Development and Performance Analysis of Real-World Applications for Distributed and Parallel Architectures *

T. Fahringer† P. Blaha∗ A. Hössinger∗∗ J. Luitz∗ E. Mehofer† H. Moritsch‡ B. Scholz†

†Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstrasse 22, A-1092, Vienna, Austria
[tf,mehofer,scholz]@par.univie.ac.at

‡Department of Business, University of Vienna
Brünner Strasse 72, A-1210 Vienna, Austria
moritsch@finance2.bwl.univie.ac.at

∗Institute of Physical and Theoretical Chemistry, Vienna University of Technology
Getreidemarkt 9/156, A-1060 Vienna, Austria
[pblaha,luitz]@homer.theochem.tuwien.ac.at

∗∗Institute for Microelectronics, Vienna University of Technology
Gusshausstr. 27-29/E 360, A-1040 Vienna, Austria
hoessinger@iue.tuwien.ac.at

Submitted to *Concurrency: Practice and Experience, John Wiley & Sons, July, 1999*

Abstract

Several large real-world applications have been developed for distributed and parallel architectures. We examine two different program development approaches: First, the usage of a high-level programming paradigm which reduces the time to create a parallel program dramatically but sometimes at the cost of a reduced performance. A source-to-source compiler, has been employed to automatically compile programs – written in a high-level programming paradigm – into message passing codes. Second, manual program development by using a low-level programming paradigm – such as message passing – enables the programmer to fully exploit a given architecture at the cost of a time-consuming and error-prone effort.

Performance tools play a central role to support the performance-oriented development of applications for distributed and parallel architectures. Scala – a portable instrumentation, measurement, and post-execution performance analysis system for distributed and parallel programs – has been used to analyse and to guide the application development by selectively instrumenting and measuring the code versions, by comparing performance information of several program executions, by computing a variety of important performance metrics, by detecting performance bottlenecks, and by relating performance information back to the input program. Experiments are shown for a NEC Cenju-4 distributed memory machine and a cluster of heterogeneous workstations and networks.

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

1 Introduction

Performance-oriented development of efficient programs for distributed and parallel systems is an error-prone and time-consuming process that may involve many cycles of code editing, compiling, executing, and performance analysing. Many different programming paradigms such as explicit message passing [15], High Performance Fortran (HPF) [20], OpenMP [9], Java RMI [39], and HPC++ [26] have been introduced for distributed and parallel architectures. A trade-off is implied by the programming paradigm employed. On the one hand, programming at a low level (i.e., message passing paradigm) enables the programmer to fully exploit and control the features of a specific architecture at the cost of a very time-consuming and error-prone programming effort. On the other hand, choosing a high-level programming paradigm can reduce the program development effort dramatically, however, sometimes at the cost of a reduced performance. There is a large variety of reasons that can cause performance losses in distributed or parallel programs. For instance, ineffective data and work parallelism, uneven load balance, compiler organization overhead, ineffective memory access behavior (i.e., cache), and high communication, synchronization and input/output overhead. The source for these performance bottlenecks can frequently be related to the intricate structure and details of an application program, to the code transformation system, or to the target architecture.

Performance tools play a crucial role to support the performance-oriented development of applications for distributed and parallel architectures by locating performance problems and mapping them back to the input program. Many existing performance monitoring and analysis systems collect and present performance data for programs that have been generated and modified by transformation systems without the possibility to relate performance data back to the input program. A performance system must have access to transformation systems in order to record code changes and associate performance problems with the input program. Commonly, performance information is provided for low-level system calls (i.e., operating and runtime library calls) that cannot be mapped to specific locations in the input program. There are performance tools that provide only summary information for entire programs without relation to specific program points or regions of interest. Crucial correlation of performance bottlenecks with exact positions in the input program is disabled which severely restricts the usefulness of such tools. Finally, many performance tools are restricted to a specific programming paradigm.

Scala [14, 38, 36, 37] is an instrumentation, measurement, and post-execution performance analysis system for distributed and parallel programs that combines a portable instrumentation system, performance data correlation, data management and measurement analysis, and an interface for performance visualization. Scala can be used to monitor and analyze the performance of many different programming paradigms ranging from high-level (i.e. High Performance Fortran) to low-level programs including data parallel, task parallel and message passing programs. Various instrumentation features are supported that enable both comprehensive and selective monitoring in order to control the monitoring overhead and the performance data generated. Performance data correlation maintains the performance relationship between the input program and code changes applied by transformation systems. Data management and measurement analysis supports a rich set of performance data reduction, filtering, summary, and analysis techniques. Many performance metrics and statistics can be computed. Scala supports several trace formats which allows the use of various visualization systems (e.g. Medea [6], and Upshot [19]). Finally, Scala has highly sophisticated scalability analysis integrated, that examines the scaling behavior of a program for varying input data and machine sizes.

This paper describes the performance-oriented development of real-world applications for distributed and parallel architectures. Scala has been used to analyse and to guide the application development by selectively instrumenting and measuring the code versions, by comparing performance information of several program executions, by computing a variety of important performance metrics, by detecting performance bottlenecks, and by relating performance information back to the input program. We used the following three applications: (1) a Monte-Carlo ion implementation simulator for three-dimensional crystalline structures [22, 5] developed by Prof. Selberherr and his associates at the Vienna University of Technology; (2) a system for pricing of financial derivatives [10] developed by Prof. Dockners' group at the University of Vienna; and (3) WIEN97 [3], a system for quantum mechanical calculations of solids developed by Prof. Schwarz and his group at the Vienna University of Technology.

The ion implementation simulator is a heterogeneous code which comprises both Fortran and C code segments. A master-slave model has been manually implemented as an MPI message passing program that exploits both data and task parallelism and runs on a cluster of heterogeneous workstations and networks. The pricing system for financial derivatives has been developed from scratch as an HPF/Fortran90 program that uses data parallelism. WIEN97 has been parallelized by employing HPF/Fortran77 which benefits from data parallelism. Whereas the pricing system has been developed from scratch with the aim to carefully uncover and exploit parallelism, the other two applications have been parallelized based on existing sequential codes. The pricing system and WIEN97 have been parallelized by using VFC [2] – a compiler that translates HPF programs into message passing programs (Fortran90 with MPI message passing calls) – and executed on a NEC Cenju-4 [32] distributed memory parallel machine.

In the next Section, we give an overview of Scala as an integrated system of a restructuring and optimizing compiler. Section 3 to 5 describe the three applications and show how they have been parallelized for distributed and parallel architectures. We demonstrate the usefulness of Scala to instrument, monitor, and analyze the performance for the application codes. Experiments are shown for a NEC Cenju-4 distributed memory machine and a cluster of heterogeneous workstations and networks. Related work is discussed in Section 6. Summary and concluding remarks are are given in Section 7.

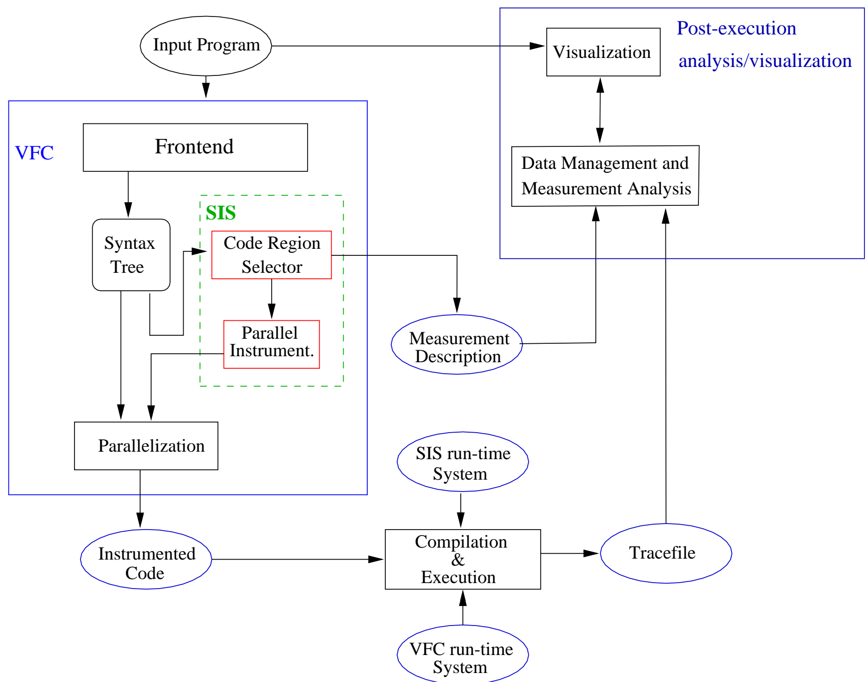

Figure 1. Execution-Driven Performance Analysis System

2 Scala

Scala is a post-execution performance system that instruments, measures, and analyses the behavior of distributed and parallel programs. The architecture of Scala is based on a portable instrumentation system, runtime-libraries that collect performance data during program execution, and post-execution performance analysis that computes various performance metrics and relates them back to the input program. In addition, Scala supports several interfaces to visualization systems. Although Scala has been integrated with an existing compiler it can be easily ported to front-ends and compilers for other programming languages and architectures by porting its instrumentation and runtime libraries.

Figure 1 shows the architecture of Scala as an integrated system of VFC [2] which is a compiler that translates Fortran programs into message passing programs (Fortran90 with MPI message passing calls). The input programs (Fortran77, Fortran90, HPF, and explicit message passing programs) of Scala are processed by the compiler front-end which generates an abstract syntax tree (AST). The Scala Instrumentation System (SIS) enables the user to select (by directives or command-line options) code regions of interest. Based on the selected code regions, SIS automatically inserts monitoring code in the AST which will collect all relevant performance information during execution of the program. SIS also generates a measurement description file that enables relating all gathered performance data back to the input program. This is a crucial aspect of Scala, as instrumentation may be done at a different level (e.g. message passing program) than the original input program (e.g. HPF). Then the compiler generates an instrumented distributed or parallel program which will be executed on the target architecture. Note that the compiler can also process explicitly distributed and parallel programs for instrumentation and performance analysis. During execution all relevant performance data is collected in a trace-file.

The trace-file provides a generic input for a post-mortem data management and measurement analysis to reduce, filter, summarize, and analyse performance information. Among others, a variety of performance metrics are computed which

includes speedup, efficiency, communication, and work distribution. Several interfaces for visualization systems have been developed in order to graphically display various performance statistics and profiles that can be shown together with the original input program.

The general structure of Scala comprises several modules which combined together provide a robust environment for advanced performance analysis:

- Scala Instrumentation System (SIS)
- Performance data correlation
- Data management and measurement analysis
- Performance visualization interface

In what follows we give a brief overview of each of these modules. A detailed description of Scala and its functionality can be found in [14, 38, 36, 37]

2.1 Scala Instrumentation System

Based on command-line options or user provided directives, SIS inserts instrumentation code in the program for each information of interest which includes: timing events, execution frequency events, values for program unknowns (unknowns in array subscript expressions, loop bounds, etc.), and array information (rank, shape, alignment, distribution, mapping, etc.). SIS supports the programmer to control monitoring and generating performance data through selective instrumentation of specific types of code regions (i.e., program, procedures, loops, communication, and I/O operations). SIS also enables instrumentation of arbitrary code regions through explicit instrumentation of all entry and exit points of code regions. Finally, instrumentation can be turned on and off by a specific instrumentation directive.

2.2 Performance Data Correlation

A crucial aspect of performance analysis is to relate performance information back to the original input program. A compiler may imply many code changes (e.g. copying, hoisting and sinking of code sections) so that the relationship between its execution dynamics and its input program is obscure. For instance, irregular programs are frequently optimized based on the inspector/executor paradigm [2] which can cause loops to be transformed into a preparation (inspector) and an execution (executor) phase. In order to examine which performance aspect corresponds to what code region, a *measurement description file* is generated and updated while the compiler is applying code transformations. For instance, records are inserted in the measurement description file that link every irregular loop with its associated inspector and executor phase.

2.3 Data management and measurement analysis

The data collected by Scala need to be analyzed and reduced so that it can be sufficiently detailed for analyzing the behavior of parallel and distributed programs. The *data management and statistical analysis* module (see Figure 1) implements several data-reduction techniques that can be applied in isolation or in combination for data management. Extensive performance data filtering is provided through a range of performance metrics from aggregate (i.e., entire program) through procedure, procedure calls, loops, arbitrary code sections, and individual source code lines. Among others, a variety of performance metrics are computed which includes speedup, efficiency, communication, and work distribution. Moreover, the coefficients of variation of communication time and computation time are good metrics to express the quality of work and communication distributions across processors. Performance metrics for program regions can be split into individual performance components (for instance, computation, communication, idle, and synchronization time). Statistical methods such as the analysis of distribution and variability provide more detailed information on the dynamic behavior of distributed and parallel programs. For example, the analysis of distribution gives information on how the communication and computation are distributed across processors and the user or compiler can apply transformations to improve the performance.

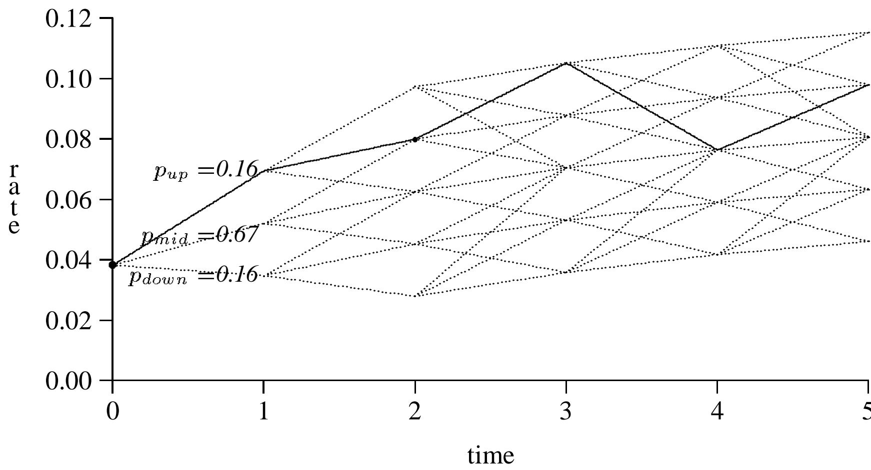

Figure 2. Hull and White tree for the Δt spotrate with selected path

2.4 Performance Visualization

Visualization of performance metrics and statistics, and also dynamic information about arrays is of crucial importance to support the programmer in performance tuning of distributed and parallel programs. Scala supports two trace formats – ALOG and Scala specific format – for collected performance data which enables the usage of several performance visualization systems. Based on the ALOG trace format we can use visualization systems of well-established systems such as Medea [6], TAU [31], and Upshot [19]. Medea is a post-mortem performance analysis and visualization system. Among others we use Medea to derive and visualize performance metrics together with the input program based on ALOG trace files. This work has been described in detail in [7].

Scala also generates Grace [17] data files for various 2D performance data visualizations. A Scala specific trace format is used as input to our own graphical user interface. This interface has been developed based on an HTML display engine which enables the user to view the input program together with performance parameters and dynamic array information (see Section 2.2).

3 Pricing of Financial Derivatives

The pricing of derivate products is an important field in finance theory. A *derivative* (or *derivative security*) is a financial instrument whose value depends on other, so called underlying securities [23]. Examples are stock options and variable coupon bonds, the latter paying interest rate dependent coupons. The pricing problem can be stated as follows: what is the price today of an instrument which will pay some cash flows in the future, depending on the development of an underlying, e.g. stock prices or interest rates ? For simple cases analytical formulas are available, but for a range of products, whose cash flows depend on a value of a financial variable in the past - so called *path dependent* products - Monte Carlo simulation techniques have to be applied [33],[28]. By utilizing massively parallel architectures very efficient implementations can be achieved [25],[41]. For a detailed description of the technique implemented see [11] and [18].

The Monte Carlo simulation is based on a discrete representation of a stochastic process that describes the dynamics of the underlying security over time [24]. In the case of interest rate dependent products, the *Hull and White tree* describes the future development of the short rate, which is used to calculate the entire interest rate curve for a specific state of the system [23]. Each state is represented by a node in a directed graph and has three successor nodes, representing increasing, constant, and decreasing interest rates. Nodes are described by (time, interest rate) pairs. Arcs are labeled with the transition probabilities $p_{up}, p_{mid}, p_{down}$. A state can be reached by more than one predecessor; this *recombining* property establishes a lattice structure. Figure 2 shows a Hull and White tree with time (i.e. in years) on the horizontal and interest rates on the vertical axis.

To price interest rate dependent products the interest rate tree is used either to solve it backwards in time or by simulating paths through the tree and averaging the corresponding prices. The Monte Carlo Simulation algorithm selects a number of N paths in the Hull and White tree from the root node to some final node (see Figure 2). Along each path, it iteratively discounts, backwards from the final node to the root node, the cash flow generated by the instrument along this path. For

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE (BLOCK) ONTO PR :: VALUE
...
   TYPE(BOND) :: B                                              ! the bond to be priced
   INTEGER :: PATH(0:N_STEPS)                                   ! path in the Hull and White tree
   REAL(DBLE) :: VALUE(1:N)                                     ! all path results

   !HPF$ INDEPENDENT, NEW(PATH), ON HOME(VALUE(I))
   DO I = 1, N
     PATH = RANDOM_PATH(0,0,N)                                  ! select a path starting at node (0,0)
                                                                ! discount the bond's cashflow to time 0
     VALUE(I) = DISCOUNT(0,CASH_FLOW(B,1,N),FACTORS_AT(PATH))
   END DO
   PRICE = SUM(VALUE)/N                                         ! mean value
...
```

Figure 3. HPF DO-Independent Code of the Pricing System

variable coupon bonds, the cash flows are path dependent, i.e. depend on the interest rates at predecessor nodes. Discounting is performed using the interest rates along this path. The resulting price of the instrument is the mean value over all selected paths. The HPF/Fortran90 code segment in Figure 3 shows the main loop of the simulation procedure TRAVERSE_DISCOUNT .

We extend the procdure in order to price also bonds with embedded options (callable or putable bonds): early exercise takes place, when the present value of future cash flows is greater (respectively less, in the case of putable bonds) than the exercise value. The effect of early exercise is modeled by a modification of the cash flows: the cashflow at the early exercise time is set to the principal payment value, and the cash flows after that time are set to zero.

For the modeling of the early exercise decision at a node k, we perform a nested Monte Carlo simulation, which samples paths in the "subtree" beginning with k. Thus we gain two levels of simulation [18]. At the first level *main* paths, starting at the root node are processed. At the second level, for each node in a main path, a number of *subpaths*, emanating from this node, is selected. Discounting along the subpaths is performed to compute the early exercise decision.

At both levels the same recursive simulation procedure TRAVERSE_DISCOUNT is used. During the main simulation, at the first level, it calls an extended DISCOUNT function, which invokes TRAVERSE_DISCOUNT to perform the nested simulation as shown in Figure 3. At this level the early exercise is handled by the standard DISCOUNT function, i.e. without further recursion.

3.1 Parallelization

During the simulation, the information at the tree nodes is potentially used the by the computation of every path. This motivates a replication of the whole tree over all processors. The storage requirements for these structures are comparatively small and not critical in terms of local memory size.

Sampling as well discounting along the paths can be done in parallel. Because all the path computations are independent from each other, they can be performed without communication. Every path computation has access to the whole tree data. After processing the individual paths, the final price is computed via a summation of the path results over all processors. A reduction operation is used, which first computes partial sums on each processor simultanously, and then sends the partial results to a selected processor which computes the final sum. This is the only operation which requires communication.

We encoded three different versions of the pricing system and executed them on a NEC Cenju-4 [32] distributed memory parallel machine. First, a data parallel version which is based on distributing array VALUE block-wise onto the maximum number of processors – by using the HPF intrinsic function NUMBER_OF_PROCESSORS() – that are available on a given architecture. The summation of the path results is replicated which causes communication. Second, we improved the first version by using the HPF reduction directive, which causes the summation of the path results to be executed by an efficient machine function. Third, the second version is improved by using the HPF DO-Independent directive, that specifies that each iteration of the main simulation loop can be executed simultaneously. Every iteration of the simulation loop is executed by the processor that owns array element VALUE(I) based on the owner-computes paradigm [2].

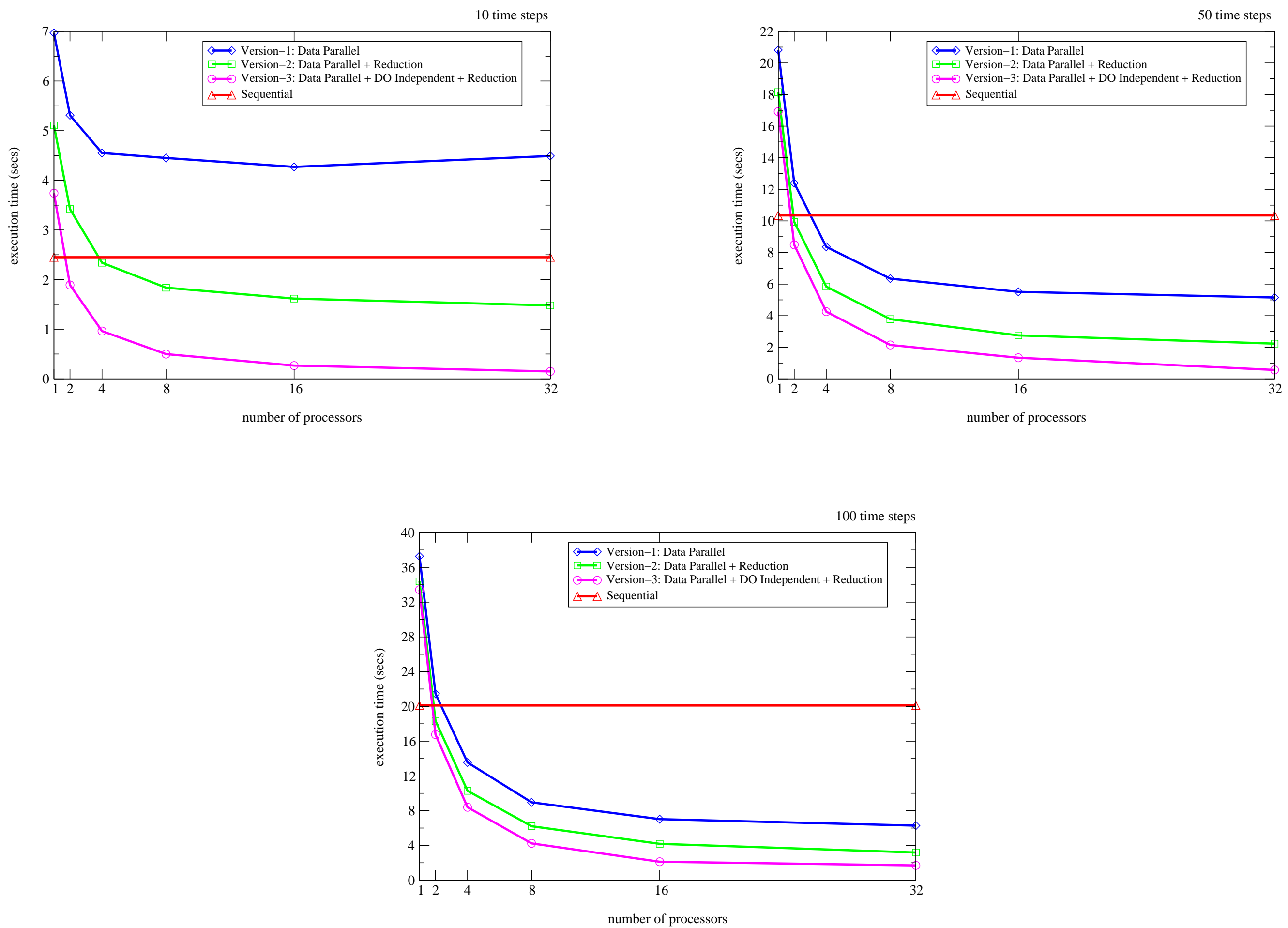

Figure 4. Measured accumulated execution times for all three versions of function TRAVERS_DISCOUNT of the system for pricing of financial derivatives (three HPF and one sequential version). Experiments have been conducted for three different problem sizes (number of time steps) and varying number of processors on a NEC Cenju-4

3.2 Experimental results and further work

VFC has been used to generate Fortran90/MPI programs based on input HPF programs. Each graph in Figure 4 shows the corresponding accumulated execution times for function TRAVERSE_DISCOUNT for all three versions including the sequential implementation of a specific problem size (number of time steps). Note that the sequential execution corresponds to the execution on a single processor. The experiments clearly show that version-3 is superior to all other versions, and version-2 is better than version-1 for all problem and machine sizes. Version-3 exploits more parallelism than version-2 due to the fact that a processor only executes a loop iteration if it owns the corresponding array element (VALUE(I)). Whereas for version-2 every processor executes all loop iterations. The array assignment, however, is only executed if a processor owns VALUE(I). Version-1 sequentializes the reduction operation which causes the largest communication overhead across all code versions. For the smallest problem size (10 time steps), version-1 performs worse than even the sequential version. For increasing problem sizes, however, the difference among the code versions becomes less dramatic, as the impact of communication on the overall performance diminishes. Version-3 shows almost linear speed-up for up to 8 processors. Larger problem sizes should cause a better performance scaling behavior for increasing number of processors.

Figure 5 shows several snapshots of the MEDEA system that visualize various performance metrics together with the input program. Note that all performance measurements have been done based on the generated Fortran90/MPI program,

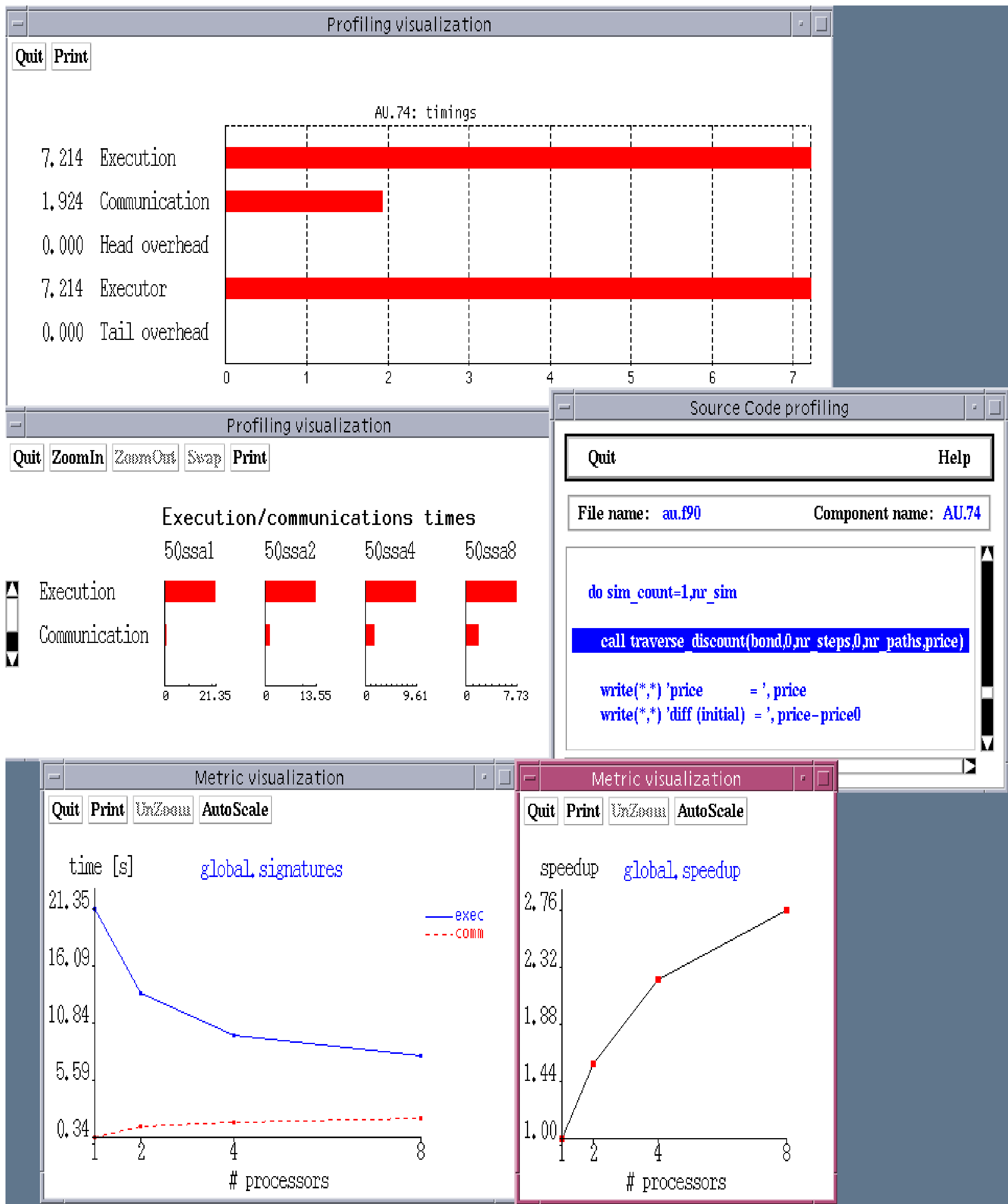

Figure 5. Snapshots of the MEDEA system which displays various performance metrics together with the code section of interest (call to function *TRAVERSE_DISCOUNT* in middle-right window).

whereas the performance metrics are displayed together with the input HPF program (see middle-right window). The upper window displays how much of the execution time (of a 8 processor version) of the call to *TRAVERSE_DISCOUNT* accounts for communication, for executing the main simulation loop in *TRAVERSE_DISCOUNT*, and for compiler overhead (head/tail) before and after the call statement. Note that communication time is part of the execution time. Furthermore, the entire communication – caused by the reduction operation – is spent in the main simulation loop of code version-3. The time for the executor corresponds to the simulation loop and the reduction operation as well. The middle-left window shows how long it took to execute the call to *TRAVERSE_DISCOUNT* and how much has been spend in communication for 1, 2, 4 and 8 processors. The lower-left and lower-right windows, respectively, show the execution signature and speedup of the entire application code for various number of processors.

The parallel simulation algorithm can cause redundant price computations due to the properties of the Hull and White tree. We plan to implement an optimized version that avoids redundant price computations by having one processor compute prices and broadcast the result to all processor that need this data. This procedure implies some extra communication, however, may save substantial computation time.

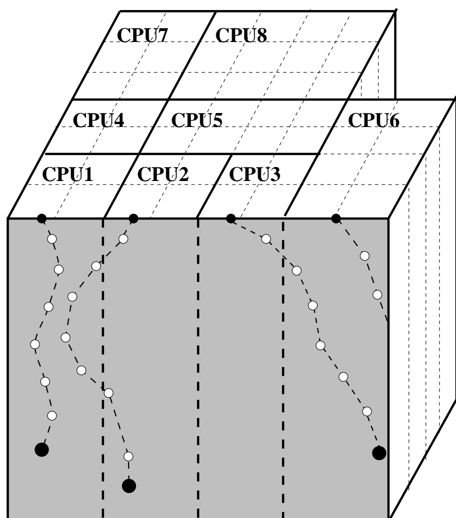

Figure 6. Schematic presentation of the distribution of the geometry among several processors

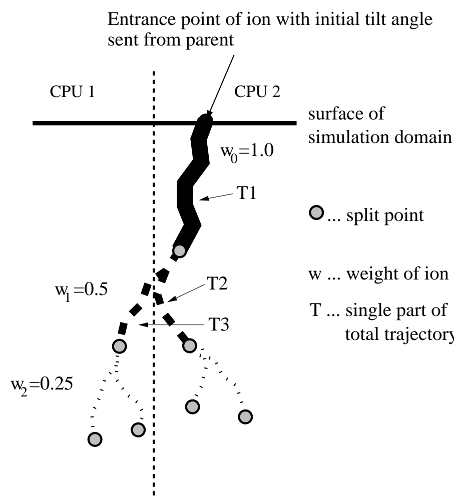

Figure 7. Schematic presentation, how the trajectory-split method is used in the parallelized Monte-Carlo ion implantation simulator

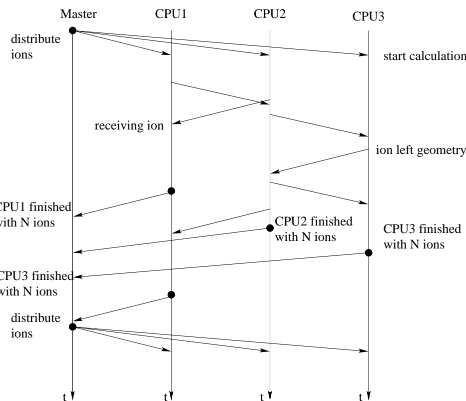

Figure 8. Communication among master and slaves (CPU1 - CPU3); t is the time-axis.

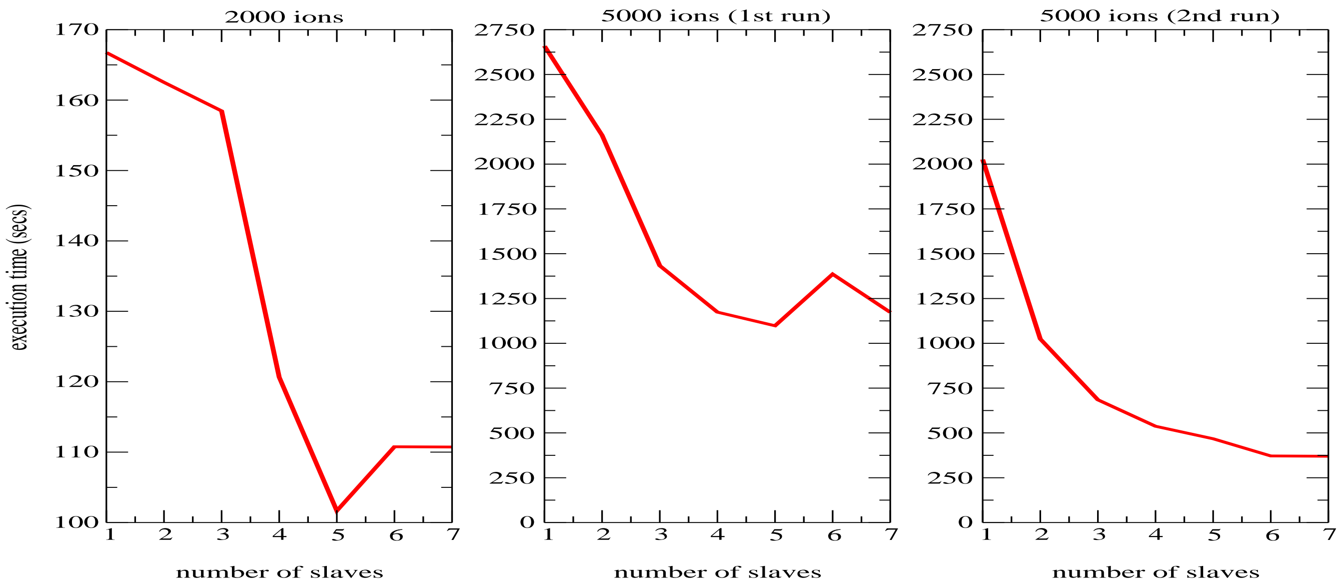

Figure 9. Execution times of the ion implantation simulator for different problem sizes and varying number of slave processors on a heterogeneous NOW

4 Ion Implantation Simulator for Three-Dimensional Crystalline Structures

When simulating semiconductor production processes, ion implantation is a very important, but also one of the most critical steps concerning the simulation time. Due to the complicated structures and the small dimensions of modern semiconductor devices, Monte-Carlo simulation methods often have to be used to describe non-planarity effects and phenomena resulting from ion channelling and large tilt angles. To reach the expected accuracy, three-dimensional simulations have to be performed with very sophisticated models [22], especially for very shallow implantation conditions. Simulations can take up several days and even weeks for realistic problem sizes which made it a first-order target for parallelization. As a serial code for the ion implantation simulator existed we followed a parallelization strategy that modified the original code and computation models [21, 22, 5] as little as possible by isolating communication and synchronization in a few routines.

The Monte-Carlo ion implantation simulation method is based on the concept that trajectories of a large number of ions – entering the device structure equally distributed over the device surface – are calculated. The distribution of the dopants and crystal damage are derived from the final positions of the ions and the generated point-defects. In order to take the influence of crystal damage on the trajectory of an ion into account a transient simulation is introduced. The ions belonging to the same time step are equally distributed over the device surface and do not interact with each other.

4.1 Parallelization

We parallelized the ion implantation simulation by distributing the geometry of the simulation domain based on a master-slave computational model. The bounding box of the simulated structure is split into small rectangular prisms. At the beginning of each time step the master processor distributes one or several prisms to a set of slave processors as shown in Figure 6. Each processor is responsible to calculate the trajectories of all ions residing inside of its assigned prisms. The ion trajectories are computed sequentially by each processor. Parallelism is exploited as all slaves can execute their ion trajectories simultaneously. The order of computing ion trajectories is arbitrary under the restriction that no damage accumulation occurs which is a very reasonable assumption. Both ions that enter through the surface as well as newly created particles can move inside of the simulation domain. The proposed master-slave parallelization method inherently models both cases without restriction. The master communicates with the slaves to inform them about a new time step of the simulation, and to maintain a record with the number of ions that reside in the geometry domain of every specific slave. The slaves communicate among each other by exchanging ions that cross the slave's geometry domain (see Figures 6 and 8). Currently, we use a static load balancing strategy based on information about the computational capabilities of each processor. For instance, if a processor

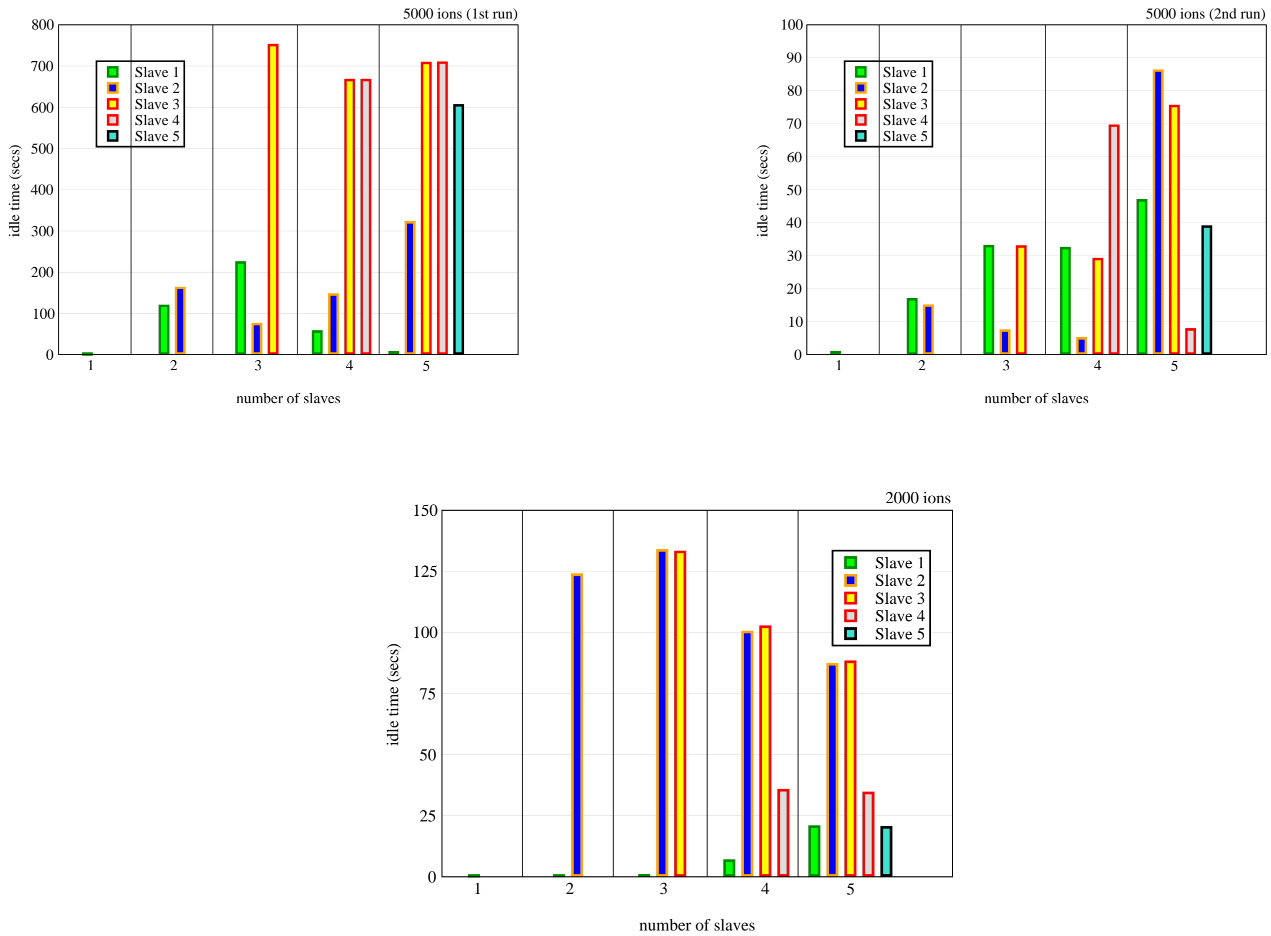

Figure 10. Idle times for the ion implantation simulator for two different problem sizes (2000 and 5000 ions) for varying number of slave nodes on a heterogeneous NOW.

P_i is twice as fast as another processor P_j then P_i gets a workload (number of ion trajectories) assigned that is twice as large than that of P_j. In order to optimize the performance of our master-slave method based on a static load balancing which is executed on a dedicated distributed or parallel architecture (exclusively used by our application), the following two constraints should be considered:

$$\sum_i \frac{O_i}{V_i} \rightarrow max \tag{1}$$

$$\frac{V_i}{CPU_i} \simeq const., \forall i \tag{2}$$

V_i, O_i, and CPU_i (e.g., floating point operations per second) are, respectively, the volume, the surface of the prismatic area, and the relative computing capability of a slave i.

The sequential version of the ion implantation simulator is composed of Fortran and C code components. Our parallel version of the ion implantation simulation exploits primarily coarse-grain parallelism and has been implemented by using MPI (message passing interface [15]) which took several months and was very error-prone.

4.2 Experimental results and further work

We ported our code onto a network of workstations (NOWs) as this architecture is well-suited to support coarse-grain parallelism and is also becoming increasingly popular due to the availability of unused computation cycles. We used a

heterogeneous NOW consisting of several DEC alpha workstations including DEC 3000 (175 MHz), DEC 7000 (200 MHz), and DEC 600 (333 MHz) workstations. The DEC 600 workstations are connected by a 100 Mbits/sec Ethernet, and all others by a 10 Mbits/sec Ethernet. This NOW is a non-dedicated system where all workstations are office computers. We examined the performance of the parallel ion implantation simulator for two different problem sizes. 2000 (192 ions per time step) and 5000 (768 ions per time step) ions have been, respectively, distributed to the slaves considering the different computing capabilities of the various workstations. The execution times (see Figure 9) have been measured during regular office time which means that there has been computational load that is not related to the given application. The measurements for the problem size of 5000 ions has been done twice in order to demonstrate the impact of different workloads (unrelated to the given application) on the NOW. For the problem sizes 2000 and 5000 ions (1st run) we observe a maximum speedup of 1.6 and 2.3, respectively, for 5 slaves. Whereas, for the second run of the 5000 ions problem size, we achieve a very reasonable speedup of 5.3 for 6 slaves. Figure 10 shows the idle times (waiting for data from other slaves and the master) for each slave of every specific execution (number of slaves is fixed) of the ion implantation simulator. Clearly, we can observe that the idle times across different slaves can be quite different. This is due to uneven workload (unrelated to the application) on the NOW and also due to the degree of ions that move from the geometry domain of one slave to another. Note the strong difference in idle times for two different executions of identical number of slaves and problem sizes (5000 ions). The slaves of these problem sizes have been executed on the same workstations but with different unrelated workload.

The experiments clearly imply that the static load balancing should be replaced by a dynamic load balancing which is sensitive towards dynamically changing application and machine characteristics (for instance, moving ions and application unrelated workload).

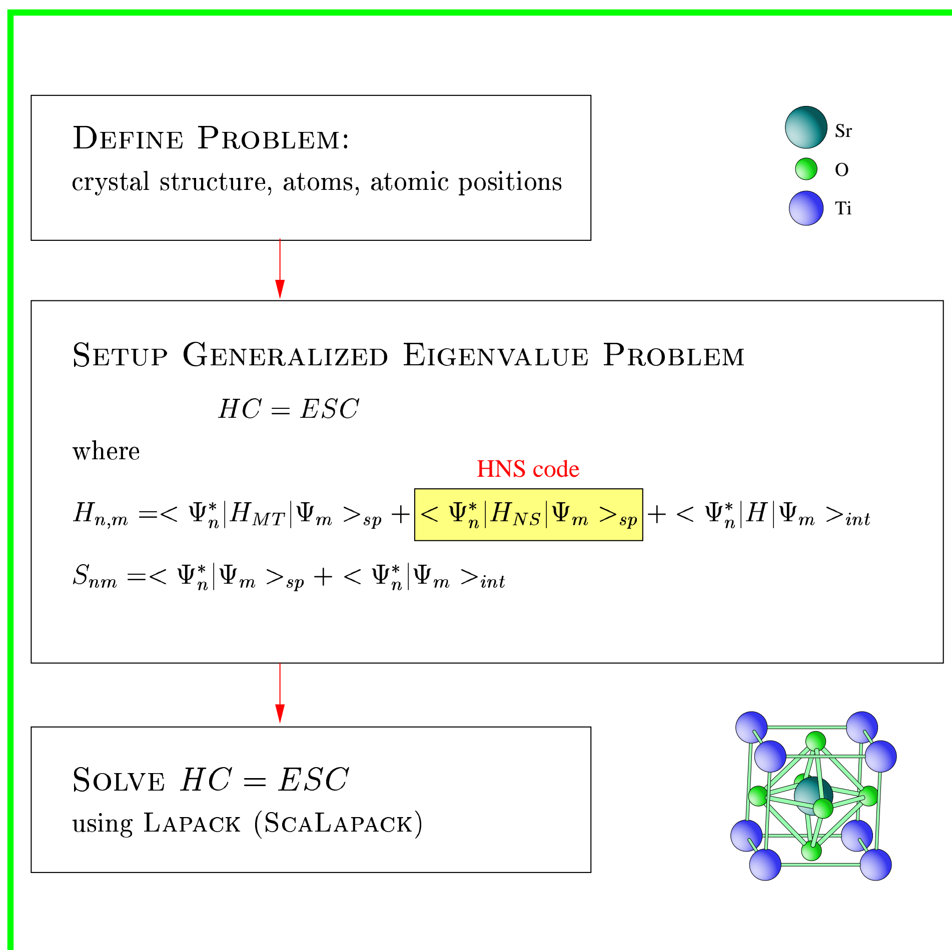

Figure 11. Computation of a crystal structure using WIEN97

5 Quantum Mechanical Calculations of Solids

During the last 16 years a program package called WIEN97 [4] has been developed and is used worldwide by more than 280 research groups. It is based on density functional theory, for which Walter Kohn received the Nobel prize for chemistry in 1998, and the LAPW method [35] which is one of the most accurate methods to investigate theoretically the properties of high technology materials. Applications to the new high temperature superconductors, magnetic structures (for magnetic recording), surfaces (catalysis) or intercalation compounds (new Li batteries) require a reliable computer code that can run even for weeks on a single CPU to produce final results. For this reason parallel computing is highly desirable.

WIEN97 calculates the electronic structure of solids. Figure 11 describes the principle tasks of such a calculation: After the definition of the problem, a *generalized eigenvalue problem* must first be setup and then solved iteratively (i.e. many times) leading to energies (eigenvalues, E) and the corresponding coefficients (eigenvectors, C). The size (N) of the corresponding Hamilton (H) and Overlap (S) matrices is related to the accuracy of the calculation and thus to the number of plane wave (PW) basis functions. About 50 - 100 PWs are needed per atom in the unit cell. For systems containing 50 up to 100 atoms per unit cell matrices of the size 2500 to 10000 must be handled.

The most CPU intensive part of WIEN is the solution of the generalized eigenvalue problem which at present is solved using modified LAPACK (or ScalaPack in parallel) routines. The second most important step is setting up the matrix elements of H and S, which are complicated sums of various terms (integrals between basis functions). A large fraction of this time is spent in the subroutine HNS, where the contributions to H due to the nonspherical potential are calculated.

In HNS radial and angular dependent contributions to these elements are precomputed and condensed in a number of vectors which are then applied in a series of rank-2 updates to the symmetric (hermitian) Hamilton matrix. HNS has 17 one-, 14 two-, 5 three-, and 6 four-dimensional arrays. The computational complexity of HNS is of the order $O(N^2)$. All floating point operations are done in double (eight bytes) precision.

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,CYCLIC) ONTO PR :: H
...
DO 60 I = 1, N
   !HPF$ INDEPENDENT, ON HOME (H(:,J))
   DO 70 J = 1, I
      H(I,J) = H(I,J) + A1R(1,J)*A2R(1,I)
      H(I,J) = H(I,J) - A1I(1,J)*A2I(1,I)
      H(I,J) = H(I,J) + B1R(1,J)*B2R(1,I)
      H(I,J) = H(I,J) - B1I(1,J)*B2I(1,I)
70    CONTINUE
60 CONTINUE
...
DO 260 I = N+1, N+NLO
   !HPF$ INDEPENDENT, ON HOME (H(:,J))
   DO 270 J = 1, I
      H(I,J) = H(I,J) + A1R(1,J)*A2R(1,I)
      H(I,J) = H(I,J) - A1I(1,J)*A2I(1,I)
      H(I,J) = H(I,J) + B1R(1,J)*B2R(1,I)
      H(I,J) = H(I,J) - B1I(1,J)*B2I(1,I)
      H(I,J) = H(I,J) + C1R(1,J)*C2R(1,I)
      H(I,J) = H(I,J) - C1I(1,J)*C2I(1,I)
270   CONTINUE
260 CONTINUE
...
```

Figure 12. HNS based on HPF DO-Independent

```
...
!HPF$ PROCESSORS :: PR(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE(*,CYCLIC) ONTO PR :: H
...
DO 60 I = 1, N
   H(I,1:I) = H(I,1:I) + A1R(1,1:I)*A2R(1,I)
   H(I,1:I) = H(I,1:I) - A1I(1,1:I)*A2I(1,I)
   H(I,1:I) = H(I,1:I) + B1R(1,1:I)*B2R(1,I)
   H(I,1:I) = H(I,1:I) - B1I(1,1:I)*B2I(1,I)
60 CONTINUE
...
DO 260 I = N+1, N+NLO
   H(I,1:I) = H(I,1:I) + A1R(1,1:I)*A2R(1,I)
   H(I,1:I) = H(I,1:I) - A1I(1,1:I)*A2I(1,I)
   H(I,1:I) = H(I,1:I) + B1R(1,1:I)*B2R(1,I)
   H(I,1:I) = H(I,1:I) - B1I(1,1:I)*B2I(1,I)
   H(I,1:I) = H(I,1:I) + C1R(1,1:I)*C2R(1,I)
   H(I,1:I) = H(I,1:I) - C1I(1,1:I)*C2I(1,I)
260 CONTINUE
...
```

Figure 13. HNS based on HPF/Fortran90 array operations

5.1 Parallelization

We used VFC [2] to generate a code in two different parallel HNS versions which are based on HPF. In both versions H, the main HNS array, has been distributed CYCLIC [20] in the second dimension onto the maximum number of processors (HPF intrinsic function NUMBER_OF_PROCESSORS) – that are available on a given architecture. In order to achieve good work distribution, CYLCIC distribution has been chosen according to triangular loop iteration spaces. In the first version (see Figure 12), we use the HPF DO-Independent directive to indicate that the iterations of DO-loops 70 and 270 can be executed

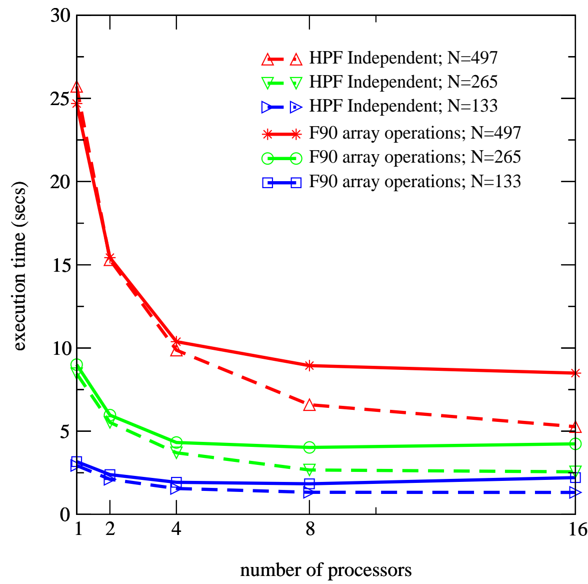

Figure 14. Execution times for two different parallel versions of the WIEN97 HNS code (HPF DO-Independent and HPF/Fortran90 array operations) for varying processors and problem sizes (N) on a NEC Cenju 4.

simultaneously. This version is solely based on Fortran77. In accordance with the owner-computes paradigm [2] an iteration is executed by the processor that owns array section H(:,J). The second code version is based on executing Fortran90 array operations [30] inside of DO-loops 60 and 260. The array operations are executed in parallel based on the owner-computes-paradigm and the HPF distribution directives. Note that both code versions have identical semantics. They differ only in their parallelization strategy.

5.2 Experimental results and further work

Figure 14 shows the execution times of both versions for three different problem sizes N (controls size of array H) on a NEC Cenju-4 with 16 processors. The HPF DO-Independent version clearly outperforms the HPF/Fortran90 version for all problem and machine sizes. VFC generates more efficient code for the HPF DO-Independent version by determining the work distribution outside of the innermost loops. Moreover, the array subscript expressions and loop bounds are only changed very little. Whereas for the second version, VFC uses ADLIB [8] to parallelize the Fortran90 array operations which requires changing array subscript expressions and loop bounds more extensively and the overhead for computing the work distribution is larger than for the HPF DO-Independent version. The measured execution times also demonstrate that the performance scales better for larger than for smaller problem sizes. For instance, the speedup achieved for a 8 processor HPF DO-Independent version is 4.0 for N=497 and 2.2 for N=133. Similarly, the speedup achieved for a 8 processor HPF/Fortran90 is 2.5 for N=497 and 1.7 for N=133.

It should also be stated that detecting parallelism and inserting HPF directives took less than 1/2 day for both parallel versions of the HNS code.

In the current work the main loop of HNS was parallelized. The initialization part, which consumes approximately 15 % of the overall execution time, will also be implemented. We will examine various data distributions for the diagonalization routines. Thereafter, we plan to parallelize the setup phase of the spherical part H_{MT}. Overall we are very confident that HPF has to potential to parallelize large and substantial portions of the WIEN97 application.

6 Related Work

TAU [31] is a sophisticated instrumentation, tracing and profiling system that has been shown to be very useful for various programming paradigms including PC++ and HPC++ [26].

Forge90 [29] reports on communication costs at the level of a generated message passing code, but not at the level of the input program.

An approach for visualizing the performance for HPF programs is described in [27]. Various insights about the interplay between data mapping and communication for HPF programs are offered by this system.

In [1] the performance of Fortran D programs is analyzed at the source-level which is based on an integration with the Fortran D compiler [16] and the Pablo performance system [34]. MPP Apprentice [40] supports post-execution performance analysis for C, C++, and Fortran90 programs on the Cray T3D machine. The previous two approaches are most similar to our approach. The Fortran D/Pablo integrated performance system has sophisticated capabilities to link performance data with distribution, alignment and mapping information for data parallel programs. It is unclear how accurate this system can record code transformations and optimizations which is a strength of Scala. Moreover, Scala collects more comprehensive information about arrays and can also describe the memory requirements for a given program. Apprentice maintains information about code restructuring for basic blocks. It reports time statistics for loops and for an entire application. Scala goes beyond basic blocks and can also record code changes that imply larger code sections than basic blocks (e.g. nested loops or procedures). Scala can also deal with new code inserted by a compiler whose performance can be linked to a specific source of the input program.

7 Conclusions

There are many different ways to develop programs for distributed and parallel systems. Frequently users write programs at a very low-level (i.e., message passing programs) in order to fully exploit the computational capabilities of a target architecture which can be very error-prone and time consuming. In recent years compilers provide extensive support to develop distributed and parallel programs at a very high-level which reduces the time effort of code development substantially but sometimes at the cost of a reduced performance. Furthermore, compilers aggressively apply code transformations in order to convert a high-level program to a program with communication and synchronization and in order to improve the resulting performance. This poses a substantial problem for performance measurement and analysis tools. Performance data is frequently monitored at the level of a generated program or target machine without the possibility to map performance data back to the user provided program. In order for performance measurement and analysis tools to be effective and useful, they must be applicable to both high- and low-level programming paradigms.

In this paper we describe the performance-oriented development of three real-world applications for distributed and parallel architectures. Two applications have been developed based on high-level programming paradigms and executed on a dedicated parallel machine (NEC Cenju-4). They benefit by fast program development and also achieve reasonable performance speedup. A third application is based on a master-slave programming model that has been manually developed and ported onto a cluster of heterogeneous workstation and networks. Good speedup figures have been observed for low system loads that are not related to the measured application. This application suffered by a static load balancing which is very insensitive towards dynamically changing application and machine characteristics.

Scala, a portable instrumentation, measurement and post-execution performance analysis tool for distributed and parallel systems, has been used to support the performance-oriented program development of all three applications. The following features of Scala have been particularly useful for these applications.

- Portable instrumentation system supports selective and comprehensive instrumentation of pre-defined types of code regions and arbitrary code regions.
- Code restructuring information of transformation systems records and collects in a measurement description file which enables to relate performance data back to the input program.
- Performance data of several executions can be compared against each other.
- Many important performance metrics can be computed (i.e., speedup, efficiency, communication and work distribution, compiler organization overhead, idle time, etc.).
- Several interfaces are supported in order to employ both self-build and external performance visualization systems.

Scala is currently being used as a performance analysis system for explicit message passing programs (C and Fortran) and for programs generated by the VFC compiler [2] (translates HPF programs into message passing Fortran90 programs based on MPI). We are currently also investigating the usefulness of Scala for performance analysis of JAVA/RMI programs. Moreover, we are in the process to integrate Scala with performance prediction [12] and symbolic analysis techniques [13] to examine the scaling behavior [37] of distributed and parallel programs.

References

[1] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed. Integrating Compilation and Performance Analysis for Data Parallel Programs. . In M. Simmons, A. Hayes, D. Reed, and E. J. Brown, editors, *Proc. of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press*, January 1996.

[2] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[3] P. Blaha, K. Schwarz, P. Dufek, and R. Augustyn. Wien95, a full-potential, linearized augmented plane wave program for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, 1995.

[4] P. Blaha, K. Schwarz, and J. Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, ISBN 3-9501031-0-4, 1999.

[5] W. Bohmayr, A. Burenkov, J. Lorenz, H. Ryssel, and S. Selberherr. Trajectory split method for Monte Carlo simulation of ion implantation. *IEEE Transactions on Semiconductor Manufacturing*, 8(4):402–407, 1995.

[6] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Medea: A tool for workload characterization of parallel systems. *IEEE parallel and distributed technology: systems and applications*, 3(4):72–80, Winter 1995.

[7] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. Integration of a compilation system and a performance tool: the hpf+ approach. In *Proc. of the International Conference on High-Performance Computing and Networking (HPCN'98), Amsterdam, The Netherlands*, pages 809–815. Lecture Notes in Computer Science, Springer Verlag, 1998.

[8] B. Carpenter. Adlib: A Distributed Array Library to Support HPF Translation. In *Proc. of the 5th Workshop on Compilers for Parallel Computers, Malaga, Spain*, June 1995.

[9] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, Jan./Mar. 1998.

[10] E. Dockner and H. Moritsch. Pricing Constant Maturity Floaters with Embeeded Options Using Monte Carlo Simulation. Technical Report AuR_99-04, Aurora Technical Reports, University of Vienna, January 1999.

[11] H. M. E. Dockner. Pricing constant maturity floaters with embedded options using monte carlo simulation. Technical Report TR1999-04, Special Research Program SFB F011 AURORA, 1999.

[12] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[13] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.

[14] T. Fahringer, B. Scholz, and M. Pantano. Execution-Driven Performance Analysis for Distributed and Parallel Systems. Technical Report, Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Wien, June 1999.

[15] M. P. I. Forum. *Document for a Standard Message Passing Interface*, draft edition, Nov. 1993.

[16] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[17] Grace User's Guide V0.2. http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html, March 1999.

[18] E. D. H. Moritsch. Numerical procedures for pricing interest rate dependent securities and their parallel implementations. Technical Report TR1999-?, Special Research Program SFB F011 AURORA, 1999.

[19] V. Herrarte and E. Lusk. Study parallel program behavior with Upshot. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, Aug. 1991.

[20] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

[21] A. Hössinger, M. Radi, B. Scholz, T. Fahringer, E.Langer, and S. Selberherr. Parallelization of a Monte-Carlo Ion Implantation Simulator for Three-Dimensional Crystalline Structures. In *Proceedings of the International Conference on Simulation of Semiconductor Processes and Devices (SISPAD99), Springer*, Kyoto, Japan, Sept. 1999.

[22] A. Hössinger and S. Selberherr. Accurate Three-Dimensional Simulation of Damage Caused by Ion Implantation . In *Proc. 2nd Int. Conf. on Modeling and Simulation of Microsystems*, pages 363–366, April 1999.

[23] J. C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, April 1997.

[24] A. W. J. C. Hull. One factor interest rate models and the valuation of interest rate derivative securities. *Journal of Financial and Quantitative Analysis*, (28):235–254, 1993.

[25] S. Z. J.M. Hutchinson. Financial simulations on a massively parallallel connection machine. *The International Journal of Supercomputer Applications*, 5(2):27–45, 1991.

[26] E. Johnson, D. Gannon, and P. Beckman. HPC++: Experiments with the parallel standard template library. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 124–131, New York, July 7–11 1997. ACM Press.

[27] D. Kimelman, P. Mittal, E. Schonberg, P. F. Sweeney, K.-Y. Wang, and D. Zernik. Visualizing the execution of High Performance Fortran (HPF) programs. In IEEE, editor, *IPPS '95: 9th International parallel processing symposium — April 25–28, 1995, Santa Barbara, CA*, International Parallel Processing Symposium, pages 750–759, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. IEEE Computer Society Press.

[28] C. S. L. Clelow. *Implementing derivative Models*. John Wiley & Sons, 1998.

[29] J. M. Levesque. FORGE90 and High Performance Fortran (HPF). In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 111–119. Springer-Verlag, 1993.

[30] M. Metcalf and J. Reid. *Fortran 90/95 explained*. Oxfor Science Publications, 1996.

[31] B. Mohr, D. Brown, and A. Malony. TAU: A portable parallel program analysis environment for pC++. In *CONPAR*, Linz, Austria, 94.

[32] T. Nakata, Y. Kanoh, K. Tatsukawa, S. Yanagida, N. Nishi, and H. Takayama. Architecture and the Software Environment of Parallel Computer Cenju-4. *NEC Research and Development Journal*, 39:385–390, October 1998.
[33] P. G. P. Boyle, M. Broadie. Monte carlo methods for security pricing. *Journal of Economic Dynamics and Control*, pages 1267–1321, 1997.
[34] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.
[35] K. Schwarz and P. Blaha. Description of an LAPW DF Program (Wien95). *Lec.Notes in Chemistry*, pages 67:139–153, 1996.
[36] X.-H. Sun, M. Pantano, and T. Fahringer. Integrated Range Comparison for Data-Parallel Compilation Systems. Technical Report 97-004, Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803-4020, April 1997.
[37] X.-H. Sun, M. Pantano, and T. Fahringer. Performance Range Comparison for Restructuring Compilation. In *1998 International Conference on Parallel Processing*, Minneapolis, Minnesota, August 1998. IEEE Computer Society Press.
[38] X.-H. Sun, M. Pantano, and T. Fahringer. Integrated Range Comparison for Data-Parallel Compilation Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(5), May 1999.
[39] Sun Microsystems. Java RMI.
[40] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D, 1994.
[41] S. Zenios. *Parallel Monte Carlo simulation of mortgage-backed securities*. Cambridge University Press, 1993.