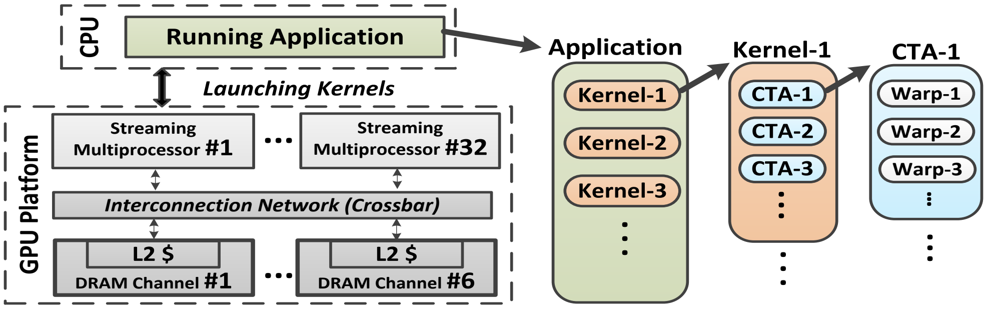


Figure 1: Target GPGPU architecture and the details of the computation hierarchy in GPGPU applications.

3 MOTIVATION

3.1 Investigating Resource Underutilization

The main philosophy of GPGPU architectures is to provide a large number of computing cores supported by a high bandwidth memory, in order to have a high throughput system. Such a resource-rich design will be cost-effective only if the main resources such as computing cores and memory are effectively utilized by hosted application. Thus, it is vital to understand the impact of different applications on the utilization of GPU resources. Typically, memory-intensive applications utilize the memory bandwidth properly, but might not need all the cores to achieve the optimal performance. In this section, we discuss different type of applications/kernels in terms of core utilization and its relationship to the available memory bandwidth.

• **Core Utilization at the Application Level:** To illustrate the effect of the number of cores on the system performance, let us examine Figure 2 (our experimental setup will be given in Section 5.1). This figure shows the application performance, as we vary the number of available cores. Among these applications, PATH is the only compute-intensive application, and its performance increases linearly as we increase the number of cores. The other two applications are memory-intensive, and we observe that their performance does not improve beyond a certain point. In fact, we even observe some performance loss in BFS as we allocate it more than about 20 cores. Since each thread has a certain memory bandwidth demand, as we increase the number of cores we also increase the number of memory transactions from cores to memory per unit of time. Beyond a certain number of cores, this increase in memory requests could cause the memory bandwidth to saturate. Therefore, beyond this point, using additional cores will not improve performance; instead, it could lead to longer memory accesses latencies. Furthermore, it could also cause too many contentions in the last level cache (as it is the case for BFS application) which consequently degrades the performance.

• **Core Utilization at the Kernel Level:** In order to have a more detailed analysis on the impact of number of active cores on the system performance, we investigate our applications at a finer granularity, i.e., at the *kernel level*. Each GPGPU application consists of one/multiple kernel(s), each of which is launched once/multiple-times during the execution of that application. Based on our observations, not only different applications but also different kernels that belong to the same application might exhibit a large variance in their resource demands, leading to diverse resource utilizations across different phases of the application.

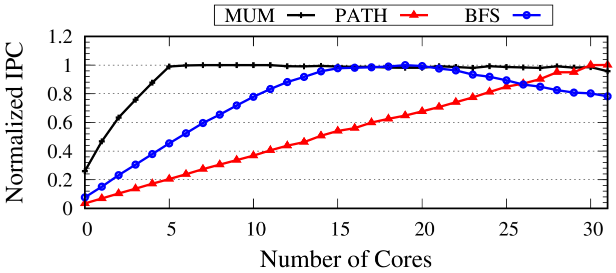


Figure 2: Effect of increasing the number of cores on the performance of different applications. The results are normalized to the highest IPC observed in each application over 32 different core allocations. (Each application consists of one main kernel that is called multiple times during the course of execution.)

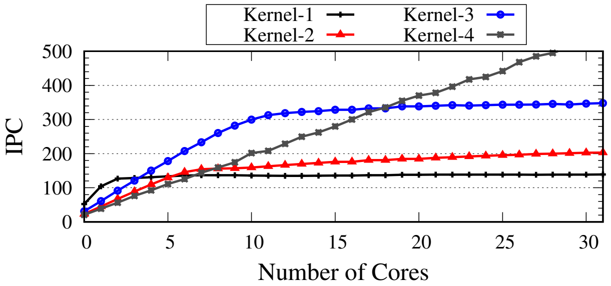


Figure 3: Effect of increasing the number of cores on the performance of four different kernels from MST application. Kernel-4 exhibits compute-intensive behavior while other kernels have memory-intensive characteristics each of which with different saturation point.

Figure 3 shows the effect of the number of cores on the performance of four different kernels from MST application. Each of these kernels is executed many times during the course of execution of this application. Considering the performance scalability of a kernel as the number of cores is varied, each of these kernels has a different saturation point where the performance does not increase with more compute resources. In this case, Kernel-4 can be classified as compute-intensive while the other three kernels are relatively memory-intensive. As can be seen, Kernel-1 performance saturates by 3 cores while Kernel-2 and Kernel-3 can effectively utilize by up to 13 and 20 cores, respectively. Thus, in order to accurately regulate number of active cores, each individual kernel of an application should be analyzed separately.

4 GPU RESOURCE MANAGEMENT

In this section, we describe our strategy to find the ideal SM count for each individual kernel. First, we discuss our dynamic CTA-based kernel characterization mechanism. Then, we explain how our proposed scheme uses the collected statistics to converge to the optimal number of SMs through feedback-driven CTA scheduling.

4.1 Run-Time Characterization

In order to investigate the effect of available compute and memory resources on the overall system performance, we monitor the memory bandwidth utilization (MBU) as well as instruction per cycle (IPC) metrics. Once a kernel is launched by the CPU, we allocate all the available SMs to that kernel. The next step is to perform sampling to study the characteristics of the running kernel. In our experimental study, we observed that assuming a fixed window size for sampling phase (in terms of number of cycles) is not accurate because the execution time of different kernels are widely variable for a fixed window size. Therefore, our goal is to have a dynamic window size for the sampling phase to accurately capture the behavior of each individual kernel at run-time. To this end, we analyze kernel-based and CTA-based sampling techniques.

• **Kernel-Based Sampling:** Within a majority of GPGPU applications, two basic properties exist. First, most of the kernels of an application are launched multiple times during the execution of the application. Second, different invocations of the same kernel exhibit very similar behavior. These two common properties motivate us to exploit the first execution of a kernel as the sampling phase. Note that, such kernel-based approach is not applicable for two different situations: First, if the kernel is launched only once during the execution of the program. Second, if the kernel does not exhibit consistent behavior over different invocations.

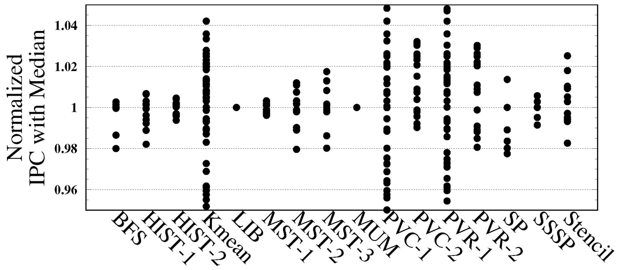


Figure 4: Normalized IPC of different kernels during the execution of the application. Kmean, PVC1, and PVR1 do not exhibit consistent behavior over different invocations. LIB and MUM consist of one main kernel that are executed only once.

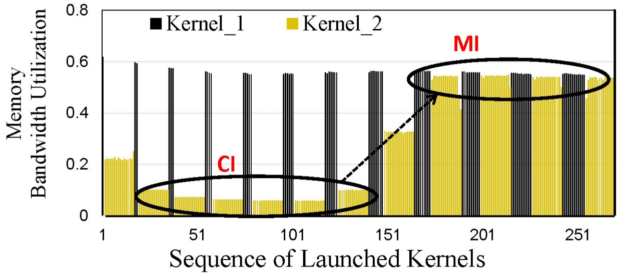


Figure 5: Transitioning between the compute-intensive (CI) and memory-intensive (MI) states in PVC. Kernel-1 exhibits memory-intensive behavior. Kernel-2 however, experiences transitions between compute-intensive and memory-intensive phases.

Figure 4 shows the normalized IPC of different kernels over different invocations. As can be seen, most of the kernels exhibit consistent behavior over different invocations. However, LIB and MUM consist of only one kernel which are executed only once. Kmean, PVC1, and PVR1 on other hand, although executed multiple times, exhibit inconsistent behavior over different executions. Figure 5 demonstrates this issue for PVC more clearly. Kernel-1 represents a memory-intensive kernel with consistent behavior. Kernel-2 however, transits between compute and memory-intensive phases. Therefore, we cannot use one execution to optimize future invocations of that kernel. Besides these two drawbacks, a kernel-based approach does not capture the behavioral transitions during each kernel execution.

• **CTA-Based Sampling:** As described in Section 2 (see Figure 2), each kernel is split into smaller blocks called CTAs that execute similar portions of the code. SMs start executing a kernel with the maximum possible number of CTAs, and whenever a CTA finishes, the CTA scheduler launches another CTA (if any) to the available SM. This procedure continues until all the CTAs are finished. Because these CTAs run similar code, each of them can represent the behavior of the kernel. This property motivates us to employ a CTA as a fine yet accurate granularity in our sampling process. Based on our experimental observations, in order to have an accurate sampling, we consider the first group of scheduled CTAs (i.e., #SMs*#CTAs-Per-SM) as the sampling window. However, the monitoring hardware continuously analyzes the behavior of running kernel, and if it recognizes noticeable changes in the behavior of running kernel, it will accordingly update the collected statistics. Unlike kernel-based approach, a CTA-based scheme can be exploited for (i) kernels that are launched only once, (ii) kernels with inconsistent behavior over different invocations, and (iii) also recognizes the behavioral changes within each kernel execution. In this work, we adopt such CTA-based sampling for our characterization purposes.

4.2 Memory-Intensive Kernels

Theoretically, as long as the available memory bandwidth is not saturated (i.e., MBU < 100%), we expect to see performance improvement as we increase the number of SMs. However, in our experimental studies we observed that the performance of memory-intensive kernels get saturated when MBU is much less than 100%. For instance, in SP, using all 32 SMs causes severe contention in last level cache while MBU is only about 60%. If we keep reducing the number of active SMs down to 11, we still observe the same IPC for SP. In other words, in memory-intensive kernels, memory bandwidth is not the only bottleneck and the running kernel might instead suffer from contention in the last level cache and/or congestion in the interconnection network while memory bandwidth is not saturated.

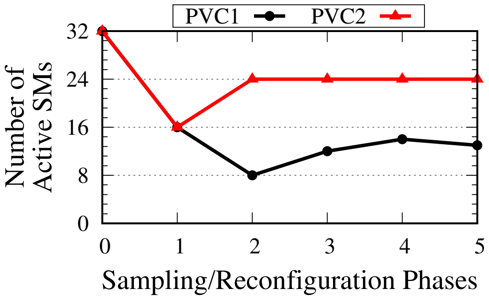


Figure 6: Converging to the optimal number of SMs for two kernels from PVC. Binary search process takes 5 (log(#SMs)) sampling/reconfiguration iterations to converge to the optimal point.

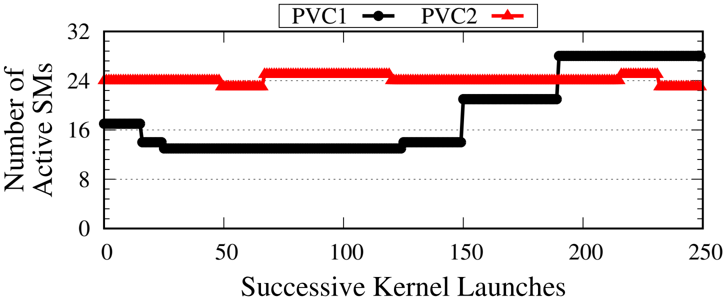


Figure 7: Different invocations of the same kernel might exhibit different behaviors: Our scheme assigns different number of SMs to PVC1 as it exhibits widely different characteristics over different invocations. In contrast, PVC2 behavior is very consistent over the course of execution. (see Figures 4 and 5)

Based on our experimental evaluations, we consider 50% memory bandwidth utilization as the primary threshold to classify a kernel as memory-intensive. In other words, kernels with MBU above that threshold could be potentially using too many SMs. Once a kernels is recognized as memory-intensive, our feedback-driven CTA scheduler gradually reduces the number of running CTAs to find the ideal number of CTAs/SMs for that kernel. Note that, the existence of such threshold is not essential. Meaning that, we can perform our optimization scheme on all kernels. However, this approach causes useless executions of our optimization scheme which could hurt the performance of compute-intensive kernels during the search process. Therefore, this threshold only eliminates wasteful optimization processes for non-memory-intensive kernels.

4.3 Regulating Number of Active Cores

The idea behind our proposed technique is to monitor IPC and MBU statistics to determine the ideal number of active SMs accordingly. The goal is to assign minimum possible number of SMs to the running kernel without losing performance. To determine the ideal number of active SMs, we employ a feedback driven CTA scheduling approach which changes the number of running CTAs over multiple samplings. To do so, we first run the kernel when it has been allocated all the SMs. If that kernel is recognized as memory-intensive, our scheme allocates that kernel fewer number of SMs, and recollects the statistics to evaluate the impact of reconfiguration on IPC and MBU. In order to converge to the ideal point faster, in this work we used a binary search for regulating number of active SMs. The binary search takes *log(Number-of-SMs)* steps to find the ideal answer. Our base architecture has 32 SMs; consequently, our scheme converges to the ideal number of SMs after 5 steps. In the proposed procedure, we compare the IPC of the new configuration with the IPC of the very first sampling phase that had all the SMs activated. When we compare two IPCs from two different configurations, we consider a small margin for IPC variations, meaning that as long as$|IPC_i - IPC_1| \leq \alpha$, we technically have the same performance. When we cross the saturation point (discussed in Figures 2 and 3) and step in the linear part of the performance curve, the new observed IPC will be considerably less than IPC_1 and our scheme accurately detects the saturation point. Figures 6 represents an optimization process for two kernels from PVC. As can be seen, search process stabilizes after 5 (log(#SMs)) sampling/reconfiguration phases. In this example, PVC1 and PVC2 are assigned 13 and 24 SMs, respectively. Figure 7 on the other hand, shows the number of assigned SMs to each kernel over different invocations. In Figures 4 and 5 we observed that PVC2 exhibits similar characteristics over different executions. Consequently, our scheme assigns almost the same number of SMs to PVC2 over different executions. PVC1 however, as discussed in Figures 4 and 5 , experiences widely different compute- and memory-intensive phases over time; therefore, the number of dedicated SMs to that changes from 13 to 28 SMs over different invocations.

Note that, off-line characterization and optimization techniques cannot recognize such variations over different executions. Therefore, techniques like (Hong and Kim 2010) fail in determining the optimal number of SMs especially for cases like PVC1 with unpredictable run-time behaviour.

Pausing Technique. CTAs, once assigned to a core, cannot be preempted, or assigned to another core (NVIDIA Corporation 2010). Therefore, during successive sampling and reconfiguration periods, we pause the SMs instead of shutting them down. Such pausing approach during the sampling phase will not cause any migration/context-switch overhead among SMs. All this process is managed by our feedback driven CTA scheduler. After determining the ideal number of active SMs (after 5 sampling and reconfiguration steps), we use a SM power-gating mechanism to turn the rest of the SMs off once they finish executing previously assigned CTAs.

Predict When to Reactivate the SMs. Once a kernel execution is finished, the CPU launches a new kernel on the GPU. At this point, the hardware needs to allocate all the available SMs to the new kernel. However, some of the SMs might be shut down for the previous kernel and the delay of powering them on could negatively affect performance of the newly launched kernel. To avoid this, we predict the time when a kernel will be finished. This can be implemented by hardware since it can measure the average execution time of a CTA at run-time (i.e., in terms of number of cycles). Therefore, based the number of left CTAs to schedule, and the average execution length of a CTA, we can determine when to reactivate the power-gated SMs in order to overlap the SM reactivation delay with the remaining execution time of the running kernel.

Hardware Overhead. Our proposed scheme monitors IPC and MBU to regulate number of active SMs in a feedback-driven fashion. To collect this information at run time, we assume that each SM has 2 counters (overall, 32*2*4 Bytes) to track the number of executed instructions as well as the number of cycles. Besides, each memory channel needs 2 counters (overall, 6*2*4 Bytes) to track the number of memory transactions and number of memory cycles. Considering all the performance counters, the proposed mechanism has an overall capacity overhead of 304 bytes.

5 EXPERIMENTAL RESULTS

5.1 Methodology

Platform: In order to evaluate our proposal, we used GPGPU-Sim v3.2.2 (Bakhoda et al. 2009), a publicly-available cycle-accurate GPGPU simulator. The details of the simulated configuration are listed in Table 1a. This configuration is similar to GTX480 configuration. In our experiments, we changed the number of active SMs between 1 and 32, and used 32 in our baseline. Each SM is supported by a separate 16KB L1D and L1I caches. SMs are connected to 6 memory channels. Each memory channel is coupled with a portion of L2 cache with a size of 256KB. Misses in L2 cache are sent to the memory (Rixner et al. 2000).

Benchmarks: Table 1b lists the applications we used in our evaluations. We consider a wide range of memory-intensive applications from various benchmark suites: CUDA SDK (NVIDIA 2011), Parboil (Stratton et al. 2012), Mars (He et al. 2008), Shoc (Danalis et al. 2010), and LonestarGPU (Burtscher et al. 2012). We classify the kernels as compute-intensive (COMP), and memory-intensive (MEM) in Table 1b. As can be seen, each of the studied applications consists of at least one memory-intensive kernel.

Performance Metrics: In this work, we focus on energy efficiency, thus we report three metrics. First, we report application performance in terms of normalized IPC with respect to the baseline configuration described in Table 1a. Second, we report the power consumption of the system using GPUWattch (Leng et al. 2013). In particular, we focus on dynamic power, leakage power, and DRAM power. Third, based on performance and power results, we calculate the energy consumption of the system. The results presented below includes all the runtime overheads brought by our approach.

(a) Baseline configuration.

SM Config.	32 Shader Cores, 1400MHz, SIMT Width=32
Resources / Core	1536 Threads (48 warps, 32 threads/warp), 48KB Shared Memory, 32684 Registers
Caches / Core	16KB 4-way L1D, 12KB 24-way Texture, 8KB 2-way Constant Cache, 2KB 4-way L1I
L2 Cache	256 KB/Memory Partition, 128B Line Size, 8-way, 700MHz
Warp Scheduler	Greedy-then-oldest (Rogers et al. 2012)
Features	Memory Coalescing, Inter-warp Merging, Immediate Post Dominator (Fung et al. 2007)
Interconnect	Crossbar, 1400MHz, 32B Channel Width
Memory Model	6 GDDR5 MCs, 2GHZ, 1 V FR-FCFS, 8 DRAM-banks/MC
GDDR5 Timing	$t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$

(b) List of GPU benchmarks: In the last column, MEM and COMP refers to memory- and compute-intensive behavior of the kernels.

Suite	Application	Abbr.	Kernel Type
Lonestar	Single-Source Shortest Paths	SSSP	MEM
Lonestar	Breadth-First Search	BFS	MEM
Lonestar	Survey Propagation	SP	MEM
Lonestar	Minimum Spanning Tree	MST	MEM-COMP
Parboil	Saturating Histogram	HIST	MEM-COMP
Shoc	2D Stencil Computation	stencil	MEM
SDK	MUMerGPU	MUM	MEM
SDK	LIBOR Monte Carlo	LIB	MEM
Mars	Kmeans Clustering	Kmean	MEM-COMP
Mars	Page View Count	PVC	MEM-COMP
Mars	Page View Rank	PVR	MEM-COMP

Table 1: Experimental setup in our evaluations.

5.2 Evaluation

In this section we analyze the impact of our proposed mechanism on the chip energy consumption, and overall system performance.

Static and Dynamic Power Consumption: Figure 8 reports the breakdown of total energy consumption in terms of static and dynamic power ratios. For each application, we have reported the energy saving gained by our proposed technique as well as the energy saving of the optimal configuration (optimal configuration is the configuration with lowest energy consumption while performance loss is kept less than 2% compared to the baseline. In order to find the optimal point, we ran the system under all 32 possible SM assignments.). As can be seen, for most of the applications the static power contribution in overall improvement is dominant which is achieved by power-gating some of the SMs. In this category of application, regulating the number of SMs does not affect the system performance. However, since we use less number of active SMs, the power-gating can effectively reduce leakage power consumption. The energy saving gained by lowering the static power consumption is linearly dependent on the number of power gated SMs.

For some of the kernels/applications (i.e., BFS, MUM, SP, and SSSP), we also observe improvements in the dynamic power consumption. As we keep decreasing the number of active SMs, for kernels running in the saturated region, we could potentially decrease the dynamic power consumption by shortening the execution time. For instance, reducing the number of active SMs from 32 to 20 in BFS, reduces the contentions in last level cache such that the miss-rate reduces from 77% to 45% which consequently improves IPC by 25%. We observed similar impacts in MUM, SP, and SSSP. Overall, our proposed mechanism achieves up to 35% and on average about 21% energy saving, which is within 8% of the optimal saving.

Performance: Figure 9 reports the average number of SMs that our proposed scheme stabilizes at for each application. Our proposed scheme reduces the number of active SMs to as low as 10 SMs and with an average of 28 SMs. Figure 10 demonstrates the impact of our proposed technique on system performance. As can be observed, four of those applications (i.e., BFS, MUM, SP, and SSP) experience severe performance loss in baseline configuration because of the contention in last level cache and/or congestion in the interconnection network. For instance, by allocating the ideal number of SMs to SSP, and BFS, their last level cache hit-rate improves by 45%, and 30%, respectively. Our technique improves the performance of those four applications by up to 25%, with an average of 12%. Our proposed mechanism reduces he performance of the remaining applications 2% on average.

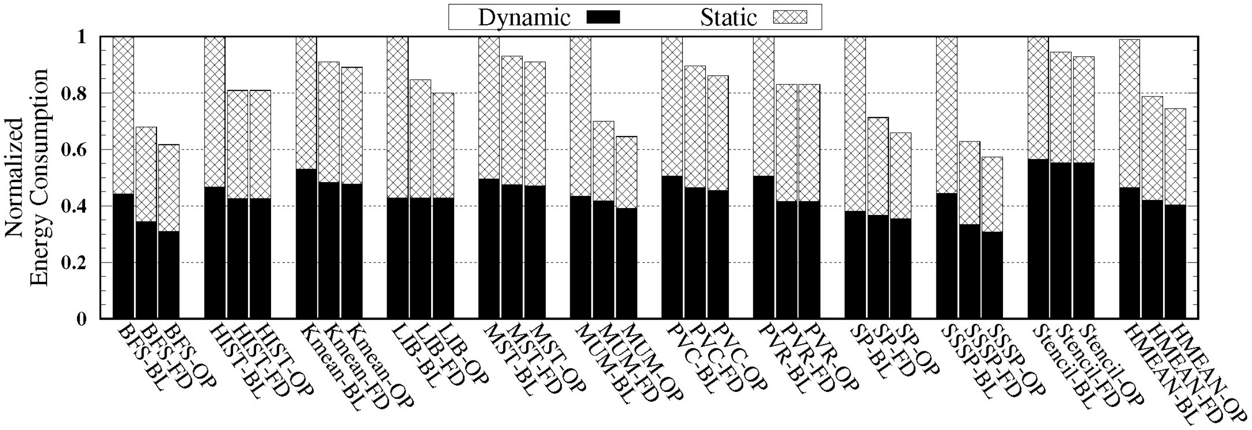


Figure 8: Energy saving gained by using optimal number of SMs. BL, FD, and OP represent BaseLine, our proposed Feedback-Driven, and OPtimal system configurations, receptively.

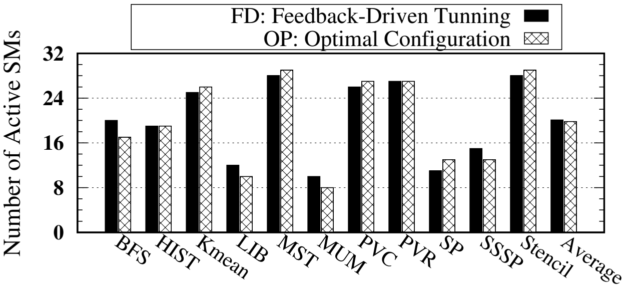


Figure 9: Average number of active SMs.

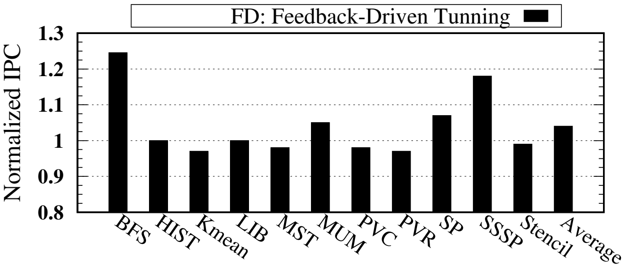


Figure 10: Normalized IPC values for different applications with respect to the baseline configuration.

Application of DVFS Techniques on Memory-Intensive Kernels: Figure 12 compares the impact of our proposed mechanism with a common DVFS scheme. We assumed that the GPGPU has 7 power-states as reported in Figure 11. A wide range of DVFS techniques are available to regulate the voltage/frequency of different resources in a computing platform. In our feedback-driven CTA scheduling approach, we adopt a SM power-gating mechanism to shut down some of the SMs. However, one can adopt of a DVFS technique to regulate voltage/frequency of the SMs in order to resolve the memory saturation problem (Leng et al. 2013). Here we discuss two major differences between power-gating and DVFS techniques: First, power-gating is more effective in reducing the leakage power compared to DVFS techniques, as reported in Figure 12. Second, although reducing the frequency of the SMs could mitigate the memory bandwidth saturation problem, it does not resolve the cache contention problem because cache contention is not a function of time but a function of the sequence of cache accesses. In other words, cache access pattern is a function of number of running threads which is modulated by our feedback-driven CTA scheduler but it is not affected by a core-side DVFS scheme. As can be seen in Figure 12, for BFS, MUM, SP, and SSSP that suffer from cache contention problem, our proposed technique considerably outperforms DVFS. Overall, our proposed technique reduces the energy consumption 21% on average while DVFS improvement is about 9% on average.

Voltage	Frequency
1.000 V	2.00 GHz
0.925 V	1.75 GHz
0.850 V	1.50 GHz
0.775 V	1.25 GHz
0.700 V	1.00 GHz
0.625 V	0.75 GHz
0.550 V	0.50 GHz

Figure 11: List of adopted V/F states to dynamically manage SM power consumption during the memory-intensive phases.

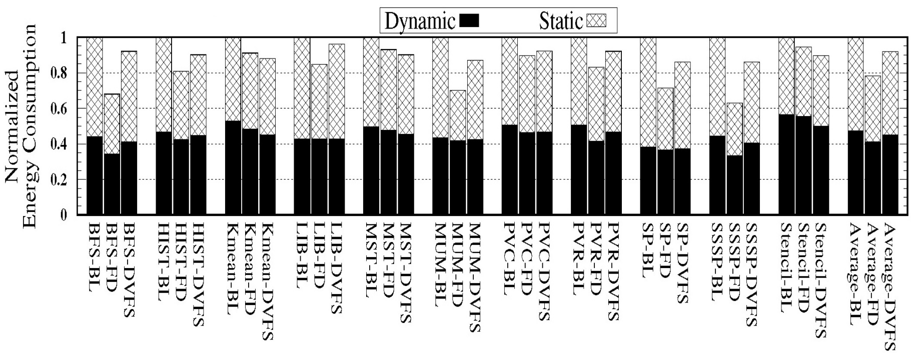


Figure 12: Energy saving gained by different techniques. BL, and FD represent BaseLine, our proposed Feedback-Driven configurations, receptively.

6 RELATED WORK

Theoretically speaking, assigning more SMs to a highly multi-threaded application improves its performance as long as the memory bandwidth does not saturate. Huang et al. (2009) evaluated the effect of number of active SMs on energy consumption and discussed that having all the SMs activated is the most energy efficient configuration. The lack of that study is that they did not consider any memory-intensive application. In order to have a more accurate analysis, we need to consider the possible congestion in the interconnection network and the contention in last level cache caused by enormous number of memory requests (issued by huge number of concurrently running threads). In this line, Guz et al. (2009) showed that increasing the parallelism improves the performance as long as the memory access latency is not affected considerably. An orthogonal category of works (Jadidi et al. 2011, Arjomand et al. 2011, Arjomand et al. 2016) exploit large last level caches and/or accelerate the memory accesses to reduce average data access latency. Li and Martínez (2005) analytically estimated the optimal number of processors to achieve the best EDP in CMPs. In GPU domain, Hong and Kim (2010) proposed an analytical model which predicts the optimal number of SMs based on offline characterizations. Our proposed mechanism however, exploits run-time characteristics for regulating number of active cores. As shown in Section 3, many kernels go into saturation state while MBU is much less than 100%. Therefore, although Hong and Kim (2010) statically provide us estimations for the number of active cores, it does not consider potential contention in the last level cache and/or congestion in the interconnection network. On the other hand, DVFS techniques (Leng et al. 2013) can be exploited to regulate the voltage/frequency of the SMs during memory-intensive phases. Although a core-side DVFS technique can improve the energy consumption of memory-intensive kernels, as discussed in Section 5, our proposed mechanism outperforms DVFS techniques in terms of reducing static leakage power as well as resolving the last level cache contention issue.

7 CONCLUSIONS

In this paper, we proposed a feedback-driven mechanism that dynamically adjusts the number of active SMs based on the kernel demand. The proposed mechanism uses a CTA-based sampling and reconfiguration scheme to dynamically analyse the kernel and determine the ideal number of active SMs. This technique reduces the chip energy consumption up to 35% and about 21% on average over the studied memory-intensive applications, which is within 8% of the optimal saving that can be obtained from an oracle scheme.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1302557, 1213052, 1439021, 1302225, 1629129, 1526750, and 1629915 and a grant from Intel.

REFERENCES

Arjomand, M., A. Jadidi, M. T. Kandemir, A. Sivasubramaniam, and C. Das. 2016, April. "MLC PCM main memory with accelerated read". In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 143–144.

Arjomand, M., A. Jadidi, A. Shafiee, and H. Sarbazi-Azad. 2011, Oct. "A morphable phase change memory architecture considering frequent zero values". In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, pp. 373–380.

Bakhoda, A., G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009, April. "Analyzing CUDA workloads using a detailed GPU simulator". In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174.

Burtscher, M., R. Nasre, and K. Pingali. 2012. "A quantitative study of irregular programs on GPUs". In *IISWC*.

Danalis, A., G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. 2010. "The Scalable Heterogeneous Computing (SHOC) benchmark suite". In *GPGPU*.

Fermi, N. 2009. "Nvidia's next generation cuda compute architecture".

Fung, W. W. L., I. Sham, G. Yuan, and T. M. Aamodt. 2007. "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow". In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pp. 407–420. Washington, DC, USA, IEEE Computer Society.

Guz, Z., E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. C. Weiser. 2009, January. "Many-Core vs. Many-Thread Machines: Stay Away From the Valley". *IEEE Comput. Archit. Lett.* vol. 8 (1), pp. 25–28.

He, B., W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. 2008. "Mars: A MapReduce Framework on Graphics Processors". In *PACT*.

Hong, S., and H. Kim. 2010, June. "An Integrated GPU Power and Performance Model". *SIGARCH Comput. Archit. News* vol. 38 (3), pp. 280–289.

Huang, S., S. Xiao, and W. Feng. 2009, May. "On the energy efficiency of graphics processing units for scientific computing". In *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–8.

Jadidi, A., M. Arjomand, and H. Sarbazi-Azad. 2011. "High-endurance and Performance-efficient Design of Hybrid Cache Architectures Through Adaptive Line Replacement". In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pp. 79–84. Piscataway, NJ, USA, IEEE Press.

Lee, J., V. Sathisha, M. Schulte, K. Compton, and N. S. Kim. 2011. "Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling". In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 111–120. Washington, DC, USA, IEEE Computer Society.

Leng, J., T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. 2013, June. "GPUWattch: Enabling Energy Optimizations in GPGPUs". *SIGARCH Comput. Archit. News* vol. 41 (3), pp. 487–498.

Li, J., and J. F. Martínez. 2005, December. "Power-performance Considerations of Parallel Computing on Chip Multiprocessors". *ACM Trans. Archit. Code Optim.* vol. 2 (4), pp. 397–422.

NVIDIA 2011. "CUDA C/C++ SDK Code Samples".

NVIDIA Corporation 2010. "NVIDIA CUDA C Programming Guide". Version 3.2.

Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. 2000, May. “Memory Access Scheduling”. Volume 28, pp. 128–138. New York, NY, USA, ACM.

Rogers, T. G., M. O’Connor, and T. M. Aamodt. 2012. “Cache-Conscious Wavefront Scheduling”. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 72–83. Washington, DC, USA, IEEE Computer Society.

Stratton, J. A., C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. 2012. “Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing”. Technical Report IMPACT-12-01.

AUTHOR BIOGRAPHIES

AMIN JADIDI received the B.Sc. degree from University of Tehran of Tehran, Iran, in 2009, and the M.Sc. degree from the Sharif University of Technology, Tehran, Iran, in 2011, all in electrical and computer engineering. He currently is a PhD candidate in Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA. His current research interests include multicore and manycore architectures, GPGPU architectures, and memory systems. His email address is axj945@cse.psu.edu.

MOHAMMAD ARJOMAND (S’09) received the B.Sc. degree from the Shahid Bahonar University of Kerman, Kerman, Iran, in 2006, and the M.Sc. and Ph.D. degrees from the Sharif University of Technology, Tehran, Iran, in 2008 and 2014, respectively, all in computer engineering. He currently holds a post-doctoral position with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA. His current research interests include multicore and manycore architectures, memory systems, storage systems, and power-aware architectures. Mr. Arjomand is a Student Member of the Association for Computing Machinery. His email address is mxa51@cse.psu.edu.

MAHMUT TAYLAN KANDEMIR is a professor in the Computer Science and Engineering Department at the Pennsylvania State University. He is a member of the Microsystems Design Lab. Dr. Kandemir’s research interests are in optimizing compilers, runtime systems, embedded systems, I/O and high performance storage, and power-aware computing. He is the author of more than 80 journal publications and over 300 conference/workshop papers in these areas. He is a recipient of NSF Career Award and the Penn State Engineering Society Outstanding Research Award. He currently serves as the Graduate Coordinator of the Computer Science and Engineering Department at Penn State. His email address is kandemir@cse.psu.edu.

CHITA R. DAS is a distinguished professor in the Department of Computer Science and Engineering at Pennsylvania State University. He currently serves as the Dean of the Computer Science Department at Penn State. His main areas of interest include parallel and distributed computer architectures, multi-core architectures, mobile computing, performance evaluation, and fault-tolerant computing. He has published more than 200 papers in the above areas, has served on many program committees, and editorial boards. He has a PhD in computer science from the University of Louisiana, Lafayette. He’s a fellow of IEEE. His email address is das@cse.psu.edu.

OPENFOAM ON GPUS USING AMGX

Thilina Rathnayake
University of Moratuwa
10400, Katubedda, Sri Lanka
thilinarmtb.10@cse.mrt.ac.lk

Sanath Jayasena
University of Moratuwa
10400, Katubedda, Sri Lanka
sanath@cse.mrt.ac.lk

Mahinsasa Narayana
University of Moratuwa
10400, Katubedda, Sri Lanka
mahinsasa@uom.lk

ABSTRACT

Field Operation and Manipulation (OpenFOAM) is a free, open-source, feature-rich Computational Fluid Dynamics (CFD) software that is used to solve a variety of problems in continuum mechanics. Depending on the type of problem and required accuracy, an OpenFOAM simulation may take several weeks to complete. For sufficiently large simulations, linear solvers consume a large portion of the execution time. AmgX is a state of the art, high performance library which provides an elegant way to accelerate linear solvers on GPUs. AmgX library provides multi-grid solvers, Krylov methods, smoothers, support for block systems and MPI. In this work, we implemented OpenFOAM solvers on GPUs using the AmgX library. We also created helper functions which enable seamless integration of these solvers with OpenFOAM. These functions will take care of converting the linear system to AmgX's format and apply the user specified configurations to solve it. Experiments carried out using a wind rotor simulation and a fan wing simulation show that the use of AmgX library gives upto 10% speedup in the total simulation time and around **2x** speedup in linear system solving portion within the simulation.

Keywords: GPU, OpenFOAM, AmgX, GPGPU, linear solvers

1 INTRODUCTION

Scientific computing has become a very important part of modern research in many science and engineering disciplines like fluid dynamics, acoustics, solid mechanics and electro-magnetics. Often scientists and engineers carry out computer simulations to model and better understand important phenomena like wind patterns around a plane or turbulence of a fluid. These computer simulations can save time, money and often gives more flexibility in trying out new things than actually carrying out them in the real world. Computational Fluid Dynamics (CFD) is an important sub-field in scientific computing. CFD is essentially a combination of fluid dynamics, numerical methods and computer science where computers are used to run numerical algorithms to solve a fluid dynamics problem. Using computers to solve fluid dynamics problems has become very popular due to the availability of high performance computers and well developed numerical methods.

Field Operation and Manipulation (OpenFOAM) (Jasak, Jemcov, Tukovic, et al. 2007) library is an open-source CFD software package widely used in both academia and industry. OpenFOAM is written in C++ and can be used to solve partial differential equations (PDEs). OpenFOAM can be used in all three phases of a simulation: pre-processing, solving and post-processing. It contains meshing tools like blockMesh for pre-processing and visualization software like ParaView for post-processing. OpenFOAM comes with built

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

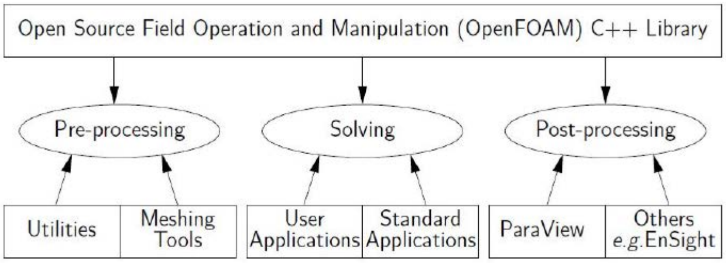


Figure 1: Overview of OpenFOAM structure (source: OpenFOAM Documentation).

in MPI functionality which allows users to decompose a given mesh into multiple chunks and use multiple computing nodes (or a single node with multiple cores) to process the chunks in parallel.

Computational scientists have been using Graphic Processing Units (GPUs) to speed up various computationally intensive tasks in scientific computing. GPUs have been widely used in CFD problems as well. GPUs have hundreds of cores that can run thousands of threads to perform vector operations over large data structures. This ability comes handy when solving large linear systems. There are many software libraries that enable usage of GPUs to run common computational kernels in numerical linear algebra. NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS) library, NVIDIA CUDA Sparse Matrix library (cuSPARSE) and NVIDIA AmgX library are some of the popular libraries with NVIDIA GPUs. In this paper we investigate how OpenFOAM can be combined with NVIDIA AmgX library to run CFD problems in GPUs.

The paper has the following structure. First we look at the structure of OpenFOAM in Section 2 and give a brief introduction to NVIDIA AmgX library in Section 3. Then we look at the related work in Section 4. We describe the methodology we followed in Section 5 and present experiments we carried out and their results in Section 6. Finally, we present the conclusions and recommendations in Section 7.

2 OPENFOAM

OpenFOAM is written in C++ and heavily uses object oriented features in C++ to build the framework required for simulations. The primary use of OpenFOAM is to create executables known as applications. These applications can be broadly categorized into *solvers* and *utilities*. Solvers are created to solve a specific problem in continuum mechanics like calculating pressure and velocities of an in-compressible flow flowing through a specific tube geometry. Utilities are designed to perform tasks that involve data manipulation. Users can create custom solvers and utilities by using OpenFOAM with some knowledge about underlying CFD algorithms, physics and programming techniques. OpenFOAM ships with pre and post processing tools. OpenFOAM utilities have been written on top of these tools to enable users to easily access them. Thus, the interface to these pre and post processing tools are consistent even though underlying tool environments can change. The overall structure of OpenFOAM is shown in Figure 1.

OpenFOAM has several methods (algorithms) to solve the linear system resulting after the discretization of the computational domain and the differential equation. Algorithm selection depends on the resulting linear system (symmetric, asymmetric), initial and boundary conditions and the convergence characteristics of the matrix. Table 1 shows the solvers available in OpenFOAM.

Different types of preconditioners and smoothers are used in OpenFOAM to solve linear systems more efficiently. Preconditioners transform the linear system so that the transformed system converges much faster than the original. Figure 2 shows the structure of linear solvers available in OpenFOAM along with preconditioners and smoothers.

Table 1: OpenFOAM linear solvers.

BICCG	Diagonal incomplete LU preconditioned BiCG solver
diagonalSolver	diagonal solver for both symmetric and asymmetric problems
GAMG	Geometric agglomerated algebraic multi-grid solver (also named Generalized geometric- algebraic multi-grid)
ICC	Incomplete Cholesky preconditioned Conjugate Gradients solver
PBiCG	Preconditioned bi-conjugate gradient solver for asymmetric lduMatrices using a run-time selectable preconditioner
PCG	Preconditioned conjugate gradient solver for symmetric lduMatrices using a run-time selectable preconditioner
smoothSolver	Iterative solver using smoother for symmetric and asymmetric matrices which uses a run-time selected smoother

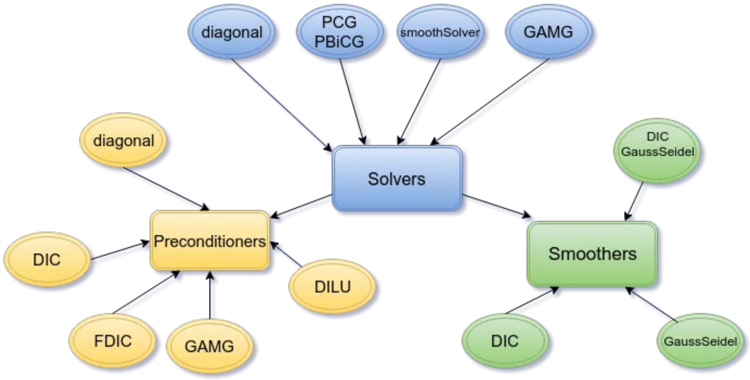


Figure 2: OpenFOAM's complete linear solver structure.

3 NVIDIA AMGX

AmgX provides a simple way to access accelerated solvers on NVIDIA GPUs (Naumov, Arsaev, Castonguay, Cohen, Demouth, Eaton, Layton, Markovskiy, Sakharnykh, Strzodka, et al. 2014). NVIDIA claims that AmgX can provide up to 10x acceleration of the computationally intense linear solver portion of simulations (AmgX 2016). One of the main advantages of using AmgX is its flexible solver composition system which allows a user to easily combine various solvers and preconditioners. AmgX has a simple C API that abstracts the parallelism and GPU implementation. Main features of the AmgX library include, according to NVIDIA (AmgX 2016):

- Flexible configuration allows for nested solvers, smoothers, and preconditioners.
- Krylov methods: PCG, GMRES, BiCGStab, and flexible variants.
- Smoothers: Block-Jacobi, Gauss-Seidel, incomplete LU, Polynomial, dense LU.
- MPI and OpenMP support.

4 RELATED WORK

In this section, we are first going to look at some of the existing literature on research conducted using OpenFOAM with GPUs and then do a survey about existing software libraries for porting OpenFOAM simulations to GPUs.

4.1 Related Research

Amaniz AlOnazi et al. (AlOnazi 2014) have tried to design and optimize OpenFOAM based CFD applications to hybrid heterogeneous HPC platforms. Although OpenFOAM supports MPI natively, according to authors, it doesn't scale well for heterogeneous systems (AlOnazi, Keyes, Lastovetsky, and Rychkov 2015). Authors have extensively studied Conjugate Gradient (CG) method and identified the bottlenecks when it is run in a distributed memory system using MPI.

Qingyun He et al. (He, Chen, and Feng 2015) have used a wide variety of existing libraries to speed up OpenFOAM solvers. Authros have implemented the magnetohydrodynamics (MHD) solver on Kepler-class GPUs using the CUDA technology. Authors claim that a GPU (GTX 770) can outperform a server-class 4-core, 8-thread CPU (Intel Core i7-4770k). They have used the following libraries for CFD acceleration of the MHD solver:

1. CUDA for OpenFOAM Link (Cufflink) (Combest and Day 2011), an open source library for GPU acceleration in OpenFOAM. It supports single and double precision.
2. SpeedIT Plugin (Vratis 2013) to OpenFOAM by Vratis released for demonstration purposes for GPU acceleration. The free version supports only single precision.
3. GPU linear solvers library for OpenFOAM (ofgpu) (Symscape 2014) by Symscape under GPL license. It supports only single precision.

Authors accelerated the MHD solver by replacing its linear system solvers since solving matrices occupy most of program running time. Also, authors replaced sparse matrix vector product (SMVP) kernels with the corresponding GPU implementations. The vector–vector scalar product is calculated using the NVIDIA CUBLAS (Nvidia 2008) library. Authors claim that they were able to get a 4x speedup for the benchmarks using a single GPU.

Jamshidi and Khunjush (Jamshidi and Khunjush 2011) have used the CUSPARSE (CUSPARSE 2017) and CUBLAS (Nvidia 2008) libraries to implement some of the OpenFOAM solvers. Author's have identified that the main computational intensive step in OpenFOAM solvers is the solving systems of linear equations. They have tested their implementations in three different multi-core platforms including the Cell Broadband Engine, NVIDIA GPU, and an Intel quad-core Xeon CPU. According to their results, the GPU implementations achieve the best performances.

4.2 Existing Software for running OpenFOAM on GPUs

Apart from the above research work, there are a few software solutions which attempt to port OpenFOAM simulation to GPUs.

4.2.1 Paralution

Paralution enables users to use multi/many-core CPU and GPU devices for various numerical linear algebra kernels (Lukarski and Trost 2014). It supports various back-ends like OpenMP, CUDA and OpenCL. Paralution is released under dual license scheme with a open source GPLv3 license and a commercial license. Free license doesn't support MPI functionality.

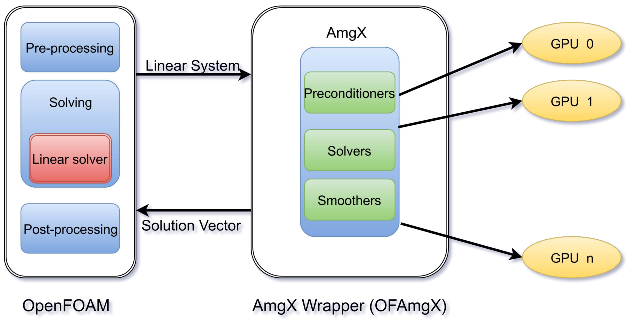


Figure 3: OFAmgX Library.

4.2.2 RapidCFD

RapidCFD is different to other libraries in that it uses GPUs for performing most of OpenFOAM's functionality, not only the linear solvers. RapidCFD avoids copying data during calculations between CPU and GPU as much as possible. Most of the data structures are created on the GPU itself. Operations on these are then done by using thrust (Hoberock and Bell 2010) library.

5 METHODOLOGY

In order to speed up the simulations, first thing we need to do is identifying the performance bottlenecks. We ran a simulation of a wind turbine using *callgrind* tool which ships with *valgrind* (Nethercote and Seward 2007) to identify the computationally intensive sections of the program.

According to profiling results, *Foam::fvMatrix::solve* method in OpenFOAM has the highest accumulated overhead (discarding the *main* method). Nearly $1/3$ of the simulation time is spent on this single method. In OpenFOAM, *Foam::fvMatrix* is the class that holds the matrix resulting from the finite volume discretization. *solve* method is the member which solves the linear system associated with this matrix. So, it can be concluded that a large portion of the simulation time is spent solving the linear system.

This fact becomes more obvious when we look at the overhead introduced by methods ignoring the overhead of the callees. Four out of the top 5 hot-spots are methods used in solving the linear system. It is evident that we can maximize our speed up by making linear solvers run faster i.e., linear solvers are the perfect candidates to be implemented in GPU.

We used the NVIDIA's AmgX library described in Section 3 to solve the OpenFOAM's linear system in the GPU. We implemented a wrapper library, **OFAmgX** which enables the easy use of AmgX's linear solvers from OpenFOAM. Figure 3 shows the overall structure and interaction of OFAmgX with OpenFOAM. We found the work done by Chuang and Barba to add AmgX support to *PETSc* (Pi-Yueh Chuang 2016) really helpful when writing our library. We especially found their load balancing methodology which is described in Section 5.2 to be very useful.

As shown in the Figure 3, pre-processing and post-processing takes place in the CPU using normal OpenFOAM utilities. But during the solving process, linear system (matrix and the right hand side vector) is copied to the GPU by the AmgX wrapper library and the AmgX solvers are invoked on the system. After the AmgX library is done with solving the system, results are copied back to the CPU by the OFAmgX.

User can specify the solution algorithm, tolerance values, pre-conditioners and smoothers in the OpenFOAM side as in a normal OpenFOAM simulation. OFAmgX will read those values and setup the solver in the GPU according to that information.

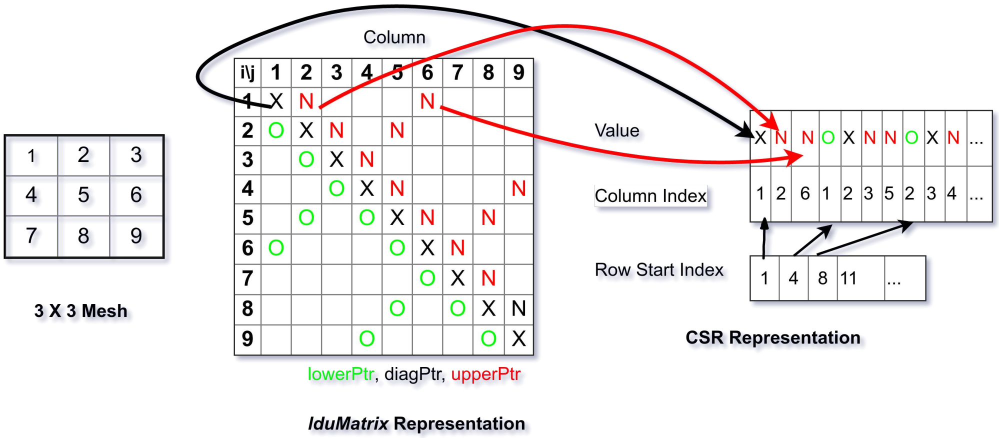


Figure 4: Conversion of lduMatrix to CSR Format.

5.1 Data Structure Conversion

Before solving the linear system, OFAmgX has to convert the matrix and the right hand vector to a format consumable by the AmgX library. After solving the system, OFAmgX has to convert the solution vector back to a OpenFOAM vector. Conversion between the vectors is pretty straight forward. But the conversion between the matrices require additional work.

OpenFOAM stores its matrices in the *lduMatrix* format (CFD-online) (OpenFOAM-Wiki 2016). In the *lduMatrix* format, lower, diagonal and upper elements of the matrix are stored separately in different arrays. Addressing for these elements are stored in another two arrays which stores the row and column index of each element. This storage method is extremely efficient for storing the matrices resulting from finite volume discretization.

AmgX solver library only accepts matrix in Compressed Sparse Row (CSR) format (Saad 2003). In CSR format, only the nonzero elements in the matrix are stored. All the nonzero elements of the matrix are stored row wise in one array (*value* in Figure 4) and the column indices of these elements are stored in another array (*column index* in Figure 4). A third array keeps track of the index of the start element in each row. To use AmgX with OpenFOAM, we need to convert *lduMatrix* format to the CSR format and this is done by OFAmgX. Figure 4 shows visually how the conversion can be done from *lduMatrix* to *CSR*.

5.2 MPI Support

OFAmgX supports using multiple GPUs in a cluster and/or node to solve the linear system with MPI. The major issue in enabling multiple GPU support is the mapping of MPI processes to the GPUs. Suppose we have a GPU cluster like Figure 5 with three nodes. First, the global communicator is split into in-node communicators which are local to each node. Figure 6 shows the cluster after this initial split.

Next, each in node (or local) communicator is divided depending on the number of GPUs available at the node and the number of MPI processes started at the node. Figure 7 shows the communicators local to each GPU device after this split. Usually, in OpenFOAM, the number of MPI processes equal to the number of cores in the node. We want to make sure that each GPU device has almost equal loads.

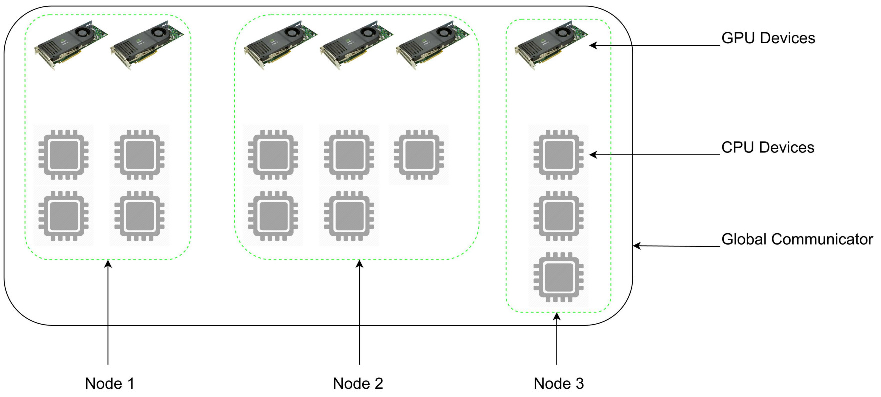


Figure 5: GPU Cluster with three nodes.

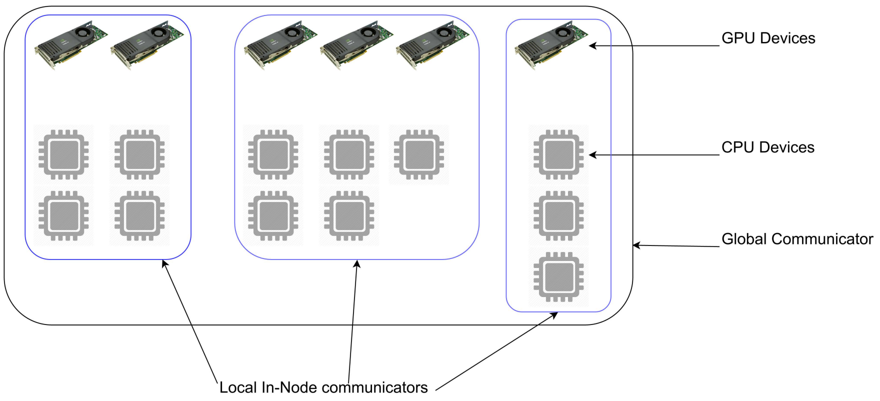


Figure 6: Global communicator split into local in-node communicators.

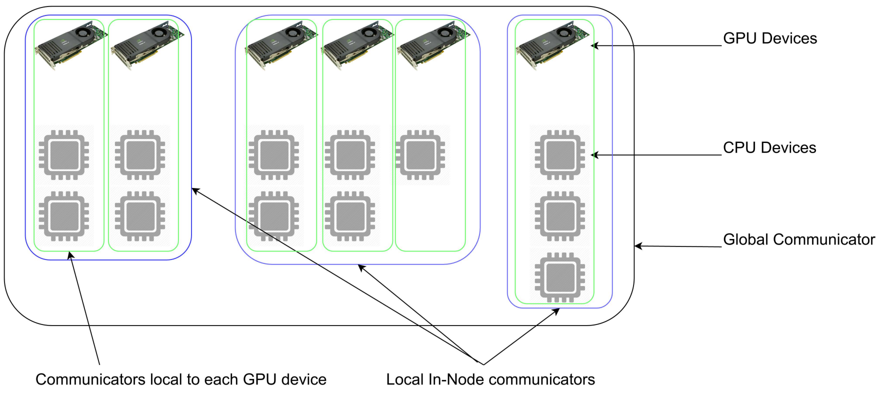

GPU Devices
CPU Devices
Global Communicator
Communicators local to each GPU device
Local In-Node communicators

Figure 7: Local in-node communicators split by device.

Suppose we have n GPU devices and m MPI processes in a given node. If m is divisible by n, then each GPU device will get $\frac{m}{n}$ MPI processes. This is the case with the leftmost and the rightmost nodes in Figure 7. Suppose m is not divisible by n and leaves a remainder r after division. Then $m-r$ is divisible by n. In this case, r devices will get $\frac{m-r}{n}+1$ MPI processes each and the rest $n-r$ devices get $\frac{m-r}{n}$ MPI processes each. This is the case with the middle node in Figure 7.

Table 2: Different Experimental Environments.

Environment	Details
A	Intel Xeon @ 2.0GHz x 32, 64 GB RAM with a Tesla C2070 GPU
B	Intel core i7 @ 3.40GHz X 4, 16 GB RAM with 2 GTX 480 GPUs
C	Intel core i7 @ 1.6GHz X 4, 8 GB RAM with 2 GTX 480 GPUs

5.3 Multiple Solver Support

In a general OpenFOAM simulations, different types of solvers may be used to solve for different physical quantities. For example, the solver used for calculating pressure in the mesh points may not be used for calculating the speed at the mesh points. So, it is essential that our wrapper supports multiple types of solvers to be used in the same simulation.

When transforming solver configurations from OpenFOAM to AmgX, different solvers used in OpenFOAM have different configurations in AmgX as well. So, the configuration string used by OFAmgX to initialize solvers in the GPU are unique. We can use this configuration string to keep track of the data structures in AmgX which are used to run a particular solver.

For a solver to be run by AmgX, a resource handle and a solver handle must be created. To support multiple solvers, we stored resource handlers of the solvers in one dictionary and solver handlers in another dictionary. During the simulation, we can select appropriate solver and resource handles depending on the configuration string and run the required solver on the GPU. Another advantage of this method is that we only need to initialize the AmgX library once. After initialization is complete, any solver can be run in the GPU. After the simulation is over, we can finalize the AmgX library. This step needs to be done only once as well. This saves a lot of time, especially in smaller simulations.

6 EXPERIMENTAL RESULTS

We benchmarked native OpenFOAM, OFAmgX library and Paralution library (Section 4.2.1) library under three different hardware environments. These environments are listed in Table 2. We used a server (A) and two desktop machines (B and C). Each machine is equipped with GPUs and two desktop machines have two GPUs each. All the benchmarks were run using Ubuntu 14.04 and CUDA 6.5. OpenFOAM version 2.4 development version was used in the benchmarks and the latest RapidCFD master from its git repository was used. All the measurements are in seconds and they report the complete simulation time including data structure conversion and memory transfer from and back to CPU.

We were unable to setup RapidCFD (Section 4.2.2) successfully in our experimental environments. This may be due to the fact that it does not support GPUs with sm_20 architecture.

We used two different simulations under each environment to benchmark the libraries. We used a simulation of a wind turbine (referred to as windLM here onwards) as our first benchmark. This wind turbine simulation has a very intricate design and has around 1,166,000 mesh points. The second simulation simulates a wing of a fan (referred to as FanWing2D here onwards) and it is relatively smaller than the former. Table 3 provides a summary of the simulations used.

We measured the execution time each simulation took under different experimental setups. We used a 95% confidence interval for the measurements. For the windLM simulation, experiments were carried out with and without MPI. Table 4 lists the execution time for windLM simulation for the three environments without using MPI. Tables 5 – 7 list the execution time for the windLM with MPI using 2, 4 and 8 MPI processes. For the experiments with MPI, multiple MPI processes are created on a single node, not on a cluster.

Table 3: Different Simulations used in the benchmarks

Simulation	**Details**
windLM	Simulation of a wind turbine with 1,166,000 mesh-points
FanWing2D	Simulation of a 2D Fan with 60,000 mesh-points

Table 4: Total simulation time for windLM without MPI (in seconds)

	OpenFOAM	**OFAmgX**	**Paralution**
A	648.93	634.07	971.87
B	313.18	296.06	288.98
C	463.46	434.15	428.73

Table 5: Total simulation time for windLM with 2 MPI processes (in seconds)

	OpenFOAM	**OFAmgX**	**Paralution**
A	361.30	350.00	Note I
B	216.80	201.15	Note I
C	404.86	365.91	Note I

Table 6: Total simulation time for windLM with 4 MPI processes (in seconds)

	OpenFOAM	**OFAmgX**	**Paralution**
A	144.33	152.33	Note I
B	154.33	141.67	Note I

Table 7: Total simulation time for windLM with 8 MPI processes (in seconds)

	OpenFOAM	**OFAmgX**	**Paralution**
A	102.67	113.67	Note II
B	154	144.33	Note II

Table 8: Total simulation time for FanWing2D without MPI (in seconds)

	OpenFOAM	**OFAmgX**	**Paralution**
A	67.61	79.51	32.71
B	29.18	26.85	31.52
C	50.44	45.24	51.76

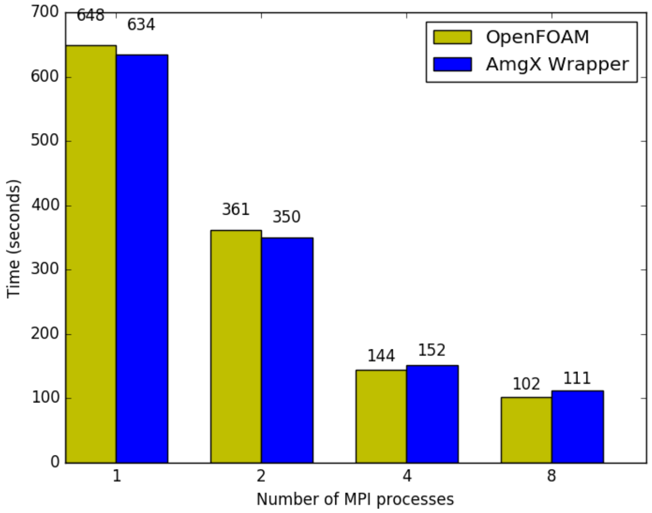


Figure 8: Performance of windLM simulation in environment A

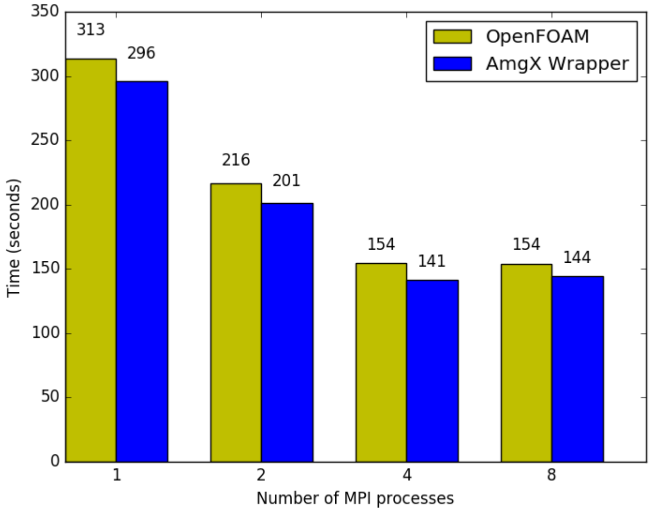


Figure 9: Performance of windLM simulation in environment B

For the FanWing2D simulation, experiments are carried out without MPI. Table 8 lists the execution time for FanWing2D simulation for the three environments without using MPI.

Notes:

I Paralution free version does not support MPI

Figures 8 and 9 summarizes the results of windLM simulation for the environments A and B respectively. In A, AmgX wrapper gets slower than OpenFOAM as the number of MPI processes increase. This is due to the fact that it has a single GPU and all the MPI processes start competing to use this GPU. In B, OFAmgX is always faster but the performance for 8 MPI processes is slower than for 4 MPI processes. This is due to the same reasons which caused the performance drop in A.

If we consider the time taken to solve the linear system instead of the total simulation time, OFAmgX gives around **2x** speedup depending on the solver. For example, Preconditioned Biconjugate Gradient (PBiCG) solver with Diagonal Incomplete LU preconditioner takes 441,383 microseconds to complete in OpenFOAM. In OFAmgX, this only takes 213,115 microseconds to complete.

7 CONCLUSIONS AND RECOMMENDATIONS

According to Section 6, in most of the experimental cases we got closer to a 8% speedup in the total simulation time by using OFAmgX compared to native OpenFOAM. In some cases it is even faster than Paralution library. Moreover, our OFAmgX supports domain decomposition using MPI and the free version of the Paralution library does not have MPI support.

OpenFOAM users can get a significant speed up by running their linear solvers in GPUs using OFAmgX library. Since we are using AmgX library, users get access to a lot of AmgX features which are not available in original OpenFOAM.

There are some areas in our wrapper library which can be improved further. Although we added support for multiple solvers to be used in the same solution, we could not get significant improvements by using multiple solvers in the same simulation. Also, we could not beat OpenFOAM's multi-grid solver (GAMG) with the multi-grid solver available in AmgX library. Improving these may involve finding the best parameter values that need to be which gives optimal performance in the algorithms implemented with AmgX.

Currently OFAmgX doesn't support block matrices. If the mesh is large enough, adding support for the block matrices can be beneficial as well. AmgX library supports block matrices and we need to investigate how this feature can be used with OpenFOAM to improve the performance.

One limitation of the OFAmgX is the overhead in converting the lduMatrix format to the CSR matrix format. This conversion needs to be done in order to feed the matrix of the linear system to the AmgX solver. This conversion takes time and is unavoidable. Also, transferring data from/to GPU is also time consuming and a limitation of the wrapper.

According to the experiments carried out, performance tend to degrade when more than four MPI processes are mapped into a single GPU. This is due to the fact that some of the MPI processes have to wait in idle till the GPU becomes available for their use. Investigating how to fix this issue is one of the challenges.

ACKNOWLEDGMENT

We would like to show our gratitude to the senate research committee of the University of Moratuwa and the LK Domain registry for supporting this research.

REFERENCES

AlOnazi, A., D. Keyes, A. Lastovetsky, and V. Rychkov. 2015. "Design and optimization of openfoam-based cfd applications for hybrid and heterogeneous hpc platforms". *arXiv preprint arXiv:1505.07630*.

AlOnazi, A. A. 2014. *Design and optimization of openfoam-based CFD applications for modern hybrid and heterogeneous HPC platforms*. Ph. D. thesis.

NVIDIA AmgX 2016. "AmgX Documentation". https://developer.nvidia.com/amgx. Online: accessed July-2016.

CFD-online. “lduMatrix”.

Combest, DP and Day, J 2011. “Cufflink: a library for linking numerical methods based on CUDA C/C++ with OpenFOAM”. http://cufflink-library.googlecode.com. Online: accessed July-2016.

CUSPARSE, NVIDIA 2017. “CUBLAS libraries”. https://developer.nvidia.com/cusparse.

He, Q., H. Chen, and J. Feng. 2015. “Acceleration of the OpenFOAM-based MHD solver using graphics processing units”. *Fusion Engineering and Design* vol. 101, pp. 88–93.

Hoberock, Jared and Bell, Nathan 2010. “Thrust: A parallel template library”. http://thrust.googlecode.com. Online: accessed July-2016.

Jamshidi, Z., and F. Khunjush. 2011. “Optimization of OpenFOAM’s linear solvers on emerging multi-core platforms”. In *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, pp. 824–829. IEEE.

Jasak, H., A. Jemcov, Z. Tukovic et al. 2007. “OpenFOAM: A C++ library for complex physics simulations”. In *International workshop on coupled methods in numerical dynamics*, Volume 1000, pp. 1–20. IUC Dubrovnik, Croatia.

Lukarski, Dimitar and Trost, Nico 2014. “PARALUTION project”. http://www.paralution.com. Online: accessed July-2016.

Naumov, M., M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, N. Sakharnykh, R. Strzodka et al. 2014. “AmgX: Scalability and performance on massively parallel platforms”. In *SIAM workshop on exascale applied mathematics challenges and opportunities. SIAM*.

Nethercote, N., and J. Seward. 2007. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In *ACM Sigplan notices*, Volume 42, pp. 89–100. ACM.

Nvidia, C. 2008. “Cublas library”. *NVIDIA Corporation, Santa Clara, California* vol. 15, pp. 27.

OpenFOAM-Wiki 2016. “Matrices in OpenFOAM”. https://openfoamwiki.net/index.php/OpenFOAM_guide/Matrices_in_OpenFOAM. Online: accessed July-2016.

Pi-Yueh Chuang, L. A. B. 2016. “Using AmgX to Accelerate PETSc-Based CFD Codes”. *GPU Technology Conference*.

Saad, Y. 2003. *Iterative Methods for Sparse Linear Systems, 2nd edition*. Philadelpha, PA, SIAM.

Symscape 2014. “ofgpu: GPU Linear Solvers for OpenFOAM”. http://www.symscape.com/gpu-1-1-openfoam. Online: accessed July-2016.

Vratis 2013. “SpeedIT Plugin for OpenFOAM”. http://speedit.vratis.com/index.php/products. Online: accessed July-2016.

AUTHOR BIOGRAPHIES

THILINA RATHNAYAKE earned a BSc Engineering in Computer Science and Engineering from the University of Moratuwa. His research interests include parallel and high performance computing and scientific computing.

SANATH JAYASENA is an Associate Professor in the Dept. of Computer Science and Engineering in the Faculty of Engineering at the University of Moratuwa, Sri Lanka. His research interests include parallel and high-performance computing, machine learning and local language computing.

MAHINSASA NARAYANA is a Senior Lecturer in the Dept of Chemical and Process Engineering in the Faculty of Engineering at the University of Moratuwa, Sri Lanka. His research interests are in the areas of renewable energy, process control, modeling and simulation.

PIVOTING STRATEGY FOR FAST LU DECOMPOSITION OF SPARSE BLOCK MATRICES

Lukas Polok
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 1/2, Brno 61266, Czech Republic
ipolok@fit.vutbr.cz

Pavel Smrz
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Bozetechova 1/2, Brno 61266, Czech Republic
smrz@fit.vutbr.cz

ABSTRACT

Solving large linear systems is a fundamental task in many interesting problems, including finite element methods (FEM) or (non-)linear least squares (NLS) for inference in graphical models such as simultaneous localization and mapping (SLAM) in robotics or bundle adjustment (BA) in computer vision. Furthermore, the problems of interest here are sparse. The most time-consuming parts are sparse matrix assembly and linear solving. An interesting property of these problems is their block structure. The variables exist in multi-dimensional space such as 2D, 3D or $\mathfrak{se}(3)$ and hence their respective derivatives are dense blocks. In our previous work (Polok et al. 2013), we demonstrated the benefits of explicitly representing blocks in sparse matrices, namely faster matrix assembly and arithmetic operations. Here, we propose and evaluate a novel sparse block LU decomposition. Our algorithm is on average 3× faster (best case 50×), causes less fill-in and has comparable or often better precision than the conventional methods.

Keywords: LU decomposition, sparse matrix, block matrix, register blocking, direct methods.

1 INTRODUCTION

Solving a linear system of the form $Ax = b$ is not trivial, unless A has specific properties such as being a triangular matrix. Then, the last element of the unknown vector x is given by a ratio of the corresponding elements in A and b. Next, the second last element can be calculated by substituting the first element and solving another simple ratio and so on, until all of x is recovered–a process called *back-substitution*. There are several algorithms for bringing a matrix into triangular form, Gaussian elimination, LU decomposition, Cholesky factorization and QR decomposition (Davis 2006). These algorithms decompose the original matrix A into a product of two or more factor matrices that are triangular or otherwise easily invertible (diagonal or orthogonal). The solution to the original linear system then becomes a sequence of solutions for each of those factors, ultimately yielding x. In the case of LU decomposition, this amounts to first solving $Ly = b$ to get an intermediate vector y, and then solving $Ux = y$.

Notably, Cholesky factorization has the lowest time complexity of all these algorithms. It decomposes $A = R^\top R$ where R is upper-triangular matrix identical to that in QR decomposition, up to the sign (Cholesky will always yield R with positive diagonal entries). It is required that A be square, positive-definite (SPD) matrix. Matrices involved in least squares problems are SPD, which makes Cholesky an attractive choice for this class of applications. The disadvantage of using Cholesky for over-determined systems is the need to form a part of the Moore-Penrose pseudoinverse, $A^+ = (A^\top A)^{-1} A^\top$ that typically results in solving a system in the form $(A^\top A)x = A^\top b$. The formation of the square matrix $A^\top A$ may in some cases increase the amount

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

of memory needed and does increase the condition number, making the system more difficult to solve from numerical precision point of view.

If the matrix A is sparse, there is another interesting facet to the Cholesky decomposition, the *fill-in*. Ideally, R would have the same sparsity pattern as the upper triangle of A. However, in the course of calculating the factorization, other non-zero entries may be introduced in R and those are referred to as the fill-in. The number of nonzeros in R directly affects the speed of the factorization and subsequent backsubstitution. The amount of fill-in is dependent on the ordering of matrix rows and columns and fill-reducing orderings have been proposed in the literature, most notably the minimum degree ordering (George and Liu 1989, Liu 1985) and the faster approximate minimum degree (AMD) ordering (Amestoy, Davis, and Duff 1996).

The older LU decomposition is more general than Cholesky, and can factorize any square, invertible matrix into a product $A = LU$, where L is a lower-triangular matrix with unit diagonal and U is a general upper triangular matrix. This means the matrix no longer needs to be SPD. Since the diagonal of L is always the same, it does not need to be stored and sometimes L and U are stored together in a single matrix. Similar to Cholesky, LU decomposition also introduces fill-in and it is also possible to use the AMD algorithm for finding a fill-reducing ordering.

Unlike Cholesky factorization, LU is not inherently numerically stable and requires *pivoting*. A pivot is an element of the diagonal of A, that will serve as a divisor for other values in the decomposition algorithm. The magnitude of the pivot is of great importance if the numerical precision is finite. Using a small pivot will amplify the values in the matrix and lead to roundoff errors. The pivoting schemes therefore choose the largest pivot, either from the current column (referred to as *partial* pivoting) or from the so far unreduced remainder of the matrix, *full* pivoting.

While these strategies improve numerical stability, they also cause row or row and column reordering. This subsequently interferes with the fill-reducing ordering of the matrix and may inadvertently increase fill-in to unacceptable levels. In this paper, we show that block-based pivoting helps to reduce this effect, while at the same time not destroying the precision of the result. This strategy is a proof of concept and shows that block pivoting is possible and applicable to all kinds of decompositions that require it, be it LU, QR or LDL$^\top$ (a different form of Cholesky factorization that can work with symmetric indefinite matrices but which requires pivoting). We chose the LU factorization to demonstrate it because it is relatively simple to implement and has a potentially greater practical impact than LDL$^\top$.

The remainder of this paper is structured as follows. The next section summarizes related work, especially from the point of view of related sparse block matrix research and of pivoting strategies that were proposed in the literature. Section 3 briefly revisits principles of LU decomposition and common forms of algorithms for performing it. Section 4 describes the design of the proposed algorithm and how it differs from the standard LU decomposition methods implemented in packages such as CSparse (Davis 2006). Finally, Section 5 describes the evaluation of the proposed method.

2 RELATED WORK

While the formats for representing sparse block matrices date back to the early basic linear algebra subprograms (BLAS) proposal (Du and Marrone 1992), there are surprisingly only a few implementations that support them. The most popular sparse matrix package, CSparse (Davis 2006), a part of SuiteSparse only supports element-wise sparse matrices in compressed sparse column (CSC) format. Despite that, it is also being extensively used in applications where block matrices occur.

E.g. Google's Ceres solver (Agarwal and Mierle 2012), a NLS implementation behind popular products such as 3D maps or Street View uses CSparse for linear solving (other choices are also available, though, by using a dense solver or an iterative one). It also defines its own block storage format that serves to accelerate the sparse matrix assembly, but this format is abandoned in favor of CSC as soon as arithmetic operations on the matrix are required.

NIST Sparse BLAS (Carney, Heroux, Li, and Wu 1994) supports matrices in the compressed sparse row (CSR) format and in addition also the block compressed sparse row (BSR), a format for block matrices where all the blocks in a single matrix are the same size, and variable block compressed sparse row (VBR), a general format for block matrices where a single matrix can contain blocks of different dimensions. Unfortunately, there are no algorithms for matrix decompositions in NIST Sparse BLAS, nor is there an associated package that would contain them. Also, the triangular solving options are limited–only matrices with unit diagonal are supported.

There are more libraries that support the BSR format, most notably Intel MKL (F. 2009) or PETsc (Balay et al. 2015). Those can be readily used for solving linear systems, although limited to matrices containing only square blocks of a single dimension. This effectively limits their use to simpler problems with only variables of a single type (multiple variable types would in most cases yield blocks of several different dimensions and thus also rectangular blocks).

In our previous work (Polok, Ila, and Smrž 2013), we proposed an efficient block matrix storage format and several algorithms for performing arithmetic operations. The results were compared to CSparse and Ceres, proving the proposed implementation superior. We furthermore demonstrated the ability to outperform other block matrix implementations used in robotics (Polok, Šolony, Ila, Zemčík, and Smrž 2013). Later on, an implementation of sparse block Cholesky was added and its variant for incremental solving was also proposed (Polok, Ila, Šolony, Smrž, and Zemčík 2013).

Blocking is a popular technique for attaining higher memory throughput in dense implementations, used e.g. in LAPACK (Anderson et al. 1987). In Eigen (Guennebaud, Jacob, et al. 2010), the partially pivoted LU decomposition is blocked, splitting the matrix to rectangular blocks. Each such block contains a part of the diagonal and all the elements under it, so that the blocking would not interfere with pivot choice. After decomposing this block, the changes are communicated to the lower-right submatrix in a blockwise manner.

Ultimately, the choice of pivoting algorithm can have a great impact on the performance, due to required communication and memory access patterns. There were many pivoting algorithms proposed in the literature. MA21 (Duff 1981a, Duff 1981b) is one of the early examples, producing such a row permutation that the matrix ends up having nonzero diagonal entries. While not the best pivoting strategy for sparse decompositions, it showed potential in improving iterative solver convergence. The latter work (Duff and Koster 1999) expanded into obtaining such an ordering that the magnitude of the diagonal entries is maximized. It explores maximum product of the diagonal entries, which is the pivoting strategy applied in this paper.

Parallel implementations of LU decomposition often try to avoid pivoting during the decomposition phase itself, often by using threshold pivoting (a pivot permutation only takes place if its magnitude is larger by a given threshold than that of the natural pivot), or by performing static pivoting beforehand (Li and Demmel 2003). This helps to reduce communication and synchronization otherwise required.

Schenk and Gärtner (2006) propose a pivoting strategy for the $\mathrm{LDL}^\top$ factorization, not entirely unlike the method proposed here. Their algorithm chooses pivots of size 1×1 or 2×2 that are factorized in blockwise fashion (and in the case of 2×2, the blocks themselves are also subject to intra-block pivoting that the authors refer to as *perturbation*). In this work, we use pivoting at the granularity of the naturally occurring blocks, rather than choosing the size of the pivots.

Algorithm 1 Two Dense LU Decomposition Algorithms.

```
Require: That $A$ is an invertible $n \times n$ matrix, $P$ is $n \times n$ identity matrix.
function PIVOTING($A$, $P$, $k$)
    $p = \text{CHOOSEPIVOT}(A_{k:end,k})$          ▷ Choose a pivot from the lower portion of column $k$.
    if $p \neq k$ then
        SWAP($A_{k,*}$, $A_{p,*}$)                 ▷ Swap the pivotal row with the next unreduced row in $A$.
        SWAP($P_{k,*}$, $P_{p,*}$)                 ▷ Swap the same rows in the permutation matrix $P$.
    end if
    return $(A, P)$
end function

function SUBMATRIXLU($\Lambda$)
    for $k = 0$ to $n - 1$ do                      ▷ For each column in $A$.
        ($A$, $P$) = PIVOTING($A$, $P$, $k$)
        for $i = k + 1$ to $n - 1$ do
            $A_{i,k} = A_{i,k} / A_{k,k}$          ▷ Divide by the chosen pivot.
        end for
        for $j = k + 1$ to $n - 1$ do              ▷ Right-looking, exclusive.
            for $l = k + 1$ to $n - 1$ do
                $A_{l,j} = A_{l,j} - A_{l,k} \cdot A_{k,j}$   ▷ Scatter contributions to the so far unreduced submatrix.
            end for
        end for
    end for
end function

function COLUMNLU($\Lambda$)
    for $k = 0$ to $n - 1$ do                      ▷ For each column in $A$.
        for $j = 0$ to $k - 1$ do                  ▷ For all elements strictly above the pivot.
            $A_{j,k} = A_{j,k} / A_{j,j}$          ▷ Divide U by the past pivots.
            for $l = j + 1$ to $n - 1$ do          ▷ Left-looking.
                $A_{l,k} = A_{l,k} - A_{j,k} \cdot A_{l,j}$   ▷ Gather contributions from the already factorized columns.
            end for
        end for
        ($A$, $P$) = PIVOTING($A$, $P$, $k$)
        for $i = k + 1$ to $n - 1$ do
            $A_{i,k} = A_{i,k} / A_{k,k}$          ▷ Divide L by the chosen pivot.
        end for
    end for
end function
```

3 LU DECOMPOSITION

In this section, the basic algorithm for LU decomposition is revised, to give insights how the blocked algorithm will be implemented. Two basic transformation of a dense algorithm are in Algorithm 1. The SUBMATRIXLU is right-looking version of the algorithm and it is a common way of implementing dense LU decomposition. It is right-looking and produces one column of L and one row of U at a time. This is sometimes referred to by the order of the loops, as the `kij` algorithm.

The (partial) pivoting is performed by choosing a particular element from the lower part of the current column of A (Algorithm 1, line 2). The choice is typically performed as:

$$p_k = \underset{j}{\operatorname{argmax}} \left(|A_{j,k}| \cdot w_j \cdot \begin{cases} 1+t & \textit{if } j = k \\ 1 & \textit{otherwise} \end{cases} \right), \; j \geq k \,, \tag{1}$$

where w_j is approximate pivot weight vector, determined by taking row-wise L_∞ norm of A or 1 if no weighting is used and t is pivot threshold or 0 if no pivot thresholding is used. Upon choosing a pivot, the corresponding row is swapped with the current row k and this change is collected in the permutation matrix P (lines 4 and 5). If the weight vector w was used, the same swap would be performed there as well. To perform full pivoting, one would choose a pivot from the entire submatrix $A_{i,j} \mid i \geq k \wedge j \geq k$.

This algorithm yields a decomposition $LU = PA$, where the matrices overwrite A with $LU - I$, where I is an identity matrix–the unit diagonal of L that is not explicitly stored. This is a common way of representing the decomposition in the dense case.

The same algorithm is, however, not well suited for direct implementation of a *sparse* decomposition, as it requires access to both rows and columns of the matrix. If using a sparse storage format such as CSC, the matrix access pattern needs to be by columns–accessing the matrix by rows amounts to searching for every element and would be overly costly. Instead, by changing the order of the loops to `kji`, it is possible to arrive at COLUMNLU that only requires column-wise access. It is a left-looking algorithm and produces one column of the factorization at a time.

Conceptually, the first half of this algorithm is triangular solving (lines 24 to 29) and the rest is just choosing the pivot and column scaling. Note that in sparse case, swapping rows for pivoting would be inefficient and the implementations instead maintain a row permutation and simply renumber rows of all elements at the end (Davis 2006). Also note that due to always having only a single unreduced column, full pivoting is not easily attained. In the sparse case, the L and U matrices are stored separately, as it saves the work of searching for the diagonal elements when back-substituting later on.

4 PROPOSED ALGORITHM

The proposed algorithm is based on the procedure COLUMNLU described in the previous section, with several differences. The key difference is the use of a sparse *block* structure described in (Polok, Ila, and Smrž 2013). It is a column-major data structure similar to VBR (Du and Marrone 1992). It allows blocks of different sizes in a single matrix where the edges of the blocks must be aligned with each other, forming non-overlapping block rows and block columns. Each block is stored as a dense matrix, with the elements of all blocks being serialized in a single pooled array. This improves cache coherency of in-order traversal of the elements. Care is also taken for the blocks to be aligned in memory to allow vectorization using SSE instructions.

From the algorithmic point of view, each $A_{i,j}$ is a dense matrix rather than a scalar value. Thus, the product at line 27 of Algorithm 1 is actually a dense matrix product. Similarly, the division at line 25 is triangular solution of the form $L^{-1}_{j,j} \cdot U_{j,k}$ and the division at line 32 is another triangular solution, this time of the form $L_{i,k} \cdot U^{-1}_{k,k}$ where the triangular block $U_{k,k}$ is on the *right*. Those expressions are both evaluated using forward and back-substitution, respectively.

Another difference is the choice of the pivot block, which we refer to as the inter-block pivoting. For element-wise sparse matrices, this can be done according to (1). For blockwise matrices, this formula cannot be used directly and a way of reducing the blocks to scalar values needs to be devised. We tested a number of different metrics, including trace or L_1, L_2 and L_∞ norms of the block or of its diagonal, to no avail. Finally, using a product of diagonal entries in a block permuted to have the largest diagonal values (Duff and Koster 1999) gave reasonable results. This stems from the fact that all the off-diagonal elements will be in sequence divided by all the diagonal elements of the pivot block in the back-substitution mentioned above, and thus the final scaling is equal to their product. We also use pivot weighting by taking block row-wise L_∞ norm of A. This helps to choose better pivots in matrices with uneven distribution of off-diagonal value magnitudes.

Packages such as CSparse perform pivoting by element renumbering, leading to unsorted CSC matrices (the order of elements in each column is undefined). This potentially creates suboptimal memory access patterns if the number of elements in each column is greater than the effective size of the CPU cache. The proposed implementation addresses this problem differently and produces and maintains an ordered representation at all times. To do that, a method described in (Gustavson 1978) is employed: a dense vector of the same size as the current block column is used, along with a bit array to accumulate the values of the blocks and the sparsity pattern, respectively. Once the decomposition of the current block column is finished, the contents of this dense accumulator are scattered to the L and U matrices, in order.

An important difference is the factorization of the pivot block. This is a simple dense LU factorization and brings a choice of partial or full intra-block pivoting (Interestingly, the factorization of the pivot block does not immediately depend on or affect any other blocks.). Full pivoting has the advantage of revealing the rank of this pivot block. If the block is rank deficient, care must be taken when evaluating $L_{i,k} \cdot U_{k,k}^{-1}$. If the columns of $L_{i,k}$ corresponding to the zero diagonal entries in $U_{k,k}$ are also null, the division needs to be avoided otherwise the floating-point arithmetics would produce special values. If, on the other hand, those columns are nonzero, using this pivot would lead to division by zero and a different pivot needs to be chosen. If there is no full-rank pivot block in the current block column then either the factorization needs to fail, or elements from multiple different blocks would need to be combined. In the proposed implementation, this problem was handled by failing the factorization since it did not occur in the tested matrices.

In any case, the intra-block pivoting required by the pivot block factorization potentially reorders the rows and columns of this block and the changes need to be reflected on the rest of the matrix. While pivoting the columns affects only the current block column and can be applied immediately to all the other blocks, pivoting the rows affects the already processed block row of L and future block rows in U. Since accessing the matrix row-wise is prohibitive, the row permutation is only applied to the rows of U as they are produced and the permutation in L is performed at the end, after the factorization finishes.

5 EXPERIMENTAL EVALUATION

The proposed algorithm was evaluated on matrices from the University of Florida Sparse Matrix Collection (Davis 1994). Since the matrices in this dataset do not contain any information about block structure, a modified algorithm, based on routines `CSRKVSTR` and `CSRKVSTC` described in Saad (1994), was used to find block matrices with a particular block size and allowing a small amount of fill-in. In addition to that, we compared the implementations on block matrices associated with some standard SLAM problems in robotics: *Parking Garage* (Kümmerle et al. 2011), *KITTI Sequence 00* (Geiger et al. 2013), *Sphere 2500* (Kaess, Ranganathan, and Dellaert 2007) and BA problems in computer vision: *Fountain-P11* (Strecha et al. 2008), *Lourakis bundle1* (Lourakis and Argyros 2004), *Mazaheri bundle_-adj* (Davis 1994), *Venice871* (Kümmerle et al. 2011), *Fast & Furious 6* (Double Negative Visual Effects, http://www.dneg.com/.) and *Guildford Cathedral* (http://cvssp.org/impart/.).

All the matrices were pre-ordered using the same fill-reducing ordering, obtained by AMD of $AA^\top$ with dense columns dropped and applied at the granularity of blocks. This means that all the tested methods operated on identical inputs. The time to produce this ordering is not included in the timing results (since all the methods would use the same ordering scheme). Ultimately, this slightly favors element-wise approaches since the time complexity of the ordering algorithm is higher than linear in the size of the matrix (Heggernes, Eisesstat, Kumfert, and Pothen 2001) and thus ordering at the level of blocks is faster than ordering at the level of elements would be.

The experiments were performed on the Salomon supercomputer, part of the IT4I Czech National Supercomputing Center. Each compute node is equipped with a pair of 12-core Xeon E5-2680 v3 running at 2.50 GHz and 128 GB of RAM. Note that these CPUs have turbo boost technology which adjusts the clock frequency based on the available thermal envelope. This function was disabled for the benchmarks, so as to not make the results dependent on the variations in the temperature. All of the processing times would be lower with turbo boost enabled. The code was compiled as x64, and used 64-bit pointers. During the tests, the computer was not running any time-consuming processes in the background. We used the `g++` (`gcc`) 4.4.7 compiler (the proposed implementation is written in C++, while CSparse is written in C).

Each test was run at least ten times and until cumulative time of at least 5 seconds was reached, and the average time was calculated in order to avoid measurement errors, especially on smaller matrices. Note that each timing run was performed in a new process, so that there would be no cache reuse. We further experimented with flushing the cache lines containing the data, using the combination of `_mm_clflush()` and `_mm_mfence()` intrinsics. Furthermore, a 100 MB block of memory was read and written to make sure that the cache was completely flushed (care was taken that these accesses would not bypass the cache).

The effect of thus flushed cache was a small slow-down, on average 3.80% for CSparse and 2.86% for the proposed method (worst case 25.86% for CSparse and 14.43% for proposed). This effect was more pronounced on smaller matrices, as the larger matrices do not fit in the cache at once anyway. The flushing of the cache did not change the ranking of the methods on any of the tested matrices. The tests presented in the remainder of the evaluations herein are without flushing the cache, as it seems more natural that the matrix to be factorized is already (partially) in the cache (since the factorization function would be most likely called on a matrix that was just produced by other computation). It also makes the presented results more comparable to the results of other researchers. But note that there is still no cache reuse between individual benchmark runs as those are performed each in a new process.

Apart from the obvious timing evaluation and also recording the statistics about the factorization sparsity, relative factorization precision was evaluated, as:

$$\frac{||PAQ - LU||}{||A||} \, , \tag{2}$$

where P and Q are the block row and column permutation matrices (Q is only used in the proposed implementation, if full intra-block pivoting is applied).

Results for the SLAM and BA datasets can be seen in Table 1. In these datasets, the block partitioning is easily anticipated (unlike in the rest of the benchmarks where the block structure was estimated and might not exactly map to the original variables in some cases). Note that the system matrices of these datasets are symmetric but neither implementation took advantage of this fact, and the numbers of nonzeros (denoted “Nnz.”) are reported for both halves of the matrix. It is a common practice in solving BA problems to apply Schur complement as:

$$A = \begin{pmatrix} C & U \\ V & L \end{pmatrix} \tag{3}$$

$$\mathrm{Schur}(A) = C - UL^{-1}V \, , \tag{4}$$

where the variables in A are partitioned in such a way that C contains the camera variables and L contains the landmark variables. Due to the structure of the problem, L is block diagonal and easily invertible. The solution to the linear system $Ax = b$ then lies in decomposition of $\mathrm{Schur}(A)$ rather than of the entire A. To

Table 1: Results on SLAM and BA datasets. The times are in milliseconds unless specified otherwise. The first group are SLAM datasets, followed by a BA dataset and finally Schur-complemented BA datasets.

			CSparse			**Proposed**		
Name	**Size**	**Nnz.**	**Time**	**LU nnz.**	**Error**	**Time**	**LU nnz.**	**Error**
Garage	9966	285696	104.273	1135362	$8.16\cdot10^{-16}$	69.415	936360	$3.53\cdot10^{-16}$
KITTI 00	27246	477072	194.675	2078526	$6.48\cdot10^{-16}$	114.596	1673244	$1.90\cdot10^{-16}$
Sphere 2500	15000	268164	2597.737	6557052	$1.46\cdot10^{-15}$	1250.732	5190048	$1.04\cdot10^{-15}$
Fountain	23853	554427	1921.963	10602408	$3.49\cdot10^{-15}$	68.678	1112814	$1.73\cdot10^{-15}$
Lourakis	3115	31572	20.703	88473	$7.55\cdot10^{-16}$	13.011	86436	$2.75\cdot10^{-15}$
Mazaheri	3330	247068	149.431	793026	$5.51\cdot10^{-16}$	80.813	614880	$2.94\cdot10^{-16}$
Venice871	5226	5469048	125.990 s	26586570	$4.30\cdot10^{-15}$	57.254 s	26439624	$1.17\cdot10^{-15}$
FF6	960	137160	291.344	702228	$1.21\cdot10^{-15}$	162.147	604584	$2.30\cdot10^{-16}$
Cathedral	552	125244	82.581	278310	$1.86\cdot10^{-15}$	54.886	263592	$3.06\cdot10^{-16}$

reflect this in the tests performed here, the matrices of BA datasets in Table 1 are first Schur-complemented and then the results are reported for the LU decomposition of this Schur complement. An exception was made for the *Fountain* dataset due to its small size–the resulting times would be very close to zero.

The proposed implementation gets consistently better times and better precision, with the exception of the *Lourakis* dataset. The precision is in the 10^{-16} to 10^{-15} range, which corresponds to the roughly 15 digits that the double precision floating-point numbers can hold. The good speed is caused by the fact that these matrices are diagonally dominant and the proposed implementation can perform most of the pivoting inside of the blocks, reaching lower fill-in and thus also lower number of arithmetic operations than CSparse.

The results on the other matrices, from the University of Florida Sparse Matrix Collection, are in Table 2. The matrices are grouped by block size, starting with 4×4 and ending with 6×6. Although the collection contains much more block matrices, they typically contain mixtures of multiple block sizes. To limit the size of the evaluation to a reasonable size, we decided to only select matrices with a single block size (note that the implementation supports multiple block sizes, e.g. *Fountain-P11* contains 6×6, 6×3, 3×6 and 3×3 blocks).

While the good precision and sparsity holds up, the speedup grows with block size and the proposed implementation is slightly slower for 4×4 blocks. Such behavior is to be expected from blocked implementation where there are more nested loops and thus a larger ratio of control flow to arithmetics instructions. This could be easily improved by loop unrolling, e.g. as suggested in Polok, Ila, and Smrž (2013). Note that CSparse failed to factorize 12 of the tested matrices and so they were omitted to save space (on those matrices, the average relative error of the proposed implementation was $5.648 \cdot 10^{-16}$, with the worst case relative error being $1.363 \cdot 10^{-15}$). Additionally, several more matrices were omitted from groups of matrices having the same structure and getting the same results (e.g. *Schenk/AFE_af_shell1* through *Schenk/AFE_af_shell9* or the *Simon/venkat* group).

While already giving good results, there are several ways to improve the implementation to be even faster. In our previous work, we proposed fixed block size (FBS) optimization (Polok, Ila, and Smrž 2013), a form of register unrolling that is conveniently accessible by using the C++ language, making different block sizes or even their mixtures easily attainable via a simple interface and without having to manually rewrite or optimize any code. Using this optimization makes this method faster also on matrices with 3×3 blocks, the proposed method is faster than CSparse on 31 out of 33 matrices with average speedup 2.46×. The results of this optimization are plotted in Figure 1. This plot shows the speedup of the optimized algorithm compared to the results reported in Tables 1 and 2 as "Proposed". The speedup is greater for larger matrices and for smaller block sizes, especially for 3×3 and 4×4 that fit well in the SSE registers. The smaller gains

Table 2: Results on matrices from the University of Florida Sparse Matrix Collection. The times are in milliseconds unless specified otherwise. The first group are matrices with 4×4 blocks, followed by 5×5 and 6×6 blocks. Note that the names of the matrices were abbreviated in some cases, in order to save space.

			CSparse			**Proposed**		
Name	**Size**	**Nnz.**	**Time**	**LU nnz.**	**Error**	**Time**	**LU nnz.**	**Error**
HB/steam3	80	928	0.043	1068	$1.37\cdot10^{-17}$	0.132	1248	$5.36\cdot10^{-20}$
Simon/raefsky3	21200	1488768	8211.861	$149\cdot10^{5}$	$1.00\cdot10^{-15}$	8800.133	$133\cdot10^{5}$	$5.56\cdot10^{-16}$
Simon/venkat01	62424	1717792	5010.632	$178\cdot10^{5}$	$3.55\cdot10^{-16}$	6870.423	$179\cdot10^{5}$	$1.77\cdot10^{-16}$
Janna/CoupC3D	416800	$223\cdot10^{5}$	4.023 h	$151\cdot10^{7}$	$4.70\cdot10^{-16}$	3.103 h	$148\cdot10^{7}$	$5.25\cdot10^{-16}$
Oberwolf./piston	2025	100015	13.396	177445	$2.26\cdot10^{-16}$	14.486	148600	$6.91\cdot10^{-17}$
Fluorem/GT01R	7980	430909	1608.026	4446585	$7.95\cdot10^{-16}$	1232.581	3744400	$8.94\cdot10^{-05}$
Schenk/shell1	504855	$176\cdot10^{5}$	446.545 s	$382\cdot10^{6}$	$7.49\cdot10^{-15}$	356.576 s	$384\cdot10^{6}$	$1.32\cdot10^{-15}$
Schenk/shell10	$151\cdot10^{4}$	$527\cdot10^{6}$	2.541 h	$168\cdot10^{7}$	$5.07\cdot10^{-14}$	1.515 h	$168\cdot10^{7}$	$8.29\cdot10^{-15}$
Schenk/0_k101	503625	$176\cdot10^{5}$	486.978 s	$398\cdot10^{6}$	$6.74\cdot10^{-15}$	379.897 s	$399\cdot10^{6}$	$2.24\cdot10^{-15}$
HB/bcsstk02	66	4356	0.285	4422	$2.21\cdot10^{-16}$	0.394	4752	$1.49\cdot10^{-16}$
HB/bcsstk14	1806	63454	52.663	366174	$9.70\cdot10^{-16}$	28.607	281952	$3.20\cdot10^{-16}$
Nasa/nasasrb	54870	2677324	20.933 s	$407\cdot10^{5}$	$1.28\cdot10^{-15}$	8.769 s	$302\cdot10^{5}$	$6.15\cdot10^{-16}$
Simon/olafu	16146	1015156	3094.454	7964682	$4.25\cdot10^{-16}$	1739.891	6871176	$1.11\cdot10^{-13}$
DNVS/shipsec1	140874	7813404	1458.87 s	$314\cdot10^{6}$	$5.75\cdot10^{-16}$	295.690 s	$210\cdot10^{6}$	$9.31\cdot10^{-16}$
DNVS/x104	108384	$102\cdot10^{5}$	367.990 s	$174\cdot10^{6}$	$2.18\cdot10^{-15}$	73.593 s	$100\cdot10^{6}$	$8.90\cdot10^{-16}$
BenElechi/B.E.1	245874	$132\cdot10^{5}$	172.599 s	$180\cdot10^{6}$	$1.83\cdot10^{-15}$	101.651 s	$180\cdot10^{6}$	$7.23\cdot10^{-16}$

on large block sizes is given by the fact that those are already quite efficient even without this optimization. Note that the optimized method is never slower, and also that the precision of the results is identical to that of the unoptimized version.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an implementation of intra-/inter-block pivoting scheme based on the maximum diagonal product scoring of the pivot blocks. It serves as a showcase that limiting pivot search to relatively small blocks can yield excellent precision while at the same time promoting locality of reference and reducing the number of nonzero elements in the resulting factorization, as shown by the experimental evaluation. The proposed method was demonstrated on LU decomposition but is applicable also on other decomposition types, such as QR or $LDL^{\top}$.

The evaluation presented here was in comparison with CSparse. While it is very popular, it is a simplical, serial code. It would be interesting to compare the proposed algorithm to also other, more advanced implementations such as superLU (Li et al. 1999), MUMPS (Amestoy, Duff, L'Excellent, and Koster 2001) or HSL MA50 (Duff and Reid 1996). This evaluation would be well beyond scope (and space) of this study and we shall report it in a follow-up paper. A quantitative comparison of performance is in Figure 2, the proposed implementation peaks at 5.714 GFLOP/s, average is 2.537 GFLOP/s (single core performance). (The figures were arrived at by calculating the number of FLOPs required for the factorization, using the functionality described at https://sf.net/p/slam-plus-plus/wiki/Counting%20FLOPS%20in%20Sparse%20Matrix%20Operations/, and dividing that by average runtime (including the symbolic factorization) reported in Tables 1 and 2. The *Fountain-P11* dataset was excluded as the proposed implementation yields substantially more sparse factorization and the resulting figure would be unrealistic (31 GFLOP/s).)

The LU decomposition is also amenable to parallelization, which is another interesting direction, especially with respect to GPU implementation. Finally, the block methods are orthogonal to multifrontal and supern-

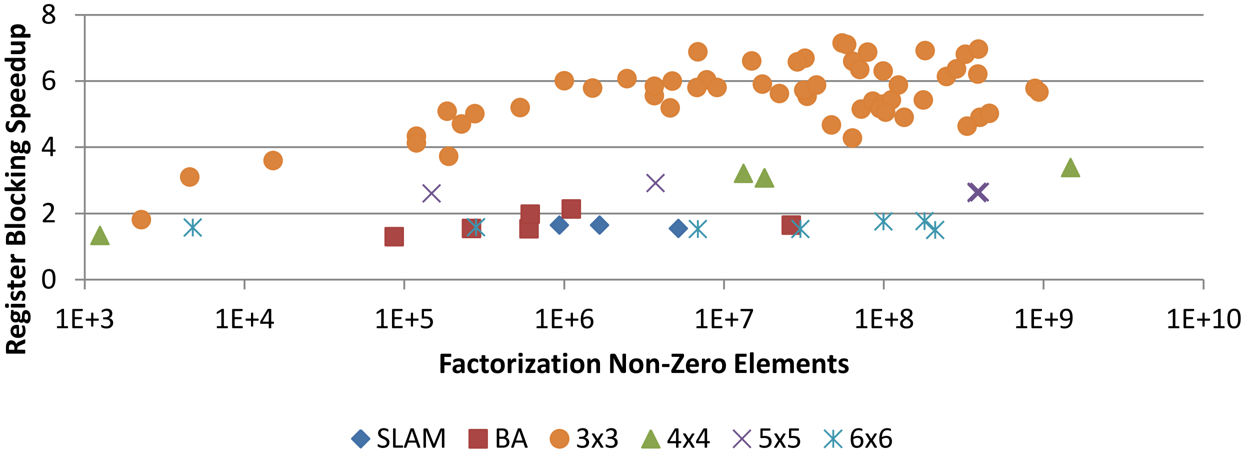


Figure 1: Relative speedup of the fixed block size (register blocking) optimization, compared to the unoptimized proposed algorithm.

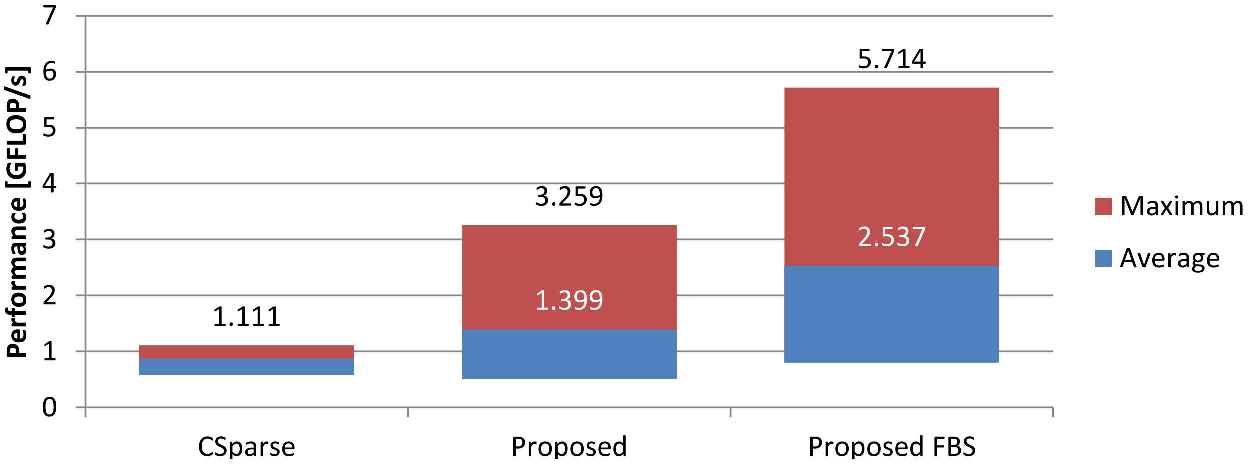


Figure 2: Quantitative evaluation of the compared algorithm performance in floating point operations per second (FLOP/s). The bottom of the bars indicates worst-case performance (0.56 GFLOP/s for CSparse, 0.50 GFLOP/s proposed and 0.78 GFLOP/s proposed with FBS optimization).

odal methods, that should both increase the performance even more, by using frontal matrices or supernode blocks and enabling dense computation on larger than the natural blocks.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science (LQ1602). The access to the Salomon supercomputer was supported by the IT4Innovations centre of excellence, via the Block LU Decomposition (BLUD) project (DD-17-5).

REFERENCES

S. Agarwal and K. Mierle 2012. “Ceres Solver”. http://ceres-solver.org/.

Amestoy, P. R., T. A. Davis, and I. S. Duff. 1996. “An approximate minimum degree ordering algorithm”. *SIAM J. on Matrix Analysis and Applications* vol. 17 (4), pp. 886–905.

Amestoy, P. R., I. S. Duff, J.-Y. L’Excellent, and J. Koster. 2001. “A fully asynchronous multifrontal solver using distributed dynamic scheduling”. *SIAM J. on Matrix Analysis and Applications* vol. 23 (1), pp. 15–41.

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney et al. 1987. *LAPACK Users’ guide*, Volume 9. SIAM.

Balay, S., S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. 2015. “PETSc Users Manual”. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory.

Carney, S., M. A. Heroux, G. Li, and K. Wu. 1994. “A Revised Proposal for a Sparse BLAS Toolkit”. Technical report, SPARER.

Davis, T. 1994. “The University of Florida Sparse Matrix Collection”. In *NA Digest*. Citeseer.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. SIAM.

Du, I., and M. Marrone. 1992. “A proposal for user level sparse BLAS”. Technical report, Rutherford Appleton Laboratory, Oxfordshire and CERFACTS, Toulouse and IBM Semea, Cagliari.

Duff, I. S. 1981a. “Algorithm 575: Permutations for a zero-free diagonal [F1]”. *ACM Trans. Math. Software* vol. 7 (3), pp. 387–390.

Duff, I. S. 1981b. “On algorithms for obtaining a maximum transversal”. *ACM Trans. Math. Software* vol. 7 (3), pp. 315–330.

Duff, I. S., and J. Koster. 1999. “The design and use of algorithms for permuting large entries to the diagonal of sparse matrices”. *SIAM J. on Matrix Analysis and Applications* vol. 20 (4), pp. 889–901.

Duff, I. S., and J. K. Reid. 1996. “The design of MA48: a code for the direct solution of sparse unsymmetric linear systems of equations”. *ACM Trans. Math. Software* vol. 22 (2), pp. 187–226.

F., J. 2009. “Intel Math Kernel Library. Reference Manual”. Technical report, Intel Corporation, Santa Clara, USA, 630813-054US.

Geiger, A., P. Lenz, C. Stiller, and R. Urtasun. 2013. “Vision meets Robotics: The KITTI Dataset”. *Intl. J. of Robotics Research*.

George, A., and J. Liu. 1989. “The evolution of the minimum degree ordering algorithm”. *SIAM Rev.* vol. 31 (1), pp. 1–19.

Gaël Guennebaud and Benoît Jacob and others 2010. “Eigen v3”. http://eigen.tuxfamily.org.

Gustavson, F. G. 1978. “Two fast algorithms for sparse matrices: Multiplication and permuted transposition”. *ACM Trans. Math. Software* vol. 4 (3), pp. 250–269.

Heggernes, P., S. Eisestat, G. Kumfert, and A. Pothen. 2001. “The computational complexity of the minimum degree algorithm”. Technical report, Technical Report No. ICASE-2001-42, Institute for Computer Applications in Science and Engineering.

Kaess, M., A. Ranganathan, and F. Dellaert. 2007, April. “iSAM: Fast Incremental Smoothing and Mapping with Efficient Data Association”. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1670–1677. Rome, Italy.

Kümmerle, R., G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. 2011, May. “g2o: A General Framework for Graph Optimization”. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*. Shanghai, China.

Li, X., J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki. 1999. “SuperLU User’s Guide”. Technical report, Technical Report No. LBNL-44289, Lawrence Berkeley National Laboratory.

Li, X. S., and J. W. Demmel. 2003. “SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems”. *ACM Trans. Math. Software* vol. 29 (2), pp. 110–140.

Liu, J. W. 1985. “Modification of the minimum-degree algorithm by multiple elimination”. *ACM Trans. Math. Software* vol. 11 (2), pp. 141–153.

Lourakis, M., and A. Argyros. 2004. “The design and implementation of a generic sparse bundle adjustment software package based on the Levenberg-Marquardt algorithm”. Technical report, Technical Report 340, Institute of Computer Science-FORTH, Heraklion, Crete, Greece.

Polok, L., V. Ila, and P. Smrž. 2013. “Cache Efficient Implementation for Block Matrix Operations”. In *Proc. of the High Performance Computing Symp.*, pp. 698–706, ACM.

Polok, L., V. Ila, M. Šolony, P. Smrž, and P. Zemčík. 2013. “Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics”. In *Robotics: Science and Systems (RSS)*.

Polok, L., M. Šolony, V. Ila, P. Zemčík, and P. Smrž. 2013. “Efficient Implementation for Block Matrix Operations for Nonlinear Least Squares Problems in Robotic Applications”. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*. IEEE.

Saad, Y. 1994. “SPARSKIT: a basic tool kit for sparse matrix computations–Version 2”. Technical report, Computer Science Department, Univ. of Minnesota, Minneapolis, MN.

Schenk, O., and K. Gärtner. 2006. “On fast factorization pivoting methods for sparse symmetric indefinite systems”. *etna* vol. 23 (1), pp. 158–179.

Strecha, C., W. von Hansen, L. Van Gool, P. Fua, and U. Thoennessen. 2008. “On benchmarking camera calibration and multi-view stereo for high resolution imagery”. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–8. IEEE.

AUTHOR BIOGRAPHIES

LUKAS POLOK was born in Brno (Czech Republic). Lukas received a MSc in Computer Science with a specialization in computer graphics and multimedia at Brno University of Technology, where he is presently employed as a researcher, working on efficient linear algebra algorithms using GPUs for general purpose calculations. His email address is ipolok ‘at’ fit.vutbr.cz.

PAVEL SMRZ is an associate professor in the Department of Computer Graphics and Multimedia, Faculty of Information Technology, Brno University of Technology, Czech Republic, where he leads the Knowledge Technology Research Group. His research interests include big data processing, hardware-accelerated machine learning, large-scale distributed and parallel processing, human-computer interaction, and information extraction. His email address is smrz ‘at’ fit.vutbr.cz.

FAULT TOLERANT VARIANTS OF THE FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION

Evan Coleman
Naval Surface Warfare Center
Dahlgren Division
17320 Dahlgren Rd
Dahlgren, VA, USA
ecole028@odu.edu

Masha Sosonkina
Dept. of Modeling, Simulation
and Visualization Engineering
Old Dominion University
5115 Hampton Blvd
Norfolk, VA, USA
msosonki@odu.edu

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology
266 Ferst Drive
Atlanta, GA, USA
echow@cc.gatech.edu

ABSTRACT

This paper presents an investigation into fault tolerance for the fine-grained parallel algorithm for computing an incomplete LU factorization. Results concerning the convergence of the algorithm with respect to the occurrence of faults, and the impact of any sub-optimality in the produced incomplete factors in Krylov subspace solvers are given. Numerical tests show that the simple algorithmic changes suggested here can ensure convergence of the fine-grained parallel incomplete factorization, and improve the performance of the use of the resulting factors as preconditioners in Krylov subspace solvers if faults do occur.

Keywords: Fault tolerance, parallel preconditioning, incomplete factorization, GPU acceleration.

1 INTRODUCTION

Fault tolerance methods are devised to increase both reliability and resiliency of high-performance computing (HPC) applications. On future exascale platforms, the mean time to failure (MTTF) is projected to decrease dramatically due to the sheer size of the computing platform (Cappello, Geist, Gropp, Kale, Kramer, and Snir 2014). There are many reports (Asanovic et al. 2006, Cappello et al. 2014, Snir et al. 2014, Geist and Lucas 2009) that discuss the expected increase in the number of faults experienced by HPC environments. This is expected to be a more prevalent problem as HPC environments continue to evolve towards larger systems. As the landscape of HPC continues to grow into one where experiencing faults during computations is increasingly commonplace, the software used in HPC applications needs to continue to change alongside it in order to provide an increased measure of resilience against the increased number of faults experienced. Sparse linear solvers constitute one of the major computational areas for applications that are run in HPC environments. These solvers are used in a variety of applications. In order to improve the performance of these solvers, oftentimes a preconditioner is used in conjunction with the Krylov sub-

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

space solver. One of the most commonly used classes of preconditioners is incomplete LU factorization. Future HPC environments are likely to include a heterogeneous mixture of computing resources containing different types of accelerators (e.g., GPUs and MICs), and therefore algorithms that can take advantage of the computing structure of accelerators naturally will be advantageous. The fine-grained parallel incomplete LU (FGPILU) algorithm proposed in (Chow and Patel 2015) is such an algorithm. The main contribution of this work is to analyze the ability of this algorithm to complete successfully despite the occurrence of a computing fault, and to offer variants of the original algorithm that aid in this goal.

Typically, faults are divided into two categories: hard faults and soft faults (e.g., Bridges, Ferreira, Heroux, and Hoemmen 2012). Hard faults cause immediate program interruption and typically come from negative effects on the physical hardware components of the system or on the operating system itself. Soft faults represent all faults that do not cause the executing program to stop; they are the focus of this work. Most often, these faults refer to some form of data corruption that is occurring either directly inside of, or as a result of, the algorithm that is being executed. Currently, they often manifest as bit-flips. As the rate that faults occurring in HPC environments continues to increase, it becomes increasingly important to ensure that these solvers are able to execute without suffering the negative consequences associated with a fault occurring. In order to properly investigate the impact of soft faults, one needs to select a fault model that fully encapsulates all of the potential impacts of a soft fault, implement the selected fault model into the algorithm to be investigated, and conduct the necessary experiments to determine the potential impact of a fault occurring during the selected algorithm. This paper examines the potential impact of soft faults on the fine-grained parallel incomplete LU factorization, and also investigates the use of fine-grained parallel incomplete LU algorithm generated preconditioners on Krylov subspace solvers. The structure of this paper is organized as follows: in Section 2, a brief summary of some related studies is provided, in Section 3, details concerning the fault model that is used throughout this work are given, in Section 4, background information is provided for the fine-grained parallel incomplete LU algorithm, in Section 5, a theoretical examination of the fine-grained parallel incomplete LU algorithm with respect to its stability in the presence of faults is undertaken, in Section 6, a series of numerical results are provided, while Section 7 concludes.

2 RELATED WORK

The expected increase in faults is detailed in Asanovic et al. 2006, Cappello et al. 2014, Snir et al. 2014, Geist and Lucas 2009. The self-stabilizing variant of the FGPILU algorithm introduced here was inspired by the self-stabilizing iterative solvers presented in Sao and Vuduc 2013, which in turn are built upon the ideas of selective reliability Bridges et al. 2012. The work done in this study to show the effectiveness of iterative methods when using a (possibly faulty) FGPILU preconditioner is done using the CG algorithm Saad 2003. The analysis of the potential performance of a Krylov subspace method using a potentially sub-optimal FGPILU algorithm is related to the analysis in Sao and Vuduc 2013. The results for the experiments conducted for this effort are presented similarly to the results in Chow and Patel 2015, Chow, Anzt, and Dongarra 2015, but with more of a focus on the impact that a soft fault can have on the execution of both the FGPILU algorithm, and the performance of an FGPILU preconditioner in a linear solver.

3 FAULT MODEL

Soft faults typically manifest as bit-flips. However, for the purposes of this study, a more numerical approach was taken to model the impact of a soft fault. It is important when looking forward towards producing fault tolerant algorithms for future computing platforms not to become too dependent on the precise mechanism that is used to model the instantiation of a fault. Much of the current research (e.g., Bronevetsky and de Supinski 2008) treats faults exclusively as a bit flip; which reflects the current method in which faults occur. Regardless of how a fault manifests in future hardware, the result will be a corruption of the data that is used by the algorithm. To this end, a more generalized, numerical scheme for simulating the occurrence of a fault is adopted. Several numerically based fault models have been utilized in recent studies. These include a perturbation-based fault model that injects a random perturbation into every element of a key data structure (Coleman and Sosonkina 2016b), and a numerical fault model that is predicated on shuffling the components of an important data structure (Elliott, Hoemmen, and Mueller 2015). Other numerical models, such as inducing a small shift to a single component of a vector have been considered as well Bridges, Ferreira, Heroux, and Hoemmen 2012. The fault model used in this paper is a modified version of the one initially developed in Coleman and Sosonkina 2016b and is related to the fault model developed in Elliott, Hoemmen, and Mueller 2015. Specifically, similar to Coleman and Sosonkina 2016b, the modified model (denoted here as mFTM) targets a single data structure and injects a small random perturbation into its each component only episodically, as opposed to doing so persistently contrary to in Coleman and Sosonkina 2016b. For example, if the targeted data structure is a vector x and the maximum size of the perturbation-based fault is ε, then proceed as follows: Generate a random number $r_i \in (-\varepsilon, \varepsilon)$ for every component x_i, where i ranges over entire length of x. Then set $\hat{x}_i = x_i + r_i$ for all i's. The resultant vector $\hat{x}$ is, thus, perturbed away from the original vector x. After a fault occurs, it is possible for an algorithm to detect the error and correct it. It was shown in Elliott, Hoemmen, and Mueller 2015 that the numerical soft-fault model proposed there corresponds to a "sufficiently bad" impact of a soft fault rather than tries to determine the "damage" *exactly* of a soft fault. By construction, the mFTM follows in the footsteps of the ones in Elliott, Hoemmen, and Mueller 2015. An exploration of the similarities and differences between the two models is presented in (Coleman and Sosonkina 2016a). Hence, simulating these numerical soft fault models for iterative algorithms may force them to run consistently through bad errors only. Furthermore, by varying the size of the perturbation in mFTM, it is possible to produce steadily impactful errors.

4 FINE-GRAINED PARALLEL INCOMPLETE LU FACTORIZATION

In the same manner as other incomplete LU factorizations, the fine-grained parallel incomplete LU (FGPILU) factorization attempts to write an input matrix A as the approximate product of two factors L and U where, $A \approx LU$. In traditional incomplete LU factorizations (for an overview, see Saad 2003), the individual components of both L and U are computed in a manner that does not lend itself naturally to parallelization. The recent FGPILU algorithm proposed in Chow and Patel 2015 allows each element of both of the factor to be computed asynchronously (i.e. independently), and progress towards the "true" incomplete LU factors in an iterative manner. To do this, the FGPILU algorithm progresses towards the factors L and U by using the property $(LU)_{ij} = a_{ij}$ for all (i, j) in the sparsity pattern S of the matrix A, where $(LU)_{ij}$ represents the (i, j) entry of the product of the current iterate of the factors L and U. This leads to the observation that the FGPILU algorithm (given in Algorithm 1) is defined by two non-linear equations:

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \qquad\qquad u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\,. \tag{1}$$

Following the analysis presented in (Chow and Patel 2015), it is possible to collect all of the unknowns l_{ij} and u_{ij} into a single vector x, then express these equations as a fixed-point iteration $x^{(p+1)} = G\left(x^{(p)}\right)$, where the function G implements the two non-linear equations described above. In a fault-free environment, it can be proven that the FGPILU algorithm is locally convergent in both the synchronous and asynchronous cases (see Section 3 in Chow and Patel 2015). The FGPILU algorithm is given in Algorithm 1. Keeping with the terminology used in (Chow and Patel 2015, Chow, Anzt, and Dongarra 2015) each of the passes that the algorithm makes in updating all of the l_{ij} and u_{ij} elements is referred to as a "sweep". After each sweep of the algorithm, the L and U factors progress closer to the L^* and U^* factors that would be found with a traditional incomplete LU factorization. To do this, the factors L and U are first seeded with an initial guess. In this study, the initial L factor will be taken to be the lower triangular part of A and the initial U will be taken to be the upper triangular portion of A. Adopting the approach in both (Chow and Patel 2015, Chow, Anzt, and Dongarra 2015) a scaling of the input matrix is first performed on A such that the diagonal

Algorithm 1: FGPILU algorithm as given in (Chow and Patel 2015)

```
Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors $L$ and $U$ such that $A \approx LU$
for $sweep = 1, 2, \ldots, m$ do
    for $(i,j) \in S$ do in parallel
        if $i > j$ then
            $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
        else
            $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
```

elements of A are equal to one. This can be accomplished by performing a similarity transformation with an appropriate scaling matrix D and using it to update A so that, $A = DAD^T$. As pointed out in (Chow and Patel 2015), this diagonal scaling is imperative to maintain reasonable convergence rates for the algorithm, so the working assumption throughout this paper is that all matrices have been scaled appropriately.

5 FAULT TOLERANCE FOR THE FGPILU ALGORITHM

In this section, some theoretical bounds on the impact of a fault on the FGPILU algorithm are developed, and these projected impacts are used to develop fault tolerant adaptations to the original FGPILU algorithm. Using the fault model described in Section 3, if a fault occurs at the computation of the k^{th} iterate (affecting the outcome of the $(k+1)^{st}$ vector, it is possible to write the corrupted $(k+1)^{st}$ iteration of x as

$$\hat{x}^{(k+1)} = G\left(x^{(k)}\right) + r\,, \tag{2}$$

where the vector r accounts for the occurrence of a fault. Note that the magnitude of r corresponds only to the soft fault that was injected (as implemented in mFMT) and is not a part of the FGPILU algorithm itself: For a no-fault sweep, $r = 0$. To track the progression of the FGPILU algorithm, it was proposed in (Chow and Patel 2015) to monitor the non-linear residual norm. This is a value $\tau = \sum_{(i,j)\in S} \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right|$, which decreases as the number of sweeps progresses the algorithm closer to the conventional ILU factorization. If a fault occurs then one or both non-linear equations from the FGPILU algorithm will have some amount of error. In particular, the update equations for l_{ij} and u_{ij} will become

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) + r_{ij}\,, \qquad u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} + r_{ij}\,, \tag{3}$$

where r_{ij} represents the component of the vector r that maps to the (i,j) location of the matrix. This shows that if a fault occurs during the computation of the incomplete LU factors that the non-linear residual norm τ will be affected. In order to ensure that a fault does not negatively affect the outcome of the algorithm, a simple monitoring of the non-linear residual norm is proposed. In principle, since $S \subset A$, when the FGPILU algorithm converges, the non-linear residual norm will be at a minimum. Further, since there is a contribution from every $(i,j) \in S$, the individual non-linear residual norms for each $(i,j) \in S$, denoted here by τ_{ij}, can be defined as $\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik}u_{kj} \right|$, where the total non-linear residual norm can always be recovered by taking the sum of all the individual non-linear residual norms over all $(i,j) \in S$. To establish a baseline for fault tolerance, define individual non-linear residual norms τ_{ij} for each $(i,j) \in S$ based on the initial guess that is used to seed the iterative FGPILU algorithm. In particular, if L^* and U^* are the initial guesses for the incomplete L and U factors, then take $l_{ij}^* \in L$ and $u_{ij}^* \in U$ and define baseline individual non-linear residual norms τ_{ij}^* using the original values τ_{ij} and the values $l_{ij}^* \in L$ and $u_{ij}^* \in U$.

Since for each sweep of the FGPILU algorithm, the components $l_{ij} \in L$ and $u_{ij} \in U$ can be computed, by testing the individual non-linear residual norms it is possible to determine if a large fault occurred. Specifically, it is of interest to determine if a fault occurred that was large enough to cause a potential divergence of the algorithm. To do this, first a tolerance t is set and then a fault is signaled if $\tau_{ij} > t$. Since the individual non-linear residual norms are generally decreasing as the FGPILU algorithm progresses, set $t = \max(\tau^*_{ij})$ initially (Line 3 of Algorithm 2), and then update t during the course of the algorithm if desired. Note that if a fault is signaled by any of the individual non-linear residual norms, it is only known that a fault occurred somewhere in the current row of the factor L or the current column of the factor U. As such, the conservative approach would require the rollback of both the current row of L and the current column of U to their values at the previous checkpoint (e.g., Lines 5 to 9 of Algorithm 2). Further, it is possible for the individual non-linear residuals as defined to increase by a small amount, especially at early iterations. To counteract the potential for reporting false positives on fault detection, the derivative of the global non-linear residual can be checked to ensure that it is also increasing before switching the current row and/or column (see Line 15 of Algorithm 2). This algorithm is detailed in Algorithm 2.

Algorithm 2: Checkpoint-Based Fault Tolerant FGPILU (CP-FGPILU)

```
Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors $L$ and $U$ such that $A \approx LU$
1  for $(i,j) \in S$ do in parallel
2      $\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right|$
3  $t = \max(\tau_{ij})$
4  for $sweep = 1, 2, \ldots, m$ do
5      if $Fault$ then
6          Set $i = \max_{i,j}(k^1_{ij})$ and $j = \max_{i,j}(k^2_{ij})$
7          Rollback $\{l_{ik}\}_{k=1}^{i-1}$ and $\{u_{kj}\}_{k=1}^{j-1}$
8          $Fault$ = FALSE
9          $sweep = sweep - 1$
10     else
11         for $(i,j) \in S$ do in parallel
12             if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj})/u_{jj}$
13             else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
14             $\tau_{ij} = \left| a_{ij} - \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \right|$
15             if $\tau_{ij} > t$ and $\tau' > 0$ then
16                 Set $k^1_{ij} = i$ and $k^2_{ij} = j$
17                 $Fault$ = TRUE
```

Note that if a fault is detected, the algorithm only restores (i.e., "Rollback") the affected row of L and column of U. Additionally, since in practice it has been proposed (Chow and Patel 2015, Chow, Anzt, and Dongarra 2015) to use a limited number of sweeps of the FGPILU algorithm as opposed to converging the algorithm according to the global non-linear residual norm, the number of sweeps conducted is decremented so that all elements of L and U are updated *at least* the desired number of times. Also, while no global communication is required to check for the presence of a fault, if a fault is detected there will be some communication required between processes to fix the effects of the fault. Note also that when using the CP-FGPILU algorithm, the size of the faults that are not caught by the algorithm are determined by the tolerance that is set. In particular, $||r|| \leq t$, where r represents a fault that was not caught by the proposed checkpointing scheme, since if $||r|| > t$ than the fault would be caught by the check on Line 15 of Algorithm 2. This, in turn, affects the update equations Eqs. (2) and (3).

Algorithm 3: Self-Stabilizing Fault Tolerant FGPILU (SS-FGPILU)

```
Input: Initial guesses for $l_{ij} \in L$ and $u_{ij} \in U$
Output: Factors $L$ and $U$ such that $A \approx LU$
for $sweep = 1, 2, \ldots, m$ do
    if $sweep \equiv 0 \mod F$ then
        for $(i, j) \in S$ do in parallel
            if $\{\|l_{ij}\|, \|u_{ij}\|\} \gg \|a_{ij}\|$ or $|\{l_{ij}, u_{ij}\} - a_{ij}|/|a_{ij}| > \beta$ or $\{l_{ij}, u_{ij}\} = \{0, \mathtt{NaN}\}$ then
              $\{l_{ij}, u_{ij}\} = a_{ij}$
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
    else
        for $(i, j) \in S$ do in parallel
            if $i > j$ then $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj})/u_{jj}$
            else $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}$
```

It is also possible to develop a variant of the checkpoint-based fault tolerant algorithm that replaces the entire factors L and U as opposed to only the affected areas - call this variant the Checkpoint All variant (CPA-FGPILU). In this case, a fault is declared if the currently computed global non-linear residual norm τ is some factor α greater than the previously computed non-linear residual norm τ_{i-1}. Note that, due to a combination of the asynchronous nature of the the FGPILU algorithm, the non-linear residual norm will not be strictly monotonically decreasing, especially as the algorithm proceeds closer to convergence. Therefore using the factor $\alpha = 1$, i.e., expecting a strict monotonic decrease, may cause the algorithm to report false positives, especially when nearing convergence.

The other variant of the FGPILU algorithm is a so-called *self-stabilizing* version that relies on completing an update sweep successfully with some regularity in order to ensure that the resulting L and U factors will form an effective preconditioner. While the two checkpoint-based fault tolerance schemes introduced above can be shown to be effective numerically (see Section 6), it is possible to recover from the occurrence of a fault without the need for storing intermediate copies of the computed L and U factors. The update sweep must be computed reliably; in particular, it cannot be negatively affected by the occurrence of a fault. In the algorithm as shown in Algorithm 3, an update sweep is expected every F iterations. The expectation is that the steps that are undertaken during the "update sweep" will be able to mitigate any potential consequences of a fault occurring during the prior $F - 1$ iterations.

5.1 Convergence of the fault tolerant FGPILU algorithm variants

In Chow and Patel 2015, the convergence of the original FGPILU algorithm (Algorithm 1) is examined by investigating the properties of the non-linear equations that define the algorithm, which are captured in the fixed point function $G(x)$ and the associated Jacobian $G'(x)$. In order to examine the convergence of both the CP-FGPILU and SS-FGPILU algorithms, consider the modified version of $G(x)$ that allows for the occurrence of a fault Since, the mFTM from Section 3 considers faults as a corruption of data via one-time perturbation, the modified Jacobian is equal to the original Jacobian. This implies that the local and global convergence results from Chow and Patel 2015 hold for the modified equations that describe the fault tolerant variants of the FGPILU algorithm. Generally, convergence for all of the variants relies on their producing the elements in the original domain of the problem (using either checkpointing or a stabilizing step); as the elements are updated convergence will eventually occur. For the proposed self stabilizing FGPILU, following Theorem 2 from Sao and Vuduc 2013, a result about the convergence may be stated as:

Theorem 1. *For any state of $l_{ij} \in L$ and $u_{ij} \in U$, if a correction is performed in the k^{th} sweep, all subsequent iterations are fault-free, no elements in the final L and U factors differ by more than β percent from the original factors in the matrix A, and β is chosen such that if a fault occurs a fault is signaled, then the SS-FGPILU algorithm will converge.*

Proof. This follows from noticing that the correcting (or "stabilizing") step (Lines 2 to 6 of Algorithm 3) ensures that the state $l_{ij} \in L$ and $u_{ij} \in U$ of the incomplete L and U factors will be in the original domain of the problem and then invoking the convergence arguments for the original FGPILU algorithm (see Chow and Patel 2015) which rely upon the assumptions and base arguments from Frommer and Szyld 2000. □

Note that finding an appropriate value for the the constant β may be difficult in practice in situations where approximate L and U factors cannot be determined by alternative means. The theorem only guarantees that if such parameters exist and can be found that the algorithm will converge successfully. The convergence of the checkpoint-based variants of the FGPILU variants follows directly from the convergence of the original FGPILU algorithm. Assuming that faults do not occur after a certain number of sweeps, the algorithm will converge under the assumption that it was successfully returned to a state not affected by a fault. Note that if a fault is detected, the state is restored to the last known good state - how recent that state is depends on the frequency with which the checkpoint is stored. More frequent storage of a "good" state via checkpointing will slow down the overall progression of the algorithm, but will provide a more recent fail-safe state if a fault is detected.

Finally, note that for all variants of the FGPILU algorithm if a fault occurs that is not caught by either the stabilizing step in Algorithm 3, or by the checkpointing step in Algorithm 2 it is possible for the Jacobian to move to a regime where the fixed point mapping that represents the FGPILU algorithm is no longer a contraction. In this case, the fault tolerance mechanisms of the FPGILU variants will not help, and subsequent iterations of the algorithm will not aid in convergence. Since the application of the FGPILU preconditioner is effectively only an approximate application of the conventional, fault-free ILU preconditioner, the application of the generated preconditioners can be expressed as, $\tilde{z_j} \approx P^{-1} v_j$. Both Chow and Patel 2015, Chow, Anzt, and Dongarra 2015 have shown that it is possible to successfully use the incomplete LU factorization resulting from the FGPILU algorithm before the algorithm has converged according to the progress of the non-linear residual. It is possible that any adverse affects that a fault may have on the convergence of the FGPILU generated incomplete LU factors will not have a meaningful impact on the convergence of the overarching iterative method (e.g. CG, GMRES, etc). This impact will be explored numerically in Section 6.

6 NUMERICAL RESULTS

The experimental setup for this study is an NVIDIA Tesla K40m GPU on the Turing High Performance Cluster at Old Dominion University. The nominal, fault-free iterative incomplete factorization algorithms and iterative solvers were taken from the MAGMA open-source software library (Innovative Computing Lab 2015). All of the results provided in this study reflect double precision, real arithmetic. The test matrices that were used predominantly come from the University of Florida sparse matrix collection maintained by Tim Davis (Davis 1994), and the matrices selected for this study are the same as the ones that were selected for the study (Chow, Anzt, and Dongarra 2015) that detailed the performance of the FGPILU algorithm on

GPUs without the presence of faults. There are six matrices selected from the University of Florida sparse matrix collection, and mimicking the approach in Chow, Anzt, and Dongarra 2015, all six of these matrices were reordered using the Reverse Cuthill-McKee (RCM) ordering in an effort to decrease the bandwidth and help to improve convergence. The two other test matrices that were used come from the finite difference discretization of the Laplacian in both 2 and 3 dimensions with Dirichlet boundary conditions. For the 2D case, a 5-point stencil was used on a 500×500 mesh, while for the 3D case, a 27-point stencil was used on a $50 \times 50 \times 50$ mesh. All of the matrices considered in this study are symmetric positive-definite (SPD) and as such the symmetric version of the FGPILU algorithm (i.e. the incomplete Cholesky factorization) was used. Also, recall from Section 4 that each of the eight matrices used in this study will be symmetrically scaled to have a unit diagonal in order to help improve the performance of the FGPILU algorithm. A summary of all of the matrices that were tested is provided in Table 1.

Table 1: Summary of the 8 symmetric positive-definite matrices used in this study.

Matrix Name	Abbreviation	Dimension	Number of Non-zeros
APACHE2	APA	715,176	4,817,870
ECOLOGY2	ECO	999,999	4,995,991
G3_CIRCUIT	G3	1,585,478	7,660,826
OFFSHORE	OFF	259,789	4,242,673
PARABOLIC_FEM	PAR	525,825	3,674,625
THERMAL2	THE	1,228,045	8,580,313
LAPLACE2D	L2D	250,000	1,248,000
LAPLACE3D	L3D	125,000	3,329,698

The experiments are divided into two sets. This first set of experiments focuses on the convergence of the FGPILU algorithm despite the occurrence of faults and features comparisons of the L and U factors produced by the preconditioning algorithms. Faults are injected into the FGPILU algorithm following the methodology described in Section 3. Due to the relatively short execution time of the FGPILU algorithm on the given test problems, a fault is induced only once during each run, at a random sweep number before convergence. Three fault-size ranges were considered: $r_i \in (-0.01, 0.01)$, $r_i \in (-1, 1)$, and $r_i \in (-100, 100)$. Results for the three ranges are averaged and presented in Section 6.1. The second set of experiments shows the impact of using in a Krylov subspace solver the preconditioners obtained from the first set of experiments. Note that in all of the experiments conducted, the condition $u_{jj} = 0$ was never encountered. Since all the test matrices are SPD, the preconditioning algorithms are Incomplete Cholesky variants, and the the solver is the preconditioned conjugate gradient (PCG), as implemented in the MAGMA library.

6.1 Convergence of FGPILU algorithm

In order to obtain representative results, a fault from each range is injected once, on a single iteration and the results are averaged over approximately 30–40 runs per problem, all of which are successfully converged cases. For the purposes of this study, the FGPILU algorithm is said to have converged successfully if the non-linear residual norm progresses below 10^{-8}. Although this threshold is unnecessarily small from a practical point of view,—it is possible to achieve good performance from a preconditioner with a larger non-linear residual norm—it was chosen so that more sweeps would have to be conducted before the algorithm converges to better judge the impact of faults. The progression of the non-linear residual norm for a single fault-free run of each problem is depicted in Fig. 1(left), which is a as an example of the typical progression of the non-linear residual norm as the algorithm progresses towards convergence.

To illustrate the potential impact of a fault, Fig. 1(right) shows the impact a fault can have on the FGPILU algorithm when it is injected (and ignored) at the beginning, the middle, or near the end of how long it would

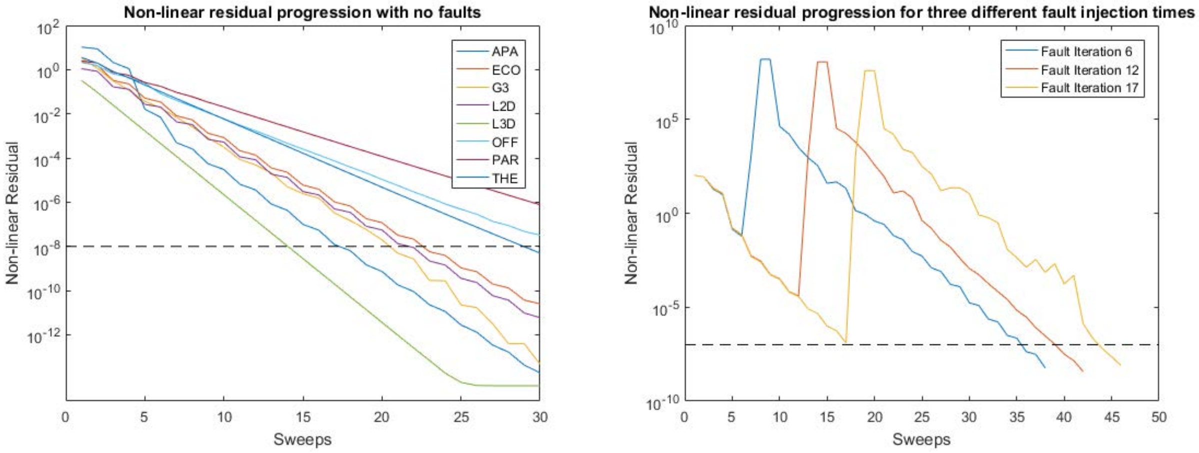


Figure 1: The progression of the non-linear residual for 30 sweeps of a typical fault-free run for each of the 8 test problems (left). The progression of the non-linear residual for the Apache test problem for three different fault injection times and fault size in the $(-1,1)$ range (right). The horizontal dashed line is indicated the FGPILU convergence tolerance of 10^{-8}.

take the algorithm to converge with no faults present. Note from Fig. 2(left) that the Apache test problem converges in 20 iterations when faults are not present. From Fig. 1(right), it may be observed that it took about twice as many sweeps for FGPILU to converge under a single occurrence of a fault; and the number of these extra sweeps is similar for the three injection places. Although the example shown in Fig. 1(right) is typical of what what was observed experimentally with the test cases selected, it is by no means general or conclusive: Faults may cause the FGPILU algorithm to diverge entirely or the resulting *L* and *U* factors may cause the PCG solver to either stagnate or even diverge. A major point of the example in Fig. 1(right) is to report the beneficial effects on FGPILU convergence of larger number of sweeps if faults are ignored in FGPILU and to show the non-monotonous decrease of the FGPILU residual norm after a fault takes place.

Aggregate results for the performance of several variants of FGPILU algorithm are provided in Fig. 2 as follows: when no attempt is made to mitigate the impact of the faults (denoted `No FT`), the CPA-FGPILU variant wherein the *L* and *U* factors may be replaced in their entirety (`CPA`), CP-FGPILU described in Algorithm 2 (`CP`), SS-FGPILU which is given in Algorithm 3 (`SS`). As in Fig. 1, additional sweeps of the FGPILU algorithm are conducted until the non-linear residual norm falls below 10^{-8}. Since the fault injection could occur on any single sweep, results from all runs are averaged to find the total number of sweeps necessary for the algorithm to converge.

Figure 2(left) shows the average number of sweeps to reach convergence for the cases that were successful. Note that this number is generally lower for the checkpoint-based schemes, but that this is not the case for all of the problems that were tested. However, the higher success rate of the CPA-FGPILU and CP-FGPILU algorithms combined with the generally faster convergence of those methods suggests that, with the parameters used in this study, they are more effective at mitigating faults. The small degradation in the number of sweeps to convergence depicted in Fig. 2(left) for certain problems (i.e., L3D) for the `No FT` variant reflects the fact that only successful runs are included in the averages here. In Fig. 2(right), a corresponding drop in the "success rate" can be seen for the problems where the increase in the number of sweeps required is not as large as expected for variants without fault mitigation. Here, a preconditioner is deemed as resulting in success if the PCG solve using it terminates before the maximum number of iterations is reached. For the FGPILU variants tested, the success rates captured in Fig. 2(right) show that both of the checkpoint-based variants are usually more successful than the self-stabilizing one at mitigating faults and producing acceptable preconditioners. It is important to note that a large, unoptimized value of

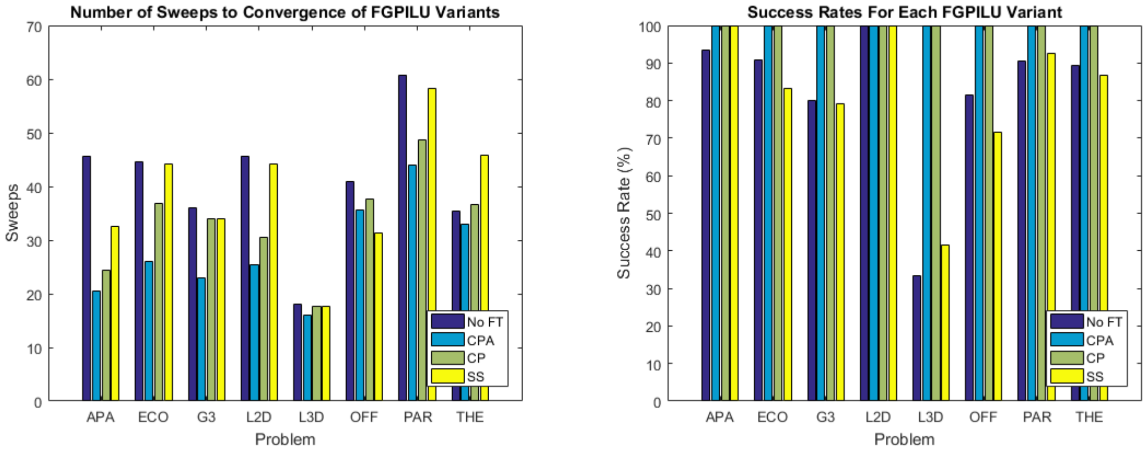


Figure 2: The numbers sweeps required for convergence for each of the 8 test problems (left). The percentage of runs that produced a preconditioner that corresponded to a successful PCG solve (right).

$\beta = 4$ was used for the percent difference check inside of the `SS` runs, and that this value may certainly be improved and tuned for the particular case at hand. The lower success rates associated with the SS-FGPILU algorithm are due to the fact that some of the smaller faults are not caught by this large value of β and the Jacobian moves to a portion of the domain where the mapping is not a contraction. Finding a way to obtain optimal parameters for the SS-FGPILU algorithm efficiently from intrinsic properties of the linear system in question is left as future work.

6.2 Preconditioner Performance in Iterative Methods

In this set of experiments, a maximum number of 3000 PCG iterations was used; any run that had not converged by that point was declared to have diverged. While all of the preconditioners to be evaluated are forms of incomplete LU decomposition, they are constructed by algorithms described in Section 6.1. For the purpose of an extended comparison, results are provided for the traditional Incomplete Cholesky (`IC`) and the Fine Grained Parallel Incomplete Cholesky (`ParIC`); neither of these two variants is subjected to faults. Figure 3 captures only the cases in which a preconditioner was successfully prepared (c.f. Fig. 2(right)). Figure 3(left) indicates that a successful FGPILU variant is typically capable of accelerating the PCG solve to the levels similar to those achieved by the no-fault constructions of incomplete LU. The timing results presented in Fig. 3(right) are for the total time required for the preconditioner preparation and PCG solve. While the former may vary much depending on which variant is considered, the latter is rather uniform across the variants due to their similar numbers of iterations performed to convergence. More efficient implementations of the fault tolerance mechanisms and a more realistic tolerance for the non-linear residual norm may improve the performance of the three fault-tolerant variants of the FGPILU algorithm.

7 CONCLUSIONS

This paper has investigated the potential of the FGPILU algorithm to tolerate and mitigate certain soft faults arising in the construction of L and U factors. Three recovery techniques were presented. Namely, they are (1) checkpointing of the entire data structure comprising the factors, (2) checkpointing of select pieces of the data structure, or (3) self-stabilizing of the algorithm to avoid checkpointing altogether. Comparisons of the three techniques suggest that, while checkpointing appears to be somewhat more robust, the self-stabilizing algorithm is competitive and may be preferred due to its greater flexibility of fault mitigation,

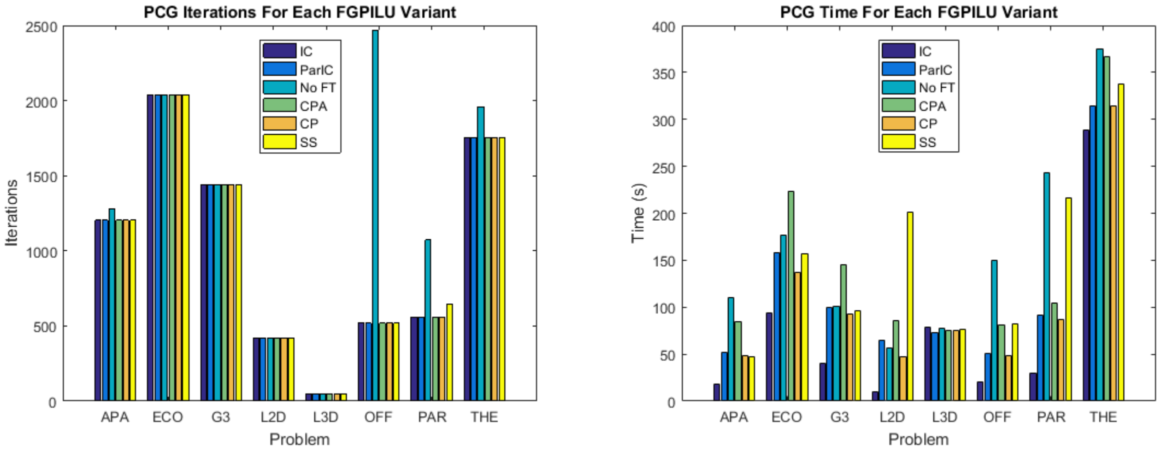


Figure 3: The numbers of iterations required for the successful PCG solves for each of the 8 test problems (left). The time required for the successful PCG solves for each of the 8 test problems (right).

e.g., when its parameters are are tuned for a better success rate on a given problem. Conversely, if the selective checkpointing scheme is optimized (e.g., by checkpointing less frequently or by rolling back fewer elements) the cost of checkpointing may be further reduced. The fault-tolerant techniques and findings presented in this paper may be readily applied to the asynchronous iterative methods in general to make them more robust in the presence of soft faults. While it has been shown that it is possible to generate a suitable ILU preconditioner with a small number of sweeps of the FGPILU algorithm in prior work, the use of asynchronous preconditioning algorithms is increasing in general, and as new asynchronous preconditioning algorithms are developed some may use the FGPILU algorithm as a building block and require the FGPILU algorithm to execute successfully inside of a more complex preconditioning scheme. In these cases, it may be important to have the FGPILU algorithm converge more completely, and the work presented here could be used as a starting point towards ensuring that can happen successfully even when computing faults happen to occur.

ACKNOWLEDGMENTS

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the U.S. Department of Energy (DOE) Office of Advanced Scientific Computing Research under the grant DE-SC-0016564 and through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC00-07CH11358, by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant, through the ILIR/IAR program at NSWC Dahlgren, and by the Turing High Performance Computing cluster at Old Dominion University. The authors would also like to thank the reviewers for their comments that helped improve the quality of this paper.

REFERENCES

Asanovic, K., R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams et al. 2006. “The landscape of parallel computing research: A view from Berkeley”. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bridges, P., K. Ferreira, M. Heroux, and M. Hoemmen. 2012. “Fault-tolerant linear solvers via selective reliability”. *arXiv preprint arXiv:1206.1390*.

Bronevetsky, G., and B. de Supinski. 2008. “Soft error vulnerability of iterative linear algebra methods”. In *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 155–164. ACM.

Cappello, F., A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. 2014. “Toward exascale resilience: 2014 update”. *Supercomputing frontiers and innovations* vol. 1 (1).

Chow, E., H. Anzt, and J. Dongarra. 2015. “Asynchronous iterative algorithm for computing incomplete factorizations on GPUs”. In *International Conference on High Performance Computing*, pp. 1–16. Springer.

Chow, E., and A. Patel. 2015. “Fine-grained parallel incomplete LU factorization”. *SIAM Journal on Scientific Computing* vol. 37 (2), pp. C169–C193.

Coleman, E., and M. Sosonkina. 2016a. “A Comparison and Analysis of Soft-Fault Error Models using FGMRES”. In *Proceedings of the 6th annual Virginia Modeling, Simulation, and Analysis Center Capstone Conference*. Virginia Modeling, Simulation, and Analysis Center.

Coleman, E., and M. Sosonkina. 2016b. “Evaluating a Persistent Soft Fault Model on Preconditioned Iterative Methods”. In *Proceedings of the 22nd annual International Conference on Parallel and Distributed Processing Techniques and Applications*.

Davis, TA 1994. “The University of Florida Sparse Matrix Collection”. http://www.cise.ufl.edu/research/sparse/matrices/.

Elliott, J., M. Hoemmen, and F. Mueller. 2015. “A Numerical Soft Fault Model for Iterative Linear Solvers”. In *Proceedings of the 24nd International Symposium on High-Performance Parallel and Distributed Computing*.

Frommer, A., and D. Szyld. 2000. “On asynchronous iterations”. *Journal of computational and applied mathematics* vol. 123 (1), pp. 201–216.

Geist, A., and R. Lucas. 2009. “Major computer science challenges at exascale”. *International Journal of High Performance Computing Applications*.

Innovative Computing Lab 2015. “Software distribution of MAGMA”. http://icl.cs.utk.edu/magma/.

Saad, Y. 2003. *Iterative methods for sparse linear systems*. Siam.

Sao, P., and R. Vuduc. 2013. “Self-stabilizing iterative solvers”. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pp. 4. ACM.

Snir, M., R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson et al. 2014. “Addressing failures in exascale computing”. *International Journal of High Performance Computing Applications*.

AUTHOR BIOGRAPHIES

EVAN COLEMAN is a scientist with the Naval Surface Warfare Center Dahlgren Division. He holds an MS in Mathematics from Syracuse University and is working on a PhD in Modeling and Simulation from Old Dominion University. His email address is ecole028@odu.edu.

MASHA SOSONKINA is a Professor of Modeling, Simulation and Visualization Engineering at Old Dominion University. Her research interests include high-performance computing, large-scale simulations, parallel numerical algorithms, and performance analysis. Her email address is msosonki@odu.edu.

EDMOND CHOW is an Associate Professor in the School of Computational Science and Engineering at Georgia Institute of Technology. His research interests are in numerical methods and high-performance computing for solving large-scale scientific computing problems. His email is echow@cc.gatech.edu.

EVALUATING EFFECTS OF APPLICATION BASED AND AUTOMATIC ENERGY SAVING STRATEGIES ON NWCHEM

Vaibhav Sundriyal
Department of Modeling,
Simulation, and Visualization Engineering
Old Dominion University
Norfolk, VA, 23529
vsundriy@odu.edu

Ellie Fought
Department of Chemistry
Iowa State University
Ames, IA 50010, USA
foughtel@iastate.edu

Masha Sosonkina
Department of Modeling,
Simulation, and Visualization Engineering
Old Dominion University
Norfolk, VA, 23529
msosonki@odu.edu

Theresa L. Windus
Department of Chemistry
Iowa State University
Ames, IA 50010, USA
twindus@iastate.edu

ABSTRACT

High-performance application developers are becoming increasingly aware of effects of the increasing energy consumption on the costs and reliability of modern computing systems. A traditional way to achieve energy savings is by changing the processor frequency dynamically during application execution. Several techniques have been proposed in the past at application, library, and transparent level. In this work, the effect of two such techniques, at application and transparent levels, are evaluated in terms of their effects on the execution time and energy consumption for different algorithms in the quantum chemistry package NWChem. Experimental results depict that there is no clear winner between the two methods since the transparent-level makes decisions without intimate knowledge of the application while the strategy based solely on application does not take into the account the platform characteristics at the runtime. Hence, it is argued that the best strategy would be a hybrid of the two levels.

Keywords: DVFS, Energy, Power, NWChem, Oversubscription.

1 INTRODUCTION

Power consumption has become a major concern in the design of modern computing systems due to the fact that power consumption varies as the product of the square of the voltage and the operating frequency which are interdependent as well. For the current topmost petascale computing platforms in the world, it is typical to consume power on the order of several megawatts, which at current prices may cost on the order of several million dollars annually. To address this challenge, power and energy optimizations are needed in modern computing platforms at all levels: application, system software, and hardware.

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA

The CPU and memory subsystems are the major energy consumers in a computing system, together contributing about 50-70% (Ge, Feng, Song, Chang, Li, and Cameron 2010) of the node power. To tackle the issue of increasing power consumption during application execution, DVFS (Dynamic Voltage and Frequency Scaling) has been one of the most used techniques in recent times. The current generation of Intel processors provides different power states (P-states) DVFS. More specifically, the Intel "Haswell-EP" microarchitecture provides a total of ten P-states. The delay of switching from one state to another depends on the relative ordering of the current and desired states, as discussed, e.g., in (Park, Shin, Chang, and Pedram 2010). The user may write a value to model-specific registers (MSRs) to change the P-state of the processor. The "Haswell" micro-architecture estimates power and energy consumption of the CPU and memory through the built-in MSRs, which certainly facilitates power-measurement efforts. The terms PF and MF are used to denote processor-frequency scaling and memory-frequency scaling, respectively, in this work.

In general, energy savings can be obtained by making judicious use of DVFS in modern computing systems at application/library/transparent levels. Since modifying the application itself can be tedious and does not provide enough information about the application characteristics on a particular hardware platform, this approach in general has been ignored by the researchers in the past.

In this work, an application and a transparent level DVFS based technique is evaluated in terms of its energy saving potential on NWChem on two different hardware platforms providing different DVFS applicability domains. The obtained results provide deep insight into the behavior of the different MP2 algorithms when operated under a specific DVFS based strategy along with the differing hardware platforms. The rest of the paper is organized as follows. Section 2 describes the related work and Section 3 provides an overview of the NWChem package. Section 4 discusses the salient features of the two DVFS based techniques. Section 5 details the experimental hardware and the results, while Section 6 concludes the paper and discusses the future work.

2 RELATED WORK

With regards to the library/automatic level application of DVFS, mainly two approaches exist. The first approach focusses on identifying stalls during the execution by measuring architectural parameters like instructions retired and memory accesses per second from performance counters as proposed in (Ge, Feng, Feng, and Cameron 2007, Hsu and Feng 2005, Huang and Feng 2009). The adaptive frequency scaling approach "ondemand" present in the linux kernel was compared to automatic strategy in (Huang and Feng 2009) and it was shown to be considerably less effective. The other approaches primarily focus on scaling processor frequency during slack or communication operations during application runtime. The techniques in the past have targeted communication intervals in parallel applications that use either explicit message passing (Freeh and Lowenthal 2005, Lim, Freeh, and Lowenthal 2006) or global address-space primitives (Vishnu, Song, Marquez, Barker, Kerbyson, Cameron, and Balaji 2010) and then scales the frequency for those intervals. Oversubscribing the processor cores (Iancu, Hofmeyr, Blagojevic, and Zheng 2010) is another technique which can be used to reduce execution time and lower power consumption of a parallel application. Since the efficacy of oversubscription is highly dependent on the inherent nature of the application and operating system features, it has not been widely used as a dedicated technique to improve energy efficiency in modern computing systems. An application based DVFS strategy focusing on NWChem was proposed in (Sundriyal, Fought, Sosonkina, and Windus 2016) which applied either PF of MF depending on the phase of the execution.

David *et al.* (David, Fallin, Gorbatov, Hanebutte, and Mutlu 2011) propose a memory-frequency scaling mechanism based on a memory bandwidth usage estimate such that frequency scaling is applied whenever memory-bandwidth usage goes below a certain level. In (Ge, Feng, He, and Zou 2016) authors study the impact of power allocation to different components of a computing system and attempt to determine the optimal allocation budget using derived heuristics and profiling. The Intel Haswell-EP processor generation is studied in terms of its new features in micro-architectural innovations and frequency scaling especially the per core frequency scaling capability, in (Hackenberg, Schöne, Ilsche, Molka, Schuchart, and Geyer 2015). An algorithm to automatically determine performance and power models of parallel applications without relying on previous execution data was proposed in (Sensi, Torquati, and Danelutto 2016).

3 NWCHEM OVERVIEW

NWChem (Valiev, Bylaska, Govind, Kowalski, Straatsma, Dam, Wang, Nieplocha, Apra, Windus, and de Jong 2010) is an ab initio computational chemistry software package that provides many methods for computing the properties of molecular and periodic systems using standard quantum mechanical descriptions of the electronic wavefunction or density. It is a scalable, portable, open-source computational chemistry software package, which was designed to be used on multiple platforms and with different computer hardware. However, the prime design target is high performance computers and efficient use of all of the hardware components on those platforms.

Møller-Plesset Second Order Perturbation Theory (Knowles, Andrews, Amos, Handy, and Pople 1991, Lauderdale, Stanton, Gauss, Watts, and Bartlett 1991), or MP2, is an electron correlation method used in association with the self-consistent field, or Hartree Fock (HF) method, in quantum chemistry. MP2 is based on the foundational Rayleigh-Schrödinger perturbation theory using the Fock operator as the unperturbed operator, and is one of the most widely used quantum mechanical correlation methods available. The HF method is an iterative procedure that assumes that each electron is in a mean field of the other electrons and, therefore, ignores the instantaneous correlations of the electrons with one another. The MP2 method is a correction to include this correlation energy. There are three common MP2 algorithms used in NWChem: semi-direct, direct, and resolution of the identity MP2 (Feyereisen, Fitzgerald, and Komornicki 1993, Bernholdt and Harrison 1996), or RI-MP2.

In two of the MP2 methods (direct and semi-direct) within NWChem, an order N^5 transformation of the atomic orbital integrals to the molecular orbital integrals is required, where N is the number of atomic orbital basis functions for the molecule of interest. This is the computational bottleneck for these MP2 energy methods. The direct MP2 performs all calculations and stores all of its integrals and other data to local memory. Any integrals that cannot be stored in memory are recalculated when they are subsequently needed in the calculations. For semi-direct MP2 some transformed integrals get stored in memory or recalculated and some (expensive ones) get written to disk. The semi-direct method is widely used, particularly because of its lower memory requirements. Both direct MP2 and semi-direct MP2 use very similar mathematical algorithms in their calculations and mostly differ in the integral storage.

RI-MP2, on the other hand, uses the resolution of the identity mathematical approximation to transform four-center integrals to three-center integrals. Each of the three-center integrals requires less time to compute, but results in more terms to calculate. RI is only exactly true when the basis set is complete which in general, is not possible for a finite basis set and so a large auxiliary basis sets is used to have a more complete basis set for the RI part. This approximation also changes the order N^5 MO integral transformation into an order N^4 operation. However, the final energy evaluations is still order N^5 due to the extra multiplications involved to approximate the four-center integrals. The NWChem implementation also uses on order N^2 total memory and order N^3 disk storage (much less than the semi-direct method, but more than the direct method).

In this work, the test cases are referred to by the system size, i.e., the number of atoms in the system, and by the MP2 algorithm employed: The test-name suffix “s” stands for semi-direct, “d” for direct, and “r” for RI-MP2. The system with 40 atoms having 335 standard and 800 auxiliary basis functions per molecule was chosen for the experimental evaluation. Although a 55-atom system was used in the authors’ work (Sundriyal, Fought, Sosonkina, and Windus 2016) to experiment with large calculations, the 40-atom input was chosen here instead, to provide fair comparisons of the platforms used in terms of efficient executions of the calculations. Conventional SCF was used in all the experiments conducted. Each test was executed at least three times to determine reproducibility of the results and the average values of those runs are presented in the paper.

4 OVERVIEW OF THE STRATEGIES

4.1 Application Based

A "hands-on" instrumentation (denoted as "HI") of the code DVFS based strategy was proposed in (Sundriyal, Fought, Sosonkina, and Windus 2016) with steps as follows to save energy:

- `If` in the startup section, `then` lower the MF while keeping PF at the maximum.
- `If` in the HF section, `then` lower the PF while keeping MF at the maximum.
- `If` in the MP2 section, `then` lower the MF and set the PF to the maximum.

The proposed HI strategy was further refined based on the range of the available PFs considered. The *aggressive* HI uses the minimum values of the respective, processor and memory frequencies to aggressively target possible energy savings. On the other hand, the *moderate* HI strategy uses the middle value in the range of available processor and memory frequencies. Since the results in (Sundriyal, Fought, Sosonkina, and Windus 2016) depicted that the *aggressive* HI provided higher energy savings compared to *moderate* HI, the *aggressive* HI is considered for evaluation purposes in this work.

4.2 Automatic Strategy

An automatic runtime strategy proposed in (Sundriyal and Sosonkina 2016b) is considered here as transparent to the application. This strategy predicts the micro-operations retired at different processor and memory frequencies along with the system power consumption and selects the appropriate processor–memory frequency pairs that minimize total energy consumption while satisfying the performance-loss constraint. The runtime strategy makes use of timeslices and uses history-window approach to predict application future behavior.

The base performance model that has been employed in the automatic strategy is

$$\mu\tau(i,j) = \frac{f_p(i)}{\mathrm{CPM}_{\mathrm{exe}} + \mathrm{MLIF}(j) \times \alpha \times \mathrm{MAPM} \times \beta \times \frac{f_p(i)}{f_p(1)}} \ . \tag{1}$$

where

- $\mu\tau(i,j)$ is the actual number of micro-operations retired per second at processor frequency $f_p(i)$ and memory frequency $f_m(j)$.
- $\mathrm{CPM}_{\mathrm{exe}}$ is the number of cycles per micro-operations retired barring the memory accesses in a second.
- α $(0 \le \alpha \le 1)$ is the OOO (out-of-order) overlap factor, which determines the extent of memory stalls overlapped with execution cycles.
- MAPM is the number of memory accesses per micro operation retired in a second.
- β is the number of cycles corresponding to the memory-access latency.

- MLIF(j) is the memory latency increase factor at the memory frequency $f_m(j)$ which depicts the relative increase in memory latency at memory frequency $f_m(j)$ compared to the highest memory frequency $f_m(1)$.

Using this performance model and an adaptive mechanism to adjust dynamically the memory and compute-intensity of an application, the strategy was shown delivering significant energy savings for the SPEC CPU$^{\text{TM}}$ 2006 and NAS parallel benchmarks.

5 EXPERIMENT SETUP AND RESULTS

Two different testbeds are used in this work, each of them having a particular DVFS granularity and subsequent effective frequency. DVFS granularity refers to the grouping of cores in a processor socket with respect to independent frequency and voltage scaling domains. In (Sundriyal and Sosonkina 2016a), it was determined that an energy saving strategy will be ineffective on an application which has variable workload nature on different cores, executing on a hardware platform which doesn't have per core DVFS granularity.

The effective frequency f_{eff} can be defined as the frequency experienced by a multicore node, when each core i $(i = 0,\ldots,n-1)$ is in a certain P-state f_i. For expressing the effective frequencies of the two platforms used in this work, n core testbeds are considered, where n is even, and supporting a specific level of the DVFS application.

1. `Marquez`[1] is comprised of a single node with two Intel Xeon CPU E5-2630 v3 "Haswell-EP" eight-core processors (two sockets) with 32 GB of main memory. The Intel Xeon E5-2630 v3 processor provides thirteen P-states ranging from 1.2 to 2.0 GHz. The Intel Xeon CPU E5-2630 v3 ("Haswell-EP") processor on Marquez has multiple fully integrated voltage regulators providing an individual voltage for each core which results in per core P-states (PCPS) (Hackenberg, Schöne, Ilsche, Molka, Schuchart, and Geyer 2015). This enables independent frequency scaling "per core" rather than "per-socket". (Marquez is funded and operated by Old Dominion University.)
2. `Styx` has an Intel i5-4590 "Haswell" quad-core processor and 8 GB of main memory with timing specification 9-9-9-24. The processor frequency ranged from 3.3 GHz to 0.8 GHz; for memory, the frequency range was from 1.6 GHz to 0.8 GHz. The processor frequency was modified by writing a specific alpha-numeric value to model specific register (MSR) IA32_PERF_CTL.

The *effective frequency* f_{eff} for the two platforms can be expressed as,

1. For the PCPS level (as in Marquez),
$$f_{\text{eff}} = f_i \,. \tag{2}$$
2. For the socket level (as in Styx),
$$f_{\text{eff}} = \max(f_0, f_1, \ldots, f_{n-1}) \,. \tag{3}$$

The experiments were conducted on a single node each of Styx and Marquez. For measuring the node power consumption, a Wattsup power meter was used with a sampling rate of 1 Hz. The processor frequency was modified by writing a specific alpha-numeric value to model specific register (MSR) IA32_PERF_CTL. A performance loss of 10% was chosen for the *automatic* strategy. Table 1 depicts the value of the relevant parameters needed for Equation (1) for the two hardware platforms. An important point to notice here is the significantly lower value of memory latency and out-of-order overlap factor in case of Marquez compared to Styx which can potentially turn a memory intensive task on Styx to a compute intensive task on Marquez.

Table 1: Parameters for Marquez and Styx relevant for the automatic strategy.

	Marquez	Styx
OOO Overlap Factor-α	0.28	0.58
Memory Latency (in Cycles)-β	200	270
Processor Frequency Range	1.2-2.4 GHz	0.8-3.3 GHz
Memory Frequency Range	1.33-2.666 GHz	0.8-1.6 GHz

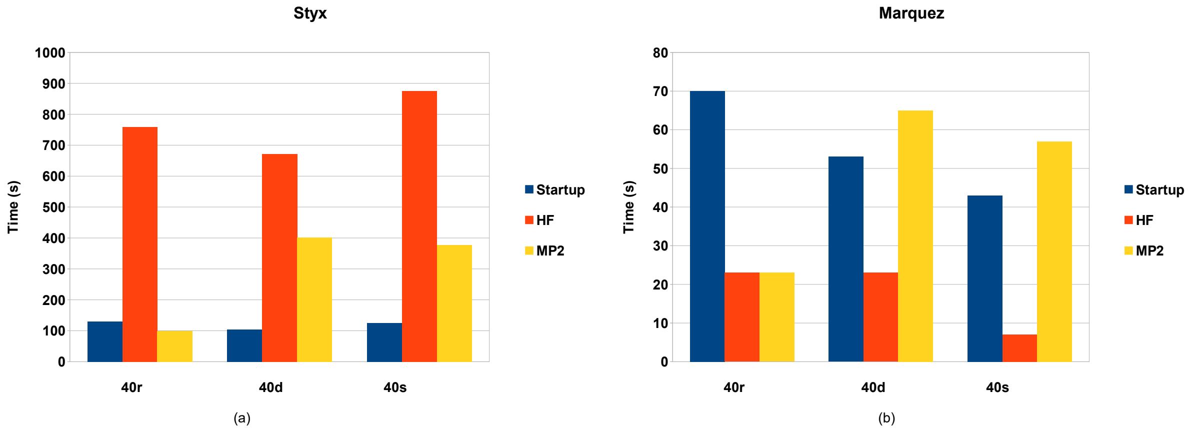


Figure 1: Execution time for different sections in the 40-atom system solved by three MP2 algorithms on (a) Styx and (b) Marquez, where 40r, 40d and 40s refer to the RIMP2, Direct and Semi-direct algorithms, respectively.

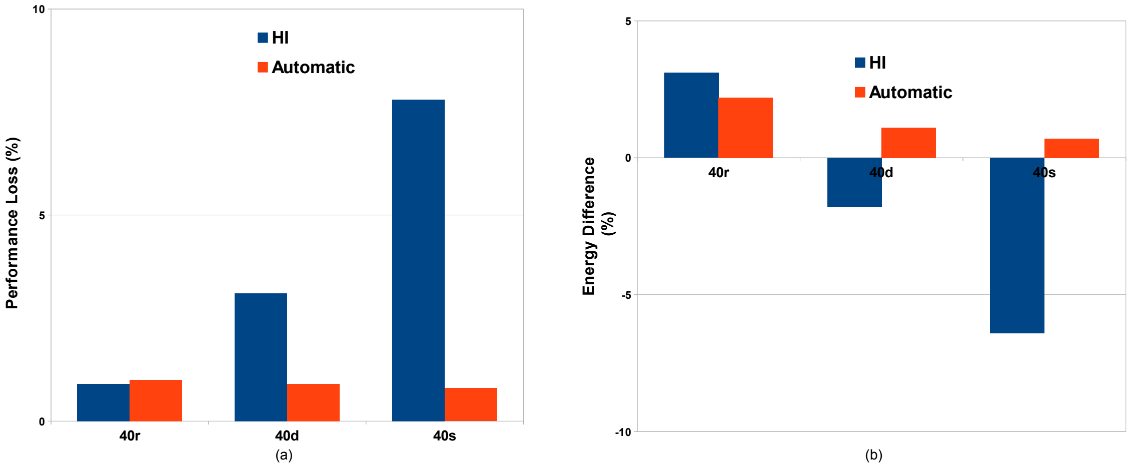
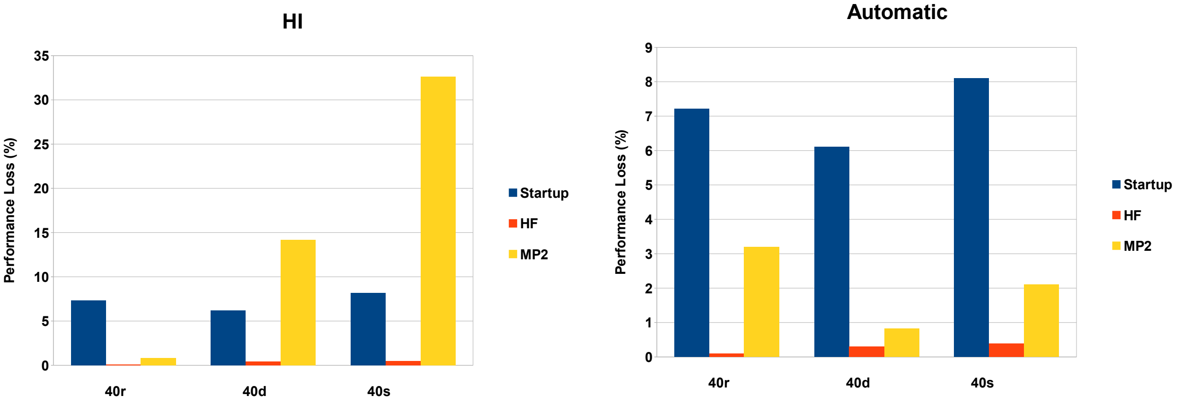


Figure 2: For Styx, total performance degradation (a) and energy difference (b) for the 40-atom system solved by the three MP2 algorithms under the HI and Automatic strategies.



(a)

(b)

Figure 3: For Styx, breakdown of performance degradation by section for the 40-atom system solved by three MP2 algorithms operating under (a) HI and (b) Automatic strategy.

5.1 Section Timings

Figure 1 depicts the execution time of different sections for the 40-atom system for the three algorithms executing on Styx (Fig. 1(a)) and Marquez (Fig. 1(b)). It can be observed that the startup section tends to take nearly the same time for all the three algorithms on Styx whereas on Marquez, it is variable. As for the HF section, it has the highest proportion of the execution time for all the inputs on Styx. On Marquez, the startup section dominates the execution time for the RIMP2 inputs, but in the case of direct and semi-direct, the HF section executes for slightly lower time than MP2 section. Considering the MP2 section, RIMP2 is the fastest of the three followed by semi-direct and direct.

5.2 Styx

Figure 2 shows the performance loss and change in energy consumption for the three MP2 algorithms when operated under the *HI* and *Automatic* strategies compared to the baseline case in which both PF and MF were set at their highest levels on *Styx*. The section wise performance loss for the 40-atom system solved by the three MP2 algorithms operating under the *HI* and *Automatic* strategies is provided in Fig. 3. It can be noted here that a negative value of energy difference denotes an overall increase in energy consumption.

In the case of RIMP2 algorithm, the *HI* and *Automatic* strategy both result in a moderate performance loss of $\sim$1%. The startup section in RIMP2 shows fairly compute intensive behavior enabling both *HI* and *Automatic* strategies to scale the memory frequency to its lowest value (0.8 GHz). During the HF section in RIMP2 algorithm, the *Automatic* strategy mostly executes at 2.8-3.0 GHz PF since the nature of the workload in terms of memory accesses per instruction remains quite variable with most of the time processor remaining in an idle state. During the MP2 stage in RIMP2, the *Automatic* strategy chooses MF of 1.066 GHz, compared to 0.8 GHz in the *HI* strategy.

For all the three algorithms, the HF section tends to be mostly I/O intensive with minor compute intensive phases in between. Therefore, reducing the PF during the HF section contributes to negligible overall performance degradation for the three algorithms operating under the *HI* strategy as noticed in Fig. 3(a). Since the behavior of HF section remains the same across the three algorithms in terms of workload, the *Automatic* strategy, ends up executing the HF mostly at the highest PF and MF and does not result in any

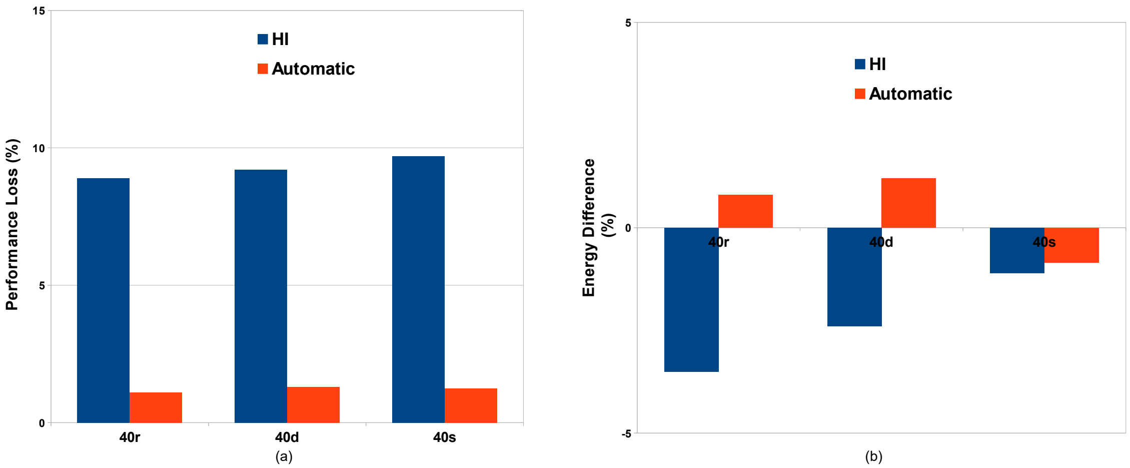


Figure 4: For Marquez: Total performance degradation (a) and energy difference (b) for the 40-atom system solved by the three MP2 algorithms under the HI and Automatic strategies.

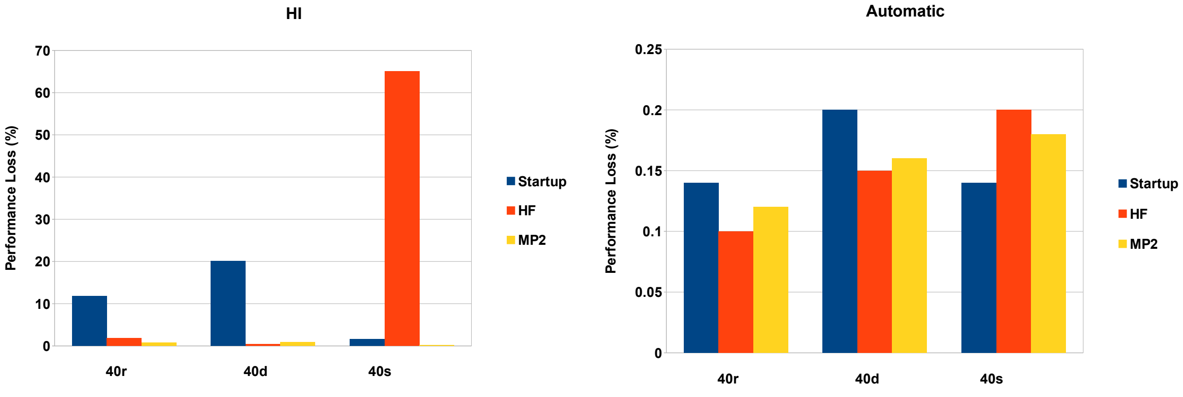
significant performance degradation. The MP2 section in RIMP2 tends to be much more compute intensive compared to direct and semi-direct MP2 and hence reducing the MF to its lowest value under *HI* results in negligible performance degradation. On the other hand, majority of the performance degradation for the overall direct and semi-direct execution is due to significant performance loss in the respective MP2 sections under the *HI* strategy. To summarize, average performance degradations across the three algorithms are 4% and 0.88% for the *HI* and *Automatic* strategies, respectively.

With regards to energy savings, the *HI* and *Automatic* strategies reduce the energy consumption by 3.1% and 2.2%, respectively for the RIMP2 algorithm. However, the direct(-1.8%) and semi-direct (-6.4%) algorithms experience an overall increase in energy consumption when operated under the *HI* strategy due to the fact that they don't offer neither substantially memory or CPU intensive behavior to benefit from either PF or MF, respectively. Also, under the *HI* strategy, the MF scaling applied to the MP2 section in direct and semi-direct algorithms degrades overall performance significantly. The *Automatic* strategy remains somewhat conservative with respect to frequency scaling throughout the direct and semi-direct execution and only applies MF of 1.33 GHz during the MP2 section. The average energy savings across the three algorithms are -1.8% and 1.33% for the *HI* and *Automatic* strategies, respectively.

The 40 atoms inputs behave largely similar to the 55 atoms system used in (Sundriyal, Fought, Sosonkina, and Windus 2016) in terms of relative timings of HF, SCF and MP2 sections and the effect of PF and MF on these timings as HF and MP2 sections tend to be compute intensive whereas SCF is largely memory/IO intensive. The difference mainly exists in the extent to which different sections in the three algorithms are affected by PF and MF. For example, the inability of *HI* strategy to provide energy savings for 40d and 40s inputs mainly comes from the resulting higher performance degradation through application of MF in the MP2 section, compared to the 55 atom inputs. At this point, the change in processor and memory intensity of these algorithms with varying system size is under investigation.

5.3 Marquez

Figure 4 depicts the performance loss and change in energy consumption for the three MP2 algorithms when operated under the *HI* and *Automatic* strategies compared to the baseline case in which both PF and MF were set at their highest levels on Marquez. The performance loss broken down for different sections of the 40-atom system execution solved by the three MP2 algorithms operating under the *HI* and *Automatic* strategies



(a)

(b)

Figure 5: For Marquez, breakdown of performance degradation by section for the 40-atom system solved by three MP2 algorithms operating under (a) HI and (b) Automatic strategy.

on Marquez is shown in Fig. 5. Due to a lower OOO overlap factor and higher speed and bandwidth provided by the DDR4 compared to the DDR3 on Styx, the three inputs tend to be relatively CPU intensive compared to their execution behavior on Styx. Consequently, the *HI* strategy results in considerable performance loss for all the three algorithms averaging 8.5%.

The *Automatic* strategy on the other hand, being aware of the change in platform and the relevant hardware parameters, executes the three inputs at the highest PF most of the time with execution dipping to 1.2 GHz PF in short bursts for direct and RIMP2 algorithms whereas the semi-direct algorithm tends to be the most compute intensive and no PF is applied to it. Also, no MF is applied to the three algorithms since the corresponding reduction in DRAM power is not enough to compensate for the performance loss incurred as per *Automatic* strategy. It can be seen in Fig. 5 that section-wise performance degradation is significantly lower for *Automatic* compared to *HI* and it mostly comes from profiling on Marquez. Therefore, the incurred performance loss for the *Automatic* strategy is much less compared to the *HI* for the three algorithms. More specifically, the average performance loss for the three algorithms is 1.2% when operated under the *Automatic* strategy.

Due to aggressive application of both PF and MF and the subsequent reduction in power which is insufficient to compensate for the performance loss, the *HI* strategy increases the energy consumption for all three algorithms, averaging -2.3%. The *Automatic* strategy on the other hand is extremely conservative in applying both PF and MF and saves energy for both RIMP2 (0.8%) and direct(1.2%). No frequency scaling is applied to the semi-direct algorithm under the *Automatic* strategy so there is a slight increase in energy consumption due to the minor profiling overhead.

6 CONCLUSIONS AND FUTURE WORK

This work studied the energy saving potential of the two variants of the DVFS based strategies evaluating three different algorithms in NWChem: RI-MP2, direct MP2, and semi-direct MP2, where the last two methods differ in the treatment of memory and disk storage of the integrals and intermediate data. The two DVFS based strategies were chosen such that one of them operates transparently to the application (automatic) and the other manages changes in the applications source code itself to insert calls to frequency scaling (HI).

The strategies were evaluated on two different platforms which differ in the processor generation, number of cores, processor frequency and amount and speed of memory. It was observed that the workload behavior of the three algorithms changed on the two platforms due to the difference in the hardware parameters and the *Automatic* strategy adapted to these changes in a better manner compared to *HI*. Overall, the *Automatic* strategy was the more conservative of the two in terms of minimizing performance degradation whereas the *HI* strategy in some cases ended up saving more energy albeit at the cost of application performance. In fact, when calculating the 40-atom system, NWChem doesn't provide enough opportunity to apply frequency scaling on the two platforms. So, even employing an *HI* moderate strategy, would not make much of a difference except it would only decrease the increase in energy savings noticed with the aggressive strategy by a bit. Since NWChem is one of the prominent quantum chemistry applications and enjoys a large user community, the findings of this paper are beneficial to an important scientific domain (ab initio quantum chemistry), which is a focus of a large number of HPC packages, such as GAMESS (Schmidt, Baldridge, Boatz, Elbert, Gordon, Jensen, Koseki, Matsunaga, Nguyen, Su, Windus, Dupuis, and Montgomery 1993) and PSI4 (Turney, Simmonett, Parrish, Hohenstein, Evangelista, Fermann, Mintz, Burns, Wilke, Abrams, Russ, Leininger, Janssen, Seidl, Allen, Schaefer, King, Valeev, Sherrill, and Crawford 2012), each having computational stages as considered in this paper.

Each of the two strategies that have been studied in this work have their own pros and cons. While the *HI* strategy is relatively easy to deploy and apply, it suffers from the fact that it is not aware of the nature of the underlying platform since the extent of application of frequency scaling is predetermined. On the other hand, the *Automatic* strategy constantly profiles the application to collect runtime information used in the decision making for frequency scaling but it can be difficult to deploy and can also suffer when the underlying hardware platform is changed since it relies on hardware performance counters. Future work would focus on getting the best of both worlds by combining these two strategies into a hybrid strategy. The hybrid strategy would consist of relevant hardware parameter information of a base platform. Depending on the platform on which NWChem would be executing, the hybrid strategy would adjust the frequency levels for the three sections by comparing the relevant hardware parameters of the base platform to the current platform. In this manner, appropriate frequencies would be chosen for the startup, HF and MP2 sections of the code without actually transparently profiling the application.

REFERENCES

Bernholdt, D. E., and R. J. Harrison. 1996. "Large-scale Correlated Electronic Structure Calculations: The RI-MP2 Method on Parallel Computers". *Chem. Phys. Lett.* vol. 250, pp. 477–484.

David, H., C. Fallin, E. Gorbatov, U. Hanebutte, and O. Mutlu. 2011. "Memory Power Management via Dynamic Voltage/Frequency Scaling". In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, pp. 31–40.

Feyereisen, M., G. Fitzgerald, and A. Komornicki. 1993. "Use of Approximate Integrals in ab initio Theory. An Application in MP2 Energy Calculations". *Chem. Phys. Lett.* vol. 208, pp. 359.

Freeh, V., and D. Lowenthal. 2005. "Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster". In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 164–173.

Ge, R., X. Feng, W. Feng, and K. Cameron. 2007, Sep.. "CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters". In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pp. 18.

Ge, R., X. Feng, Y. He, and P. Zou. 2016, Aug. "The Case for Cross-Component Power Coordination on Power Bounded Systems". In *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 516–525.

Ge, R., X. Feng, S. Song, H. Chang, D. Li, and K. Cameron. 2010. "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications". *Parallel and Distributed Systems, IEEE Transactions on* vol. 21, pp. 658–671.

Hackenberg, D., R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. 2015, May. "An Energy Efficiency Feature Survey of the Intel Haswell Processor". In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 896–904.

Hsu, C., and W. Feng. 2005, nov. "A Power-Aware Run-Time System for High-Performance Computing". In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 1.

Huang, S., and W. Feng. 2009, May. "Energy-Efficient Cluster Computing via Accurate Workload Characterization". In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pp. 68–75.

Iancu, C., S. Hofmeyr, F. Blagojevic, and Y. Zheng. 2010. "Oversubscription on Multicore Processors". In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–11.

Knowles, P., J. Andrews, R. Amos, N. Handy, and J. Pople. 1991. "Restricted Møller—Plesset Theory for Open-Shell Molecules". *Chem. Phys. Lett.* vol. 186, pp. 130–136.

Lauderdale, W., J. Stanton, J. Gauss, J. Watts, and R. Bartlett. 1991. "Many-body Perturbation Theory with a Restricted Open-shell Hartree—Fock Reference". *Chem. Phys. Lett.* vol. 187, pp. 21–28.

Lim, M., V. Freeh, and D. Lowenthal. 2006. "Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs". In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*.

Park, J., D. Shin, N. Chang, and M. Pedram. 2010. "Accurate Modeling and Calculation of Delay and Energy Overheads of Dynamic Voltage Scaling in Modern High-Performance Microprocessors". In *2010 International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 419–424.

Schmidt, M. W., K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, and J. J. Montgomery. 1993, November. "General atomic and molecular electronic structure system". *J. Comput. Chem.* vol. 14, pp. 1347–1363.

Sensi, D. D., M. Torquati, and M. Danelutto. 2016, December. "A Reconfiguration Algorithm for Power-Aware Parallel Applications". *ACM Trans. Archit. Code Optim.* vol. 13 (4), pp. 43:1–43:25.

Sundriyal, V., E. Fought, M. Sosonkina, and T. L. Windus. 2016. "Power Profiling and Evaluating the Effect of Frequency Scaling on NWChem". In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, pp. 19:1–19:8. San Diego, CA, USA, Society for Computer Simulation International.

Sundriyal, V., and M. Sosonkina. 2016a. "Effect of Frequency Scaling Granularity on Energy-Saving Strategies". *Submitted to the International Journal of High Performance Computing Applications.*.

Sundriyal, V., and M. Sosonkina. 2016b. "Joint Frequency Scaling of Processor and DRAM". *The Journal of Supercomputing* vol. 72 (4), pp. 1549–1569.

Turney, J., A. Simmonett, R. Parrish, E. Hohenstein, F. Evangelista, J. Fermann, B. Mintz, L. Burns, J. Wilke, M. Abrams, N. Russ, M. Leininger, C. Janssen, E. Seidl, W. Allen, H. Schaefer, R. King, E. Valeev, C. Sherrill, and T. Crawford. 2012. "Psi4: An Open-source ab initio Electronic Structure Program". *Wiley Interdisciplinary Reviews: Computational Molecular Science* vol. 2 (4), pp. 556–565.

Valiev, M., E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong. 2010. "NWChem: A Comprehensive and Scalable Open-source Solution for Large Scale Molecular Simulations". *Computer Physics Communications* vol. 181 (9), pp. 1477 – 1489.

Vishnu, A., S. Song, A. Marquez, K. Barker, D. Kerbyson, K. Cameron, and P. Balaji. 2010. “Designing Energy Efficient Communication Runtime Systems for Data Centric Programming Models”. In *Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM ’10, pp. 229–236. Washington, DC, USA, IEEE Computer Society.

AUTHOR BIOGRAPHIES

VAIBHAV SUNDRIYAL is working as a Research Scientist at ODU Research Foundation. He holds a PhD in Computer Engineering from Iowa State University. His research interests lie in computer architecture and power management especially in HPC systems. His email address is vsundriy@odu.edu.

ELLIE FOUGHT is a graduate student in Department of Chemistry at Iowa State University working towards her Ph.D. She is working on incorporating power considerations when optimizing quantum mechanical algorithms. She has also examined NMR properties for organic complexes. Her email address is foughtel@iastate.edu.

MASHA SOSONKINA received her Ph.D. from Virginia Tech, and currently is Professor in the Department of Modeling, Simulation and Visualization Engineering at the Old Dominion University. Her research interests include applied computational mathematics, high performance computing and chemical and biological sciences. Her email address is msosonki@odu.edu.

THERESA L. WINDUS is a Professor in the Department of Chemistry at Iowa State University. She earned her Ph.D. from Iowa State University in 1993 and did post-doctoral research at Northwestern University. Her research interests include high performance computing and methods for accurate energies. Her email address is twindus@iastate.edu.