Solving 3D incompressible Navier-Stokes equations on hybrid CPU/GPU systems

Yushan Wang
Université Paris-Sud
France
yushan.wang@lri.fr

Marc Baboulin
Université Paris-Sud and Inria
France
marc.baboulin@inria.fr

Karl Rupp
Vienna University of Technology
Austria
rupp@iue.tuwien.ac.at

Oliver Le Maître
LIMSI-CNRS
France
olm@limsi.fr

Yann Fraigneau
LIMSI-CNRS
France
yann.fraigneau@limsi.fr

Keywords: Navier-Stokes equations, prediction-projection method, parallel computing, Graphics Processing Unit (GPU)

Abstract

This paper describes a hybrid multicore/GPU solver for the incompressible Navier-Stokes equations with constant coefficients, discretized by the finite difference method. By applying the prediction-projection method, the Navier-Stokes equations are transformed into a combination of Helmholtz-like and Poisson equations for which we describe efficient solvers. As an extension of our previous paper [1], this paper proposes a new implementation that takes advantage of GPU accelerators. We present numerical experiments on a current hybrid machine.

1. INTRODUCTION TO THE PROBLEM OF NAVIER-STOKES

The incompressible Navier-Stokes (NS) equations [2] are the fundamental bases of many computational fluid dynamics problems [3] as they model fluid flows with constant density. These equations are widely used in both academic and industrial contexts to compute fluid flows in various application domains (*e.g.*, low-speed aerodynamics, oceanography, biology, industrial systems...). In their simplest form, the NS equations express the conservation of momentum (Newton's second law) and mass, with a viscous fluid stress proportional to the deformation rate (Newtonian fluid). Closed form solutions are generally non-available, in particular because of the non-linear character of the equations, and various computational methods have been proposed for their numerical approximation. We mentioned in [1] possible numerical methods for solving the NS equations, depending on the type of problem. More specifically for GPU architectures, several NS solvers have been developed in recent years (see *e.g.* [4] that uses finite element method). In this paper, we consider a finite difference discretization of the prediction-projection method [5], and we present a hybrid solver for the resolution of the 3D incompressible Navier-Stokes equations.

$$\left\{ \begin{array}{rcl} \dfrac{\partial \mathbf{V}}{\partial t} + \nabla \cdot (\mathbf{V} \otimes \mathbf{V}^T) & = & -\nabla P + \dfrac{1}{\mathrm{Re}} \Delta \mathbf{V}, \\ \nabla \cdot \mathbf{V} & = & 0. \end{array} \right. \quad (1)$$

In Eq. (1), $\mathbf{V} = (V_1, V_2, V_3)^T(x_1, x_2, x_3, t)$ is the 3D velocity vector and $P = P(x_1, x_2, x_3, t)$ is the fluid pressure. The Reynolds number Re is the ratio of the characteristic inertial and viscous forces. As Re increases, non-linear effects arising from the non-linear convection term $\nabla \cdot (\mathbf{V} \otimes \mathbf{V}^T)$ become more important and the flow exhibits fluctuations at smaller scales requiring higher resolution. We restrict ourself to simple rectangular computational domains supporting uniform discretization grids. The prediction-projection method is a time-integration approach which transforms the original NS problem into sequences of Helmholtz-like and Poisson problems which are easier to solve numerically. Details on the prediction-projection method can be found in [1]. The NS problem has to be complemented with initial and boundary conditions. We do not discuss boundary conditions and only mention without details that our NS solver can deal with Neumann, Dirichlet and periodical boundary conditions. In Section 2, we describe the structure of the Helmholtz-like and Poisson equations resulting from the prediction-projection method. Then, Section 3 presents algorithms and implementations for solving these two equations using MAGMA [6, 7] which is a dense linear algebra library for heterogeneous multicore/GPU architectures with interface similar to LAPACK. Performance and numerical tests are given in section 4.

2. SOLVING THE HELMHOLTZ AND POISSON EQUATIONS

According to the prediction-projection method, the Navier-Stokes Eq. (1) is transformed into a Helmholtz-like equation for the velocity field and a Poisson equation for the pressure. In this section, we present the numerical methods used for solving these equations and the resulting structures.

2.1. Helmholtz-like equation

Upon introduction of a suitable time-integration scheme (*e.g.*, with an explicit treatment of the non-linearities), the prediction step consists of solving uncoupled 3D Helmholtz-like equations for the components of the prediction velocity $\mathbf{V}^*$ given the current solution $\mathbf{V}^n$ at time t_n:

$$(\mathbf{I}-\alpha\Delta)(V_i^*-V_i^n)=S_i^n, \quad i=1,2,3, \tag{2}$$

where $\mathbf{I}$ is the identity matrix, Δ is the 3D Laplacian operator, $\alpha>0$ is a constant that depends on Re and the time step Δt. The known source S_i^n contains the explicit convection terms $\nabla\cdot(\mathbf{V}^n\otimes\mathbf{V}^{n^T})$. The boundary conditions for the increments $V_i^*-V_i^n$ in Eq. (2) can be of Neumann, Dirichlet or periodic type, but are always homogeneous in the Neumann and Dirichlet cases for steady boundary conditions. In the remainder of this paper, Eq. (2) will be simply referred to as Helmholtz equation.

An Alternating Direction Implicit (ADI) method [8] is then used to approximate the 3D Laplacian operator Δ by the product of three 1D operators Δ_1, Δ_2 and Δ_3. Eq. (2) is accordingly transformed into a system of three equations which are solved successively.

$$\begin{cases} (\mathbf{I}-\alpha\Delta_1)r &= S_i^n, \\ (\mathbf{I}-\alpha\Delta_2)r' &= r, \\ (\mathbf{I}-\alpha\Delta_3)(V_i^*-V_i^n) &= r', \end{cases} \tag{3}$$

with $i=1,2,3$ and r, r' intermediate fields.

As illustrated in Fig. 1 in the 2D case, the computational domain is divided equally into subdomains (shown by bold lines) and a uniform grid is applied on each subdomain. For a subdomain, the boundaries include layers of interface cells with all its neighbor subdomains and possibly layers of real boundary cells. Let $\mathcal{D}_i$ denote the number of subdomains in directions $i=1,2,3$ such that $\mathcal{D}=\mathcal{D}_1\mathcal{D}_2\mathcal{D}_3$ is the total number of subdomains. For simplicity, we assume subdomains with number n_i of grid points in each direction (including interface and/or boundary points). It allows to define a unique mesh size h_i along each dimension. In addition, $n=n_1n_2n_3$ is the total number of points in a subdomain while the total number of computational points (including boundary but not interface points) is computed by

$$\mathcal{N}=\prod_{i=1}^{3}\mathcal{N}_i=\prod_{i=1}^{3}(\mathcal{D}_i(n_i-2)+2).$$

For simplicity, we detail the GPU solver for $i=1$ and note $S_1^n=f$ in system (3). The treatment of the other two components is similar. Using the second-order finite difference discretization of the 1D Laplacian operators, the system (3) leads to tridiagonal linear systems. For instance, ordering the unknowns according to $i=1\rightarrow i=2\rightarrow i=3$ order, the discretization of the operator $(\mathbf{I}-\alpha\Delta_x)$ yields the block tridiagonal system:

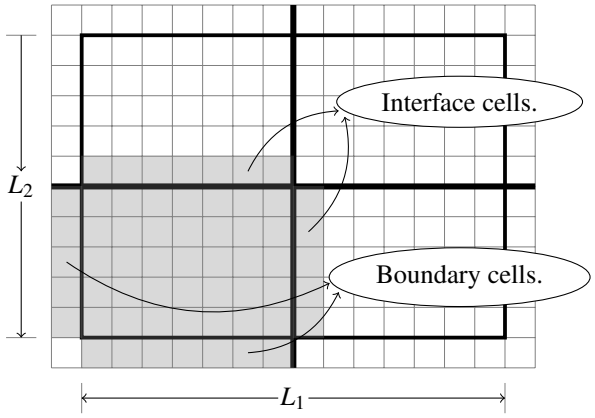

Figure 1. Example of a 2D domain decomposition and interface layer.

$$\begin{pmatrix} B & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{pmatrix}\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{\mathcal{N}_2\times\mathcal{N}_3} \end{pmatrix}=\begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{\mathcal{N}_2\times\mathcal{N}_3} \end{pmatrix}, \tag{4}$$

where $B=\mathbf{I}-\dfrac{\alpha}{h_1^2}\begin{pmatrix} c & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & c \end{pmatrix}\in\mathbb{R}^{\mathcal{N}_1\times\mathcal{N}_1}$,

$c=\begin{cases} -1, & \text{for Neumann BC} \\ -2, & \text{for Dirichlet BC} \end{cases}$, and $r_j,\ f_j\in\mathbb{R}^{\mathcal{N}_1}$, $j=1,2,...,\mathcal{N}_2\times\mathcal{N}_3$.

For a periodic boundary conditions the matrix B has a different structure:

$$B_{\text{periodic}}=\mathbf{I}-\frac{\alpha}{h_1^2}\begin{pmatrix} -2 & 1 & & & 1 \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ 1 & & & 1 & -2 \end{pmatrix}.$$

A naive approach to solve Eq. (4) with B_{periodic} can be expensive. Instead, the Sherman-Morrison algorithm [9] is applied to reduce B_{periodic} to a tridiagonal matrix. When the boundary condition types are constant along each faces of the computational domain (here the two faces with normal in direction $i=1$), the blocks B in system (4) are all the same so its resolution reduces to a smaller system with multiple RHS:

$$B\begin{bmatrix} r_1 & r_2 & \dots & r_{\mathcal{N}_2\times\mathcal{N}_3} \end{bmatrix}=\begin{bmatrix} f_1 & f_2 & \dots & f_{\mathcal{N}_2\times\mathcal{N}_3} \end{bmatrix} \tag{5}$$

Eq. (5) is naturally parallelized according to the number of blocks along $i = 2$ and $i = 3$ directions as shown in Eq. (6), while along the $i = 1$ direction, the Schur complement [10] is applied to ensure the continuity of the solution across subdomain interfaces.

$$B\,[r_1 \quad r_2 \quad \dots \quad r_{n_2\times n_3}]^{(j,k)} = [f_1 \quad f_2 \quad \dots \quad f_{n_2\times n_3}]^{(j,k)}, \qquad j = 1,...,\mathcal{D}_2; \quad k = 1,...,\mathcal{D}_3. \quad (6)$$

2.2. Poisson equation

The prediction velocity $\mathbf{V}^*$ solution of Helmholtz Eq. (2) needs be corrected to enforce the divergence free conditions. This is achieved by computing a correction potential ϕ by means of solving the Poisson equation

$$\Delta\phi = \alpha\nabla\cdot\mathbf{V}^*, \quad (7)$$

with homogeneous Neumann or periodic boundary conditions. Eq. (7) is rewritten as $L\phi = s$ as follows

$$(L_1 + L_2 + L_3)\phi = s, \quad (8)$$

where $L_i = \dfrac{\partial^2}{\partial x_i^2}$, $i = 1,2,3$.

Many methods exist for solving the Poisson equation based on multigrid techniques [11] and Fourier transformation [12]. The method we use in our solver is based on partial diagonalization and uses the eigen decompositions of L_1 and L_2:

$$\begin{aligned} L_1 &= Q_1\Lambda_1 Q_1^{-1}, \\ L_2 &= Q_2\Lambda_2 Q_2^{-1}, \end{aligned}$$

where $\Lambda_{i=1,2}$ are the eigenvalue diagonal matrices and $Q_{i=1,2}$ the eigenvector matrices of the 1D operators.

By defining

$$\begin{aligned} s' &= Q_1^{-1}Q_2^{-1}s, \\ \phi' &= Q_1^{-1}Q_2^{-1}\phi, \end{aligned}$$

we obtain a new tridiagonal system for the $i = 3$ direction:

$$(\Lambda_1 + \Lambda_2 + L_3)\phi' = s'. \quad (9)$$

Obviously, Eq. (9) is also a block tridiagonal system by using the standard second order approximation of the operator L_3 and the $i = 3 \to i = 1 \to i = 2$ ordering. In contrast to Eq. (4), the tridiagonal blocks in Eq. (9) are not identical. The matrix form of Eq. (9) is represented by Eq.(10):

$$\begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_{\mathcal{N}_1\times\mathcal{N}_2} \end{pmatrix} \begin{pmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{\mathcal{N}_1\times\mathcal{N}_2} \end{pmatrix} = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{\mathcal{N}_1\times\mathcal{N}_2} \end{pmatrix}, \quad (10)$$

where the k-th tridiagonal block D_k is:

$$\begin{pmatrix} d_{kk}-1 & 1 & & & \\ 1 & d_{kk}-2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & d_{kk}-2 & 1 \\ & & & 1 & d_{kk}-1 \end{pmatrix} \in \mathbb{R}^{\mathcal{N}_3\times\mathcal{N}_3},$$

with $d_{kk} = \lambda_1(id_1) + \lambda_2(id_2)$, where λ_1 and λ_2 are eigenvalues of L_1 and L_2 respectively, and $id_1 = (k\%(\mathcal{N}_1\times\mathcal{N}_3))/\mathcal{N}_3$ and $id_2 = k/(\mathcal{N}_1\times\mathcal{N}_3)$ are the corresponding $i = 1$ and $i = 2$ index for the k-th block (% denotes the modulo operator).

3. ALGORITHMS AND IMPLEMENTATIONS

This section describes how we can solve Helmholtz and Poisson equations using heterogeneous multicore/GPU architectures. For each solver we first identify the most time-consuming task in the calculation using an existing CPU solver SUNFLUIDH which is based on MPI [13] and is developed at LIMSI[1]. Then, we propose an efficient algorithm to perform this task on a CPU/GPU machine.

3.1. Helmholtz solver

The solution of the Helmholtz equations can be split in two main tasks: construction of tridiagonal system (including computations of convection-diffusion flux and source term) and tridiagonal solve. Fig. 2 shows how the global computational time is distributed among these tasks when considering one iteration of the SUNFLUIDH solver for a Helmholtz problem (mesh size = 240^3) on a multicore system. We observe that solving the tridiagonal systems (including reordering) represents about 2/3 of the execution time. In the following we explain two possible methods for performing this task efficiently using GPU accelerators.

A classical method to solve Eq. (5) is the Thomas algorithm, which corresponds to a Gaussian elimination without pivoting. We consider a general diagonally dominant tridiagonal system (11) $Ax = s$ where $A \in \mathbb{R}^{m\times m}$ and $x, s \in \mathbb{R}^m$. Then the system is solved using Algorithm 1.

$$\begin{pmatrix} b_1 & c_1 & & & \\ \ddots & \ddots & \ddots & & \\ & a_i & b_i & c_i & \\ & & \ddots & \ddots & \ddots \\ & & & a_m & b_m \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} s_1 \\ \vdots \\ s_i \\ \vdots \\ s_m \end{pmatrix}. \quad (11)$$

[1] Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur (http://www.limsi.fr).

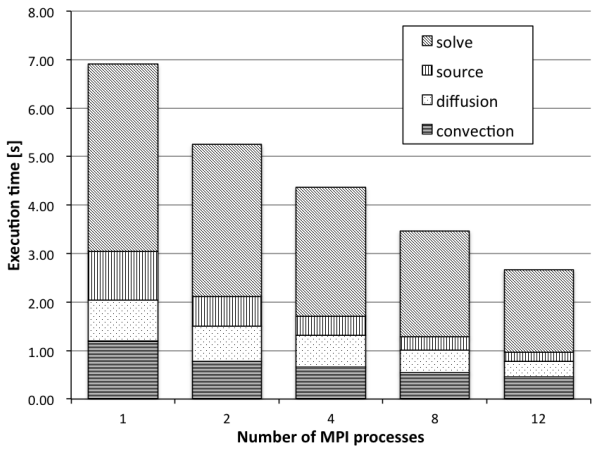

Figure 2. Time breakdown in Helmholtz equation (Intel Xeon E5645 2×6 cores 2.4 GHz.)

Algorithm 1: Thomas algorithm.

Data: Diagonal matrix (a,b,c), RHS s.
Result: Solution x (stored in s_i).

1 **Forward elimination**: **for** $i = 2$ **to** m, **do**
2 $\quad b_i = b_i - \dfrac{c_{i-1} \times a_i}{b_{i-1}}$
3 $\quad s_i = s_i - \dfrac{s_{i-1} \times a_i}{b_{i-1}}$
4 **end**
5 **Backward substitution**: $s_m = \dfrac{s_m}{b_m}$
6 **for** $i = m-1$ **to** 1, **do**
7 $\quad s_i = \dfrac{s_i - c_i \times s_{i+1}}{b_i}$
8 **end**

Algorithm 1 can be easily extended to address systems with multiple RHS by adding loops in lines 3, 5 and 7 of the algorithm. Inside a given RHS row, the operation will be performed by different GPU threads (see Algorithm 2).

To solve one equation of system (3), without considering the domain decomposition, data needed to be copied from CPU to GPU includes the three diagonals and the RHS array. Thus, to solve the whole system, we need to transfer three tridiagonal matrices and the source term. Once these arrays have been transferred (during the initialization step of the solver), we keep them in GPU memory and only use copies for computations.

As system (3) is solved successively, the solution of the first equation is used as the RHS of the second equation. However, the matrices in system (3) are block tridiagonal only if the variables are ordered in a certain way according to the direction. So we need a GPU kernel which deals with the reordering according to the solving direction. Algorithm 3

Algorithm 2: GPU implementation of Thomas algorithm with multiple RHS.

Data: Device copies a_d, b_d, c_d and s_d of the tridiagonal matrix and the RHS array.
Result: Solution x_d (stored in s_d).

1 **Forward elimination**: **for** $i = 2$ **to** m, **do**
2 $\quad b_d[i] -= \dfrac{c_d[i-1] \times a_d[i]}{b_d[i-1]}$
3 $\quad$ **for** *each thread* j **do**
4 $\quad\quad s_d[(j-1)m+i] -= \dfrac{s_d[(j-1)m+i-1] \times a_d[i]}{b_d[i-1]}$
5 $\quad$ **end**
6 **end**
7 **Backward substitution**: **for** *each thread* j **do**
8 $\quad s_d[jm] = \dfrac{b_d[m]}{s_d[jm]}$
9 **end**
10 **for** $i = m-1$ **to** 1, **do**
11 $\quad$ **for** *each thread* j **do**
12 $\quad\quad s_d[(j-1)m+i] = \dfrac{s_d[(j-1)m+i] - c_d[i] \times s_d[(j-1)m+i+1]}{b_d[i]}$
13 $\quad$ **end**
14 **end**

is an example of reordering from $i = 1 \to i = 2 \to i = 3$ to $i = 2 \to i = 3 \to i = 1$.

Algorithm 3: Reorder the solution array after the first solve to fit for the second solve.

int id_1, id_2, id_3;
for *each thread* i **do**
$\quad id_1 = i\%n_1$;
$\quad id_2 = (i\%(n_1 \times n_2))/n_1$;
$\quad id_3 = i/(n_1 \times n_2)$;
$\quad array_{new}[id_2 + id_3 \times n_2 + id_1 \times n_2 \times n_3] = array_{old}[i]$;
end

Another method for solving Eq. (5) is by using the explicit inverse of matrix B. We can find in [14] an expression for the inverse of a general non-singular tridiagonal matrix A used in Eq. (11) where each component of A^{-1} can be expressed as

$$A^{-1}_{ij} = \begin{cases} (-1)^{i+j} c_i c_{i+1} \ldots c_{j-1} \theta_{i-1} \phi_{j+1} / \theta_n, & i < j, \\ \theta_{i-1} \phi_{i+1} / \theta_n, & i = j, \\ (-1)^{i+j} a_{j+1} a_{j+2} \ldots a_i \theta_{j-1} \phi_{i+1} / \theta_n, & i > j, \end{cases} \tag{12}$$

where θ_i's verify the recurrence relation

$$\theta_i = b_i \theta_{i-1} - c_{i-1} a_i \theta_{i-2}, \quad \text{for } i = 2, \ldots, m,$$

with initial conditions $\theta_0 = 1$ and $\theta_1 = b_1$, and ϕ_i's verify the recurrence relation

$$\phi_i = b_i\phi_{i+1} - c_i a_{i+1}\phi_{i+2}, \quad \text{for } i = m-1, ..., 1,$$

with initial conditions $\phi_{m+1} = 1$ and $\phi_m = b_m$, and we also observe that $\theta_m = |A|$.

We use formulas (12) to compute the inverse of B in the initialization step of the solver and store the inverse in memory. The solution of the Eq. (5) is then computed by the matrix-matrix multiplications:

$$\begin{bmatrix} r_1 & r_2 & \dots & r_{\mathcal{N}_2\times\mathcal{N}_3} \end{bmatrix} = B^{-1} \begin{bmatrix} f_1 & f_2 & \dots & f_{\mathcal{N}_2\times\mathcal{N}_3} \end{bmatrix}.$$

3.2. Poisson solver

Fig. 3 represents the time breakdown for one iteration of a Poisson problem (mesh size = 240^3) using SUNFLUIDH on a multicore system. According to the partial diagonalization method mentioned in Section 2.2., the main tasks are base projections and tridiagonal solve. We observe in Fig. 3 that the most time-consuming part is the base projection which corresponds to matrix-matrix multiply (including reordering). In the remainder of this section we explain how to improve this calculation using GPU accelerators.

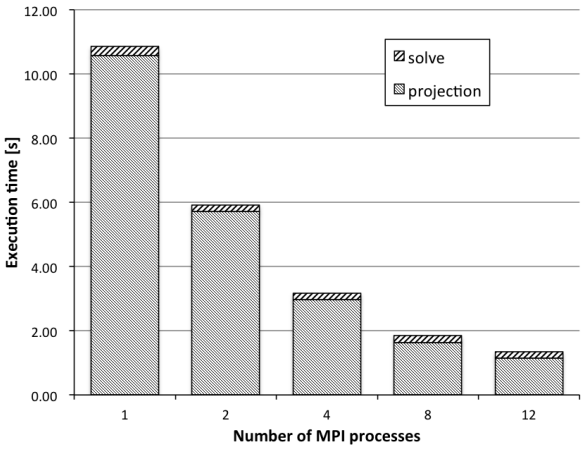

Figure 3. Time breakdown in Poisson equation (Intel Xeon E5645 2×6 cores 2.4 GHz.)

We can also improve the tridiagonal solver even though it does not cost much time. For solving Eq. (8) we can exploit parallelism at the matrix level instead of the RHS level as shown in Algorithm 4. For example in the forward elimination step, we associate each elimination operation with one GPU thread which can be considered as doing multiple eliminations simultaneously.

To reduce the execution time spent in matrix-matrix multiplication, we take advantage of GPU accelerator by calling the MAGMA routine `magmablas_dgemm`. One remark is

Algorithm 4: GPU implementation of Thomas algorithm with multiple matrices.

Data: Device copies a_d, b_d, c_d and s_d of the tridiagonal matrix and the RHS array.
Result: Solution x_d (stored in s_d).

Forward elimination: **for** $i = 2$ **to** m, **do**
  for *each thread* j **do**
    $b_d[(j-1)m+i] -= \dfrac{c_d[(j-1)m+i-1] \times a_d[jm+i-1]}{b_d[jm+i-1]}$
    $s_d[(j-1)\times m+i] -= \dfrac{s_d[(j-1)m+i-1] \times a_d[jm+i-1]}{b_d[jm+i-1]}$
  end
end
Backward substitution: **for** *each thread* j **do**
  $s_d[jm-1] = \dfrac{b_d[jm-1]}{s_d[jm-1]}$
end
for $i = m-1$ **to** 1, **do**
  for *each thread* j **do**
    $s_d[(j-1)m+i] = \dfrac{s_d[(j-1)m+i] - c_d[(j-1)m+i] \times s_d[(j-1)m+i+1]}{b_d[(j-1)m+i]}$
  end
end

that we need to reorder the variables according to the direction considered. For example, to compute the new source term s', we have to reorder the source array s by the order $i = 2 \to i = 3 \to i = 1$ in order to use the `magmablas_dgemm` routine to compute $Q_2^{-1}s$. Next, we have to once again reorder the result $Q_2^{-1}s$ in the order of $i = 1 \to i = 2 \to i = 3$ to proceed to second multiplication with Q_1^{-1}. For the same reason, the product $Q_1^{-1}Q_2^{-1}s$ must be ordered by $i = 3 \to i = 1 \to i = 2$ to fit in the block tridiagonal structure and the solution needs to be once again reordered according to the multiplication factor. The algorithm for the reordering is exactly the same as Algorithm 3 shown in Section 3.1..

Let us now study how to implement on GPU the matrix-matrix multiply for the matrices Q_i described in Section 2.2. with decomposed domain. Suppose that the 3D domain is divided into p subdomains along $i = 1$ direction as shown in Fig. 4. According to the principles of domain decomposition, each subdomain is associated to a multicore processor P_i and to a GPU G_i.

Let us consider for instance the multiplication of Q_1^{-1} and s. As the source array s is already distributed on the corresponding processor or accelerator, and its size is often impor-

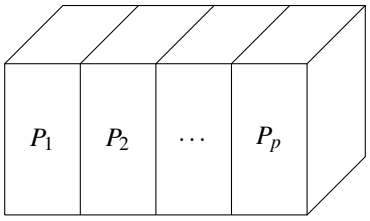

Figure 4. 3D domain decomposition along $i = 1$ direction.

tant, we do not want to re-distribute s by column blocks to perform the usual parallel matrix-matrix multiplication. The rearrangement of s can be very expensive especially when s is stored on the GPU memory. On the other hand, we distribute the matrix Q_1^{-1} by column blocks to the corresponding processor or accelerator as shown in Fig. 5.

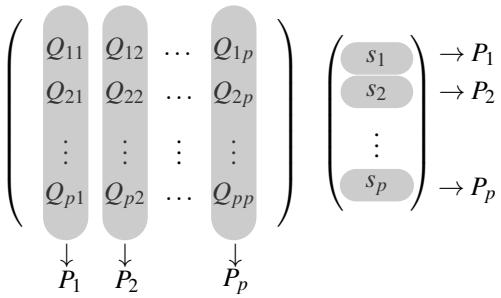

Figure 5. Distribution of $Q_1^{-1} = \{Q_{ij}\}_{i=1,\ldots,p;j=1,\ldots,p}$ and s on multiple processors.

With Q_1^{-1} distributed in blocks, on accelerator G_i, we multiply Q_{ji} and s_i with $j = 1,2,...,p$, where $Q_{ji} \in \mathbb{R}^{n_1 \times n_1}$ and $s_i \in \mathbb{R}^{n_1 \times (n_2 \times n_3)}$. Once all the p multiplications are computed, we send the results back to processor P_i and we call MPI routines `MPI_ALLTOALL` and `MPI_REDUCE` to distribute the block multiplication results and to obtain the final multiplication result as shown in Fig. 6.

To sum up, we compute Q_1, Q_2, Q_1^{-1} and Q_2^{-1} in the initialization phase and copy the corresponding column blocks to the assocoated GPU. While doing the matrix-matrix multiplication, we first transfer (or not if it is already on the GPU) the source array s to the GPU and then call the MAGMA routine `magmablas_dgemm` to obtain the blocks $Q_{ij}s_j$ which are then sent back to processor. Then we proceed the information exchange via MPI routines and send back to GPU the final solution in order to perform the next multiplication or the tridiagonal solve.

4. NUMERICAL EXPERIMENTS

Our experiments have been carried out on a multicore processor Intel Xeon E5645 (2 sockets $\times$ 6 cores) running at 2.4 GHz. The cache size per core is 12 MB and the size of the main memory is 48 GB. This system hosts two GPU NVIDIA

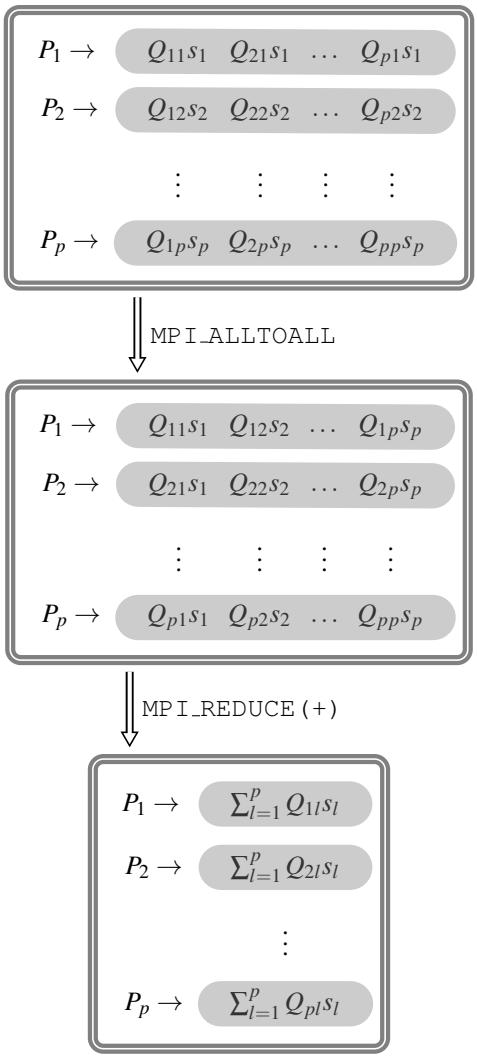

Figure 6. Matrix-matrix multiplication with multiple subdomains.

Tesla C2075 running at 1.15 GHz with 6 GB memory each. MAGMA was linked with the libraries CUDA 5.0 [15] and MKL 10.3.8 [16] respectively, for multicore and GPU. In particular we compare the performance of our new hybrid NS solver with the existing NS solver SUNFLUIDH.

4.1. Helmholtz and Poisson solvers

The 3D Helmholtz test problem that we consider is defined as follows.

$$\begin{cases} \mathbf{V}(\mathbf{x}) - \alpha\Delta\mathbf{V}(\mathbf{x}) &= \mathbf{S}(\mathbf{x}), & \mathbf{x} \in \Omega = (0,1)^3, \\ \mathbf{V}(\mathbf{x}) &= 0, & \mathbf{x} \in \partial\Omega, \end{cases} \tag{13}$$

where $\mathbf{S} = (1 + 3\alpha\pi^2)\mathbf{V}$, $\mathbf{x} = (x_1, x_2, x_3)$ and $\alpha = 10^{-7}$ is de-

fined in Eq. (3). The exact solution is:

$$\mathbf{V}(\mathbf{x}) = \begin{pmatrix} sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \\ sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \\ sin(\pi x_1)sin(\pi x_2)sin(\pi x_3) \end{pmatrix}.$$

In Fig. 7 we compare the two methods described in Section 3.1. for solving the tridiagonal systems resulting from Eq. (13) for different mesh sizes (Thomas algorithm and explicit inverse). For both methods we computed the absolute error given by $||u-u_h||/\sqrt{\mathcal{N}}$ where u and u_h are respectively the exact and approximate solutions. We noticed that the two errors are the same. Regarding the performance, Fig. 7 represents the execution time for the two methods. For instance, for a mesh size of 256^3, using an explicit inverse enables us to gain a factor 4 over the Thomas algorithm. However, as computing the explicit inverse suits only for problems with same boundary conditions for each direction, the standard Thomas algorithm will be still useful in more general Helmholtz problems.

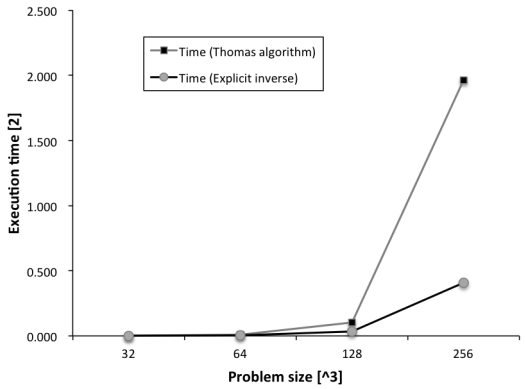

Figure 7. Performance of Helmholtz solver using Thomas algorithm and explicit inverse.

Let us now consider the following Poisson problem.

$$\begin{cases} \Delta\phi(\mathbf{x}) &= s(\mathbf{x}), & \mathbf{x} \in \Omega = (0,1)^3, \\ \dfrac{\partial\phi}{\partial\mathbf{n}}(\mathbf{x}) &= 0, & \mathbf{x} \in \partial\Omega, \end{cases} \tag{14}$$

where $s = -\pi^2\phi$ and the exact solution is

$$\phi(\mathbf{x}) = cos(\pi x_1)cos(\pi x_2)cos(\pi x_3).$$

The performance of the Poisson solver is presented in Fig. 8. We observe that, when the problem size grows, the forward error decreases and the execution time increases.

Table 1 lists the time breakdown for one iteration of the Helmholtz and Poisson solvers and compares the performance of the GPU solvers with the corresponding CPU

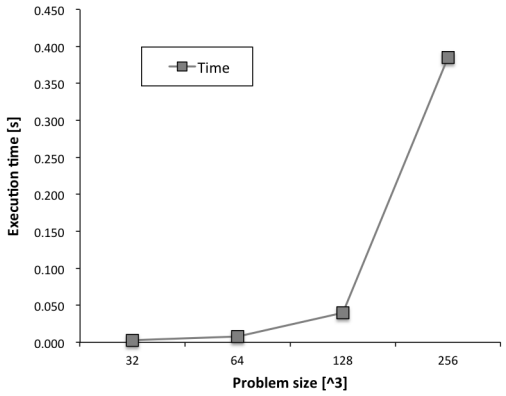

Figure 8. Performance of Poisson solver.

solvers. We observe that the GPU implementations enable us to accelerate the calculations with roughly a factor 8 and 5 for the Helmholtz and Poisson solvers respectively when compared with the CPU solvers using MPI or multithreading. The data transfer from CPU to GPU for the Helmholtz solver includes three RHS vectors (size 240^3) and three matrices B^{-1} (240^2). For the Poisson solver, the amount is three diagonals of size 240^3, one RHS vector of size 240^3, and the matrices $Q_1, Q_1^{-1}, Q_2, Q_2^{-1}$ of size 240^2. Consequently, the data movements for the Poisson solver require 1/3 more time than for the Helmholtz solver, which is confirmed in Table 1.

	Helmholtz (with B^{-1})	Poisson
Transfers CPU-GPU (only once)	85	109
Matrix multiplication	216	96
Solution reordering	126	84
Tridiagonal system solve	-	165
Total CPU solver (12 MPI procs)	2700	1460
Total CPU solver (12 threads)	2750	1760
Total GPU solver	342	345

Table 1. Time (ms) distribution for Poisson and Helmholtz GPU solvers (mesh size = 240^3).

4.2. Hybrid NS solver

We study the global performance of the hybrid multicore+GPU NS solver using OpenMP [17]. Fig. 9 depicts the execution time for one time iteration (the dashed line represents the sequential time using SUNFLUIDH). The problem size is 128^3 for one domain/MPI process and $128 \times 128 \times 64$ per subdomain for two subdomains/MPI processes. Each MPI process corresponds to an hexacore processor and is associated to one GPU. We observe that the computational time is reduced by at least 20% when using two GPUs instead of one. A potential of improvement relies on the use of optimized communication (*e.g.* asynchronous calls to GPU).

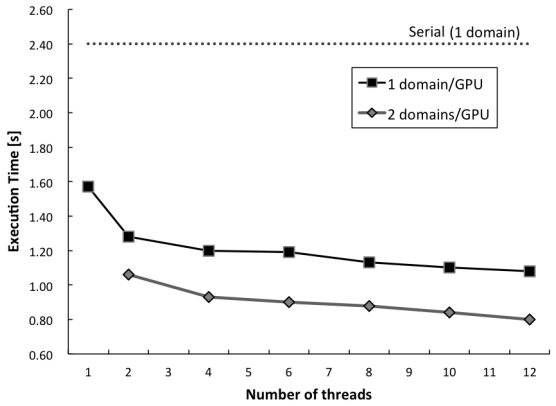

Figure 9. Performance of multicore+GPU NS solver.

5. CONCLUSION

We described a hybrid multicore+GPU Navier-Stokes solver that includes the solution of the Helmholtz and Poisson equations. The performance results showed significant accelerations by using GPU devices. Even though the GPU is used only in several parts of the algorithm, the speed-ups are encouraging, which allows us to continue in this direction. As a future work, we plan to extend the use of GPU in other parts of the NS solver, such as the construction of the tridiagonal Helmholtz system mentioned at the beginning of Section 3.1.. Another improvement will concern the optimization of the memory transfer between CPU and GPU.

ACKNOWLEDGMENT

This work was supported by Région Île-de-France and Digitéo[2], CALIFHA project, contract No 2011-038D. It was also partially funded by Inria-Illinois Joint Laboratory on PetaScale Computing. We thank Franck Cappello and Argonne National Laboratory (Mathematics and Computer Science Division) for hosting the first author in summer 2013. We also thank Joël Falcou (Université Paris-Sud, France) for fruitful discussions on GPU computing.

REFERENCES

[1] Y. Wang, M. Baboulin, J. Dongarra, J. Falcou, Y. Fraigneau, and O. Le Maître. A parallel solver for incompressible fluid flows. *Procedia Computer Science*, (18):439–448, 2013.

[2] P. Constantin and C. Foias. *Navier-Stokes Equations*. University of Chicago Press, 1988.

[3] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 3 edition, 1996.

[4] D. Göddeke, S. H. M. Buijssen, H. Wobker, and S. Turek. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. *International Conference on High Performance Computing & Simulation (HPCS'09)*, pages 12–21, 2009.

[5] A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Mathematics of Computation*, 22(104):745–762, 1968.

[6] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[7] M. Baboulin, J. Dongarra, and S. Tomov. Some issues in dense linear algebra for multicore and special purpose architectures. In *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA'08)*, volume 6126-6127 of *LNCS*. Springer, 2008.

[8] J. Douglas Jr. Alternating direction methods for three space variables. *Numerische Mathematik*, (4):41–63, 1962.

[9] J. Sherman and W. J. Morrison. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics*, 21(1):124–127, 1950.

[10] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2 edition, 2003.

[11] A. McAdams, E. Sifakis, and J. Teran. A parallel multigrid Poisson solver for fluids simulation on large grids. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, 2010.

[12] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *Journal of the ACM*, 12(1):95–113, 1994.

[13] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard*. Int. J. Supercomputer Applications and High Performance Computing, 1994.

[14] R. A. Usmani. Inversion of a tridiagonal Jacobi matrix. *Linear Algebra and its Applications*, 212-213:413–414, 1994.

[15] *NVIDIA CUDA Programming Guide*. 2012. `https://developer.nvidia.com`.

[16] Intel. *Math Kernel Library (MKL)*. `http://www.intel.com/software/products/mkl/`.

[17] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, 2008.

[2] `http://www.digiteo.fr`