Int. J. Appl. Math. Comput. Sci., 2010, Vol. 20, No. 1, 109–121
DOI: 10.2478/v10006-010-0008-4

TWO IMPLEMENTATIONS OF THE PRECONDITIONED CONJUGATE GRADIENT METHOD ON HETEROGENEOUS COMPUTING GRIDS

TIJMEN P. COLLIGNON, MARTIN B. VAN GIJZEN

Delft Institute of Applied Mathematics
Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands
e-mail: {t.p.collignon,m.b.vangijzen}@tudelft.nl

Efficient iterative solution of large linear systems on grid computers is a complex problem. The induced heterogeneity and volatile nature of the aggregated computational resources present numerous algorithmic challenges. This paper describes a case study regarding iterative solution of large sparse linear systems on grid computers within the software constraints of the grid middleware GridSolve and within the algorithmic constraints of preconditioned Conjugate Gradient (CG) type methods. We identify the various bottlenecks induced by the middleware and the iterative algorithm. We consider the standard CG algorithm of Hestenes and Stiefel, and as an alternative the Chronopoulos/Gear variant, a formulation that is potentially better suited for grid computing since it requires only one synchronisation point per iteration, instead of two for standard CG. In addition, we improve the computation-to-communication ratio by maximising the work in the preconditioner. In addition to these algorithmic improvements, we also try to minimise the communication overhead within the communication model currently used by the GridSolve middleware. We present numerical experiments on 3D bubbly flow problems using heterogeneous computing hardware that show lower computing times and better speed-up for the Chronopoulos/Gear variant of conjugate gradients. Finally, we suggest extensions to both the iterative algorithm and the middleware for improving granularity.

Keywords: grid computing, large sparse linear systems, iterative methods, conjugate gradient methods, Chronopoulos/Gear CG, GridSolve middleware, bubbly flows.

1. Introduction

The solution of sparse linear systems is a computational bottleneck for many large scale numerical simulations. In order to solve these systems, which may consist of millions of equations, combined computing power of many processors is needed. Dedicated parallel hardware, however, is expensive.

A natural idea to provide cheap parallel computing power is to use readily available non-dedicated hardware, and thus to make better use of existing resources. This idea has given rise to the concepts of grid computing and computational grids, see, e.g., (Foster and Kesselman, 2004). In grid computing, a pool of computational tasks is dynamically distributed over a computational grid, which can be a local cluster of computers, but it can also be a group of computers at geographically different locations. This approach has proven to be successful for embarrassingly parallel applications where the tasks do not require interprocessor communication, as exemplified by the SETI@home project (Anderson *et al.*, 2002).

For numerical solution of linear systems of equations, however, inter-task communication is unavoidable. For this application, developing efficient parallel numerical algorithms for dedicated homogeneous systems is a difficult problem, but becomes even more challenging when applied to heterogeneous systems. In particular, the heterogeneity of the computational nodes and the variability in network performance offer new algorithmic problems.

In this paper we study different implementations of the Conjugate Gradient (CG) method (Hestenes and Stiefel, 1952) on a heterogeneous computational grid. We use the GridSolve library (Dongarra *et al.*, 2007), which is a mature grid middleware for accessing remote computational resources. Load balancing is achieved using a simple resource-aware data partitioning strategy. The number of synchronisation points in the CG algorithm, which is in its standard implementation equal to two, can be reduced to one by using the implementation proposed by Chronopoulos and Gear (1989).

We apply our approach to the bubbly flow problem, which is an important example of a moving boundary problem. Our numerical experiments show that by minimising the number of synchronisation points and by devoting more work to the preconditioning phase, speed-up can be achieved for the solution of systems of equations, despite the fact that for this application the tasks are tightly coupled.

The remainder of the paper is organised as follows: In the next section we describe in detail our architecture-aware conjugate gradient algorithm. This includes a description of the test problem, a description of GridSolve and its data management strategies, and several details concerning our implementation of a sparse iterative solver on grid computers. Section 3 contains experimental results and in Section 4 we give concluding remarks and some suggestions for future work.

2. Heterogeneous sparse linear solvers in GridSolve

2.1. Motivation. This work is part of a larger project where we want to apply the immersed boundary method (Peskin, 2002; Mittal and Iaccarino, 2005) to simulate general moving boundary problems using grid computers. Examples of such problems are the swimming of fish, airflow around wind turbine rotor blades, and bubbly flows. These simulations involve numerical solution of the governing fluid equations on a structured grid, where the most expensive part usually consists of solving a large sparse linear system $Ax = b$ at each time step. When using a pressure-correction method (van Kan, 1986) to solve the governing equations for bubbly flows on a highly refined mesh, such a large sparse linear system arises from a finite difference discretisation of the following Poisson equation with discontinuous coefficients and Neumann boundary conditions:

$$\begin{cases} -\nabla \cdot \left(\dfrac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}) \right) = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ \dfrac{\partial}{\partial \mathbf{x}} p(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega, \end{cases} \tag{1}$$

for some functions f and g. Here, Ω and $\partial\Omega$ denote the computational domain and boundary, respectively, while p and ρ represent the pressure and density. In this paper we will consider the 3D test problem taken from (van der Pijl *et al.*, 2005; Tang and Vuik, 2007a). It is a two-phase bubbly flow problem where we have two separate fluids Γ_0 and Γ_1, representing water (high-density phase) and water vapour (low-density phase), respectively. The corresponding density function has a large jump, defined by

$$\rho(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in \Gamma_0, \\ \tau, & \mathbf{x} \in \Gamma_1, \end{cases} \tag{2}$$

where we typically have $\tau = 10^{-3}$. Such a discontinuity in the coefficient results in a highly ill-conditioned system, making it a difficult problem for iterative methods. In this paper, we restrict ourselves to a cubical unit domain with a single bubble with the radius 0.25 located in the centre of the computational domain. For more details on applying the pressure-correction method to bubbly flows, the reader is referred to (van der Pijl *et al.*, 2005).

Applying standard finite differences to (1) on a structured $n_x \times n_y \times n_z$ mesh results in the linear system

$$Ax = b, \tag{3}$$

where A is an $n \times n$ block pentadiagonal Symmetric Positive Semi-Definite (SPSD) sparse matrix and $n = n_x n_y n_z$. This implies that the solution x is determined up to a constant. However, it can be shown that for our particular case this does not pose any problems for the iterative solver (Tang and Vuik, 2007b).

The reason why we chose to solve this system using a Preconditioned Conjugate Gradient (PCG) method is twofold: (i) it is the obvious choice for large and sparse SP(S)D systems, and (ii) the CG method consists of three basic computational kernels (i.e., matrix-vector multiplication, inner product, vector update), which are simple to implement and relatively straightforward to parallelise on (dedicated) parallel computers. Also, we combine CG with a (block) Jacobi preconditioner due to its attractive parallelisation properties.

2.2. Brief overview of GridSolve. GridSolve (GS) is a distributed programming system which uses a client-server model for solving complex problems remotely on global networks (Dongarra *et al.*, 2007; YarKhan *et al.*, 2006). It is an instantiation of the GridRPC model, an emerging standard for a Remote Procedure Call (RPC) mechanism on grid computers (Seymour *et al.*, 2002). The GridRPC Application Programming Interface (API) is defined within the Global Grid Forum (Lee *et al.*, 2007). Other projects that use the GridRPC API are DIET (Caron and Desprez, 2006), NetSolve (Seymour *et al.*, 2005), Ninf-G (Tanaka *et al.*, 2003), and OmniRPC (Sato *et al.*, 2003).

Software environments such as GridSolve are often called Network Enabled Servers (NES). These systems typically consist of six components: clients, agents, servers, databases, monitors, and schedulers. We will elaborate on the specific details of these components in the context of the version 0.17.0 of GS (see Fig. 1). The GS servers (component 3) are software components that are started on each computational node, which may consist of a single CPU or a cluster. The server monitors the workload of the node and keeps an updated list of the services (or *tasks*) that are installed on the server. For example, a task can be a single `dgemm` or a parallel MPI job. Services can be added or modified without restarting the server.

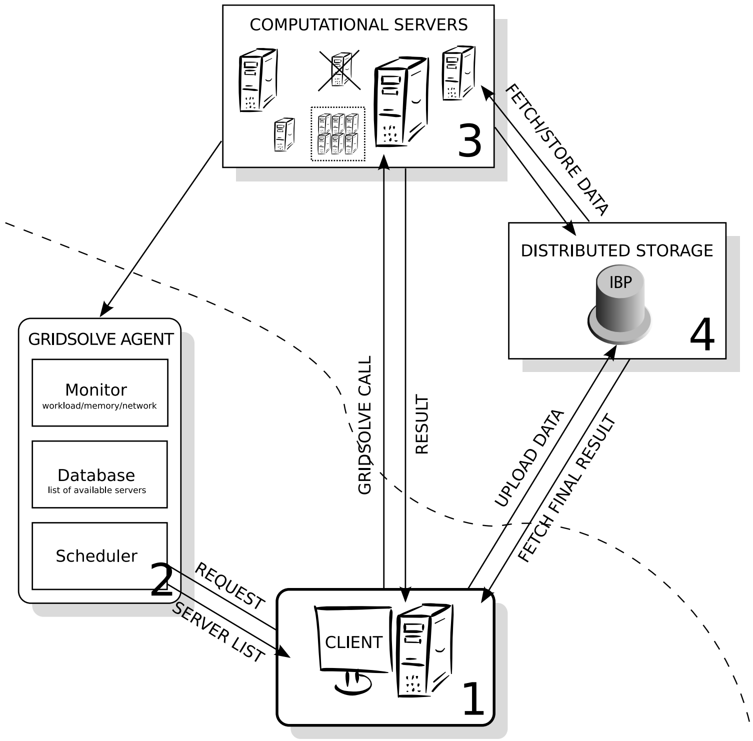


Fig. 1. Schematic overview of GridSolve. The dashed line represents a (geographical) distance between the client and servers.

A single GridSolve agent (Component 2) actively monitors the server properties such as CPU speed, memory size, computational services, and availability. These properties are stored in a database on the agent node and are periodically updated. When a GridSolve client program (Component 1) written in either C, Fortran, or Matlab uses the GridRPC API to initiate a GS call to a remote problem, the GS middleware first contacts the agent.

Based on the problem complexity, the size of the input parameters, and the available computational resources, the agent then returns a list of servers sorted by minimum completion time. The client resorts to the list after performing a quick network performance test. Input parameters are sent to the first server on the list and the task, which can be either blocking or non-blocking, is executed on the server. The result (if any) is then sent back to the client. If a task should fail, it is transparently resubmitted to the next server on the list.

To determine the completion time of a particular task on server s, the total flop count of the problem is divided by the *effective speed* of the server. The latter is calculated using

$$\frac{s_{\text{flops}} \times s_{\text{ncpu}}}{\frac{s_{\text{workload}}}{100} + 1.0}, \tag{4}$$

where s_{flops} is the speed of server s in flops determined by the multiplication of two dense matrices of fixed size, s_{ncpu} is the number of CPUs in the node, and $s_{\text{workload}} \in [0, 100]$ denotes the periodically updated workload. This means that, if a server is fully occupied, it can be used effectively half of the time, which is a realistic assumption.

The main advantages of GridSolve are that it is easy to use, install, and maintain. It allows convenient access to advanced remote computational resources. Furthermore, fault tolerance is supported through a simple but effective mechanism. Nevertheless, the current implementation has several obvious limitations. For example, remote servers cannot communicate directly, which imposes a severe constraint on the type of applications that can be efficiently solved using the current implementation of GridSolve. It is therefore naturally suited for coarse-grained applications such as parametric studies and 'embarrassingly parallel' problems. In contrast, traditional parallel iterative solvers are inherently fine-grained, and much research needs to be performed before iterative solvers can be efficiently applied in grid computing.

In the current GridSolve model, separate tasks communicate data through the client, resulting in bridge communication. As a result, input and output data associated with a task are continuously being sent back and forth between the client and the server using a possibly slow network connection. Also, any data that read or generated locally during the execution of a task are lost after it finishes. Several strategies such as data persistence and data redistribution have been proposed to tackle these deficiencies for different implementations of the GridRPC API (Caron *et al.*, 2005; Brady *et al.*, 2006; 2008; Lastovetsky *et al.*, 2006; Zuo and Lastovetsky, 2007; Desprez and Jeannot, 2004).

In GridSolve there is a partial solution to the first problem called the Distributed Storage Infrastructure (DSI). At the Logistical Computing and Internetworking (LoCI) Laboratory of the University of Tennessee, the IBP (Internet Backplane Protocol) middleware has been developed based on this approach (Beck *et al.*, 2002). To avoid multiple transmissions of the same data between the client and the server, the client can upload data to an IBP data depot which is in close proximity to the computational servers. Subsequently, a data handle is sent to the server and the task can fetch and update the data on the IBP depot (component (4) in Fig. 1). Using the DSI can be considered programming for a shared memory model.

An approach similar to that of (Brady *et al.*, 2006) in which the RPC model of NetSolve is extended to include communication between remote servers has been developed for GridSolve (Brady *et al.*, 2008). In the future, we hope to use this extension and apply it to our problem.

2.3. Resource-aware load balancing. We are interested in solving large sparse linear systems $Ax = b$ using GridSolve with architecture-aware dynamic load balancing. For this purpose, it is insufficient to let the agent return a sorted list based on problem complexity and available resources, as is normally done in GridSolve. Instead, we need to use a slightly different approach. Suppose that the client wishes to use s servers to solve a linear system. The scheduler in the GridSolve agent has been enhanced so that it creates simple (non-homogeneous) partitioning of the computational work over s servers using information about currently available resources. It then returns the partitioning and a list of the said servers to the client, after which the client initiates a series of non-blocking calls *explicitly specifying* the size and location of each task. Thus we ensure that the computational task is being performed on the intended server, in accordance with our partitioning. Unfortunately, the fault-tolerance mechanism within the original GridSolve is now being circumvented, because the tasks cannot be resubmitted to another server should a task fail.

Algorithm 1 shows the general resource-aware CG algorithm. Note that the tasks use DSI file handles to manipulate the vectors on the IBP depot. The specific structure of the CG tasks will be discussed in the next section. After each iteration step, the client may decide to repartition the work and assign the work to different computational servers in the network accordingly.

2.4. Partitioning algorithm and CG schemes. The matrix originating from 2D discretisation of a Poisson problem on a structured grid is a block tridiagonal matrix. This matrix has a structure similar to 3D discretisation of our test problem. In both cases, the block-rows roughly contain the same number of non-zeros, so we chose simple non-homogeneous block-row partitioning. For illustration purposes, Fig. 2 shows heterogeneous partitioning using four servers on a 8×8 grid for a 2D Poisson problem. The partitioning of the block pentadiagonal matrix from the corresponding 3D problem is performed similarly.

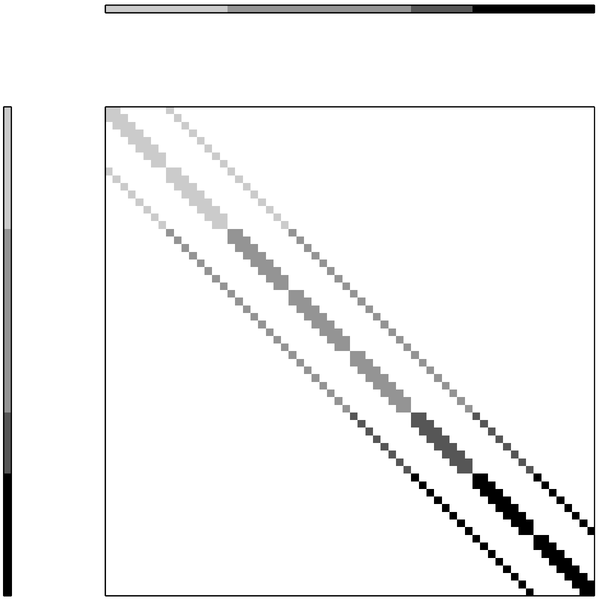


Fig. 2. Heterogeneous block-row partitioning for $s = 4$ and $k = 8$ for a 2D Poisson problem.

The input and output vectors, shown at the top and left side, respectively, are distributed in the same manner. More specifically, the effective speed of s servers is calculated using (4), together with the total flop count of a single CG iteration step. The size of each task is then determined accordingly.

The standard preconditioned conjugate gradient method was implemented first and is shown in Algorithm 2. There are two natural synchronisation barriers, namely, the two inner products for computing α and ρ. Note that synchronisation consists of three separate steps. First, at the end of a subtask the relevant data are updated on the depot (e.g., line 5). The subtask then returns the partial inner product to the client (line 6). Finally, at the start of the next subtask, it reads the relevant data from the depot (line 7). As a result, the depot contains a full copy of the vectors $\mathbf{x}, \mathbf{r}, \mathbf{z}, \mathbf{p}$, and $\mathbf{q}$ at all times. This ensures that, in the case of a server failure during an iteration step, all essential data are preserved on the IBP depot and the iteration process may continue (possibly at a later time) without any problem.

The scheme as it is depicted in Algorithm 2 suggests

Algorithm 1 Resource-aware Preconditioned Conjugate Gradient Algorithm; s servers

```
Agent partitions work based on available computational resources;
Client sets initial values and uploads initial vectors such as $r_0$ to IBP data depot; Set $k = 0$;
while CG not converged and $k < k_{\max}$ do
    Client assigns CG tasks to $s$ servers and waits until tasks have completed;
    Client repartitions work if significant change in workload and/or computational resources has occurred;
    Set $k = k + 1$;
end while
Client reads final answer from IBP depot;
```

Algorithm 2 Preconditioned CG; Task i with three subtasks.

```
Require: File handles to vectors $\mathbf{x}, \mathbf{r}, \mathbf{z}, \mathbf{p}, \mathbf{q}$ on IBP data depot and parameter $k$.
Ensure: Preconditioner $K$.
 1: // Each server i performs the following:
 2: Read $\mathbf{r}^i$ from depot
 3: Solve $\mathbf{z}^i$ from $K\mathbf{z}^i = \mathbf{r}^i$
 4: Compute $\rho^i = (\mathbf{r}^i, \mathbf{z}^i)$
 5: Update $\mathbf{z}^i$ on depot
 6: –SYNCHRONIZE– (Client sums $\rho^i$)
 7: Read $\mathbf{z}^i$ and $\mathbf{p}^i$ from the depot
 8: if $k = 1$ then
 9:     Set $\mathbf{p}^i = \mathbf{z}^i$
10: else
11:     Set $\beta^i = \rho/\rho_{\text{old}}$
12:     Set $\mathbf{p}^i = \mathbf{z}^i + \beta^i \mathbf{p}^i$
13: end if
14: Compute $\mathbf{q}^i = A\mathbf{p}^i$
15: Compute $\alpha^i = \rho/(\mathbf{p}^i, \mathbf{q}^i)$
16: Update $\mathbf{p}^i$ and $\mathbf{q}^i$ on depot
17: –SYNCHRONIZE– (Client sums $\alpha^i$)
18: Read $\mathbf{x}^i, \mathbf{r}^i, \mathbf{p}^i$, and $\mathbf{q}^i$ from depot
19: Set $\mathbf{x}^i = \mathbf{x}^i + \alpha \mathbf{p}^i$
20: Set $\mathbf{r}^i = \mathbf{r}^i - \alpha \mathbf{q}^i$
21: Update $\mathbf{x}^i$ and $\mathbf{r}^i$ on depot
22: Check convergence; continue if necessary
23: Clients sets $\rho_{\text{old}} = \rho$
```

Algorithm 3 Preconditioned CG; Chronopoulos/Gear variant; Task i.

```
Require: Handles to $\mathbf{x}, \mathbf{r}, \mathbf{w}, \mathbf{p}, \mathbf{q}$, and $\mathbf{s}$ on IBP data depot and parameters $\alpha$ and $\beta$.
Ensure: Preconditioner $K$ and initial values: Solve $\mathbf{w}$ from $K\mathbf{w} = \mathbf{r}$; $\mathbf{s} := A\mathbf{v}$; $\rho := (\mathbf{r}, \mathbf{w})$; $\mu := (\mathbf{s}, \mathbf{w})$; $\alpha := \rho/\mu$.
// Each server i performs the following:
Read $\mathbf{x}^i$ and $\mathbf{p}^i$ from depot
Depending on bandwidth of matrix read appropriate portions of vectors $\mathbf{q}, \mathbf{r}, \mathbf{w}$, and $\mathbf{s}$.
Set $\mathbf{p}^i = \mathbf{w}^i + \beta \mathbf{p}^i$
Set $\mathbf{q}^i = \mathbf{s}^i + \beta \mathbf{q}^i$
Set $\mathbf{x}^i = \mathbf{x}^i + \alpha \mathbf{p}^i$
Set $\mathbf{r}^i = \mathbf{r}^i - \alpha \mathbf{q}^i$
Check convergence; continue if necessary
Solve $\mathbf{w}^i$ from $K\mathbf{w}^i = \mathbf{r}^i$
Compute $\mathbf{s}^i = A\mathbf{w}^i$
Compute $\rho^i = (\mathbf{r}^i, \mathbf{w}^i)$
Compute $\mu^i = (\mathbf{s}^i, \mathbf{w}^i)$
Update $\mathbf{x}^i, \mathbf{r}^i, \mathbf{w}^i, \mathbf{p}^i, \mathbf{q}^i$, and $\mathbf{s}^i$ on depot
Return $\rho^i$ and $\mu^i$ to client
–SYNCHRONIZE–
Client sums $\rho^i$ and $\mu^i$ for all $i$
Client sets $\beta = \rho/\rho_{\text{old}}$
Client computes $\alpha = \rho/(\mu - \rho\beta/\alpha)$
Client sets $\rho_{\text{old}} = \rho$
```

that there is an additional synchronisation point at line 21. By simply rearranging terms, this can be avoided. Note that, in the context of grid computing and iterative methods, the number of synchronisation points should be kept to an absolute minimum. In our case, synchronisation does not only involve returning the partial inner products to the client, but also the (expensive) transfer of large vectors to and from the depot. Therefore, this rearrangement of terms is neither trivial nor futile.

To increase granularity, we have also implemented the Chronopoulos/Gear variant of preconditioned CG (Chronopoulos and Gear, 1989; Dongarra *et al.*, 1998), which has a single synchronisation point, see Algorithm 3. This scheme introduces additional $2n$ flops in each iteration step compared with the original scheme. Generating the matrix resulting from discretising the Poisson equation with varying density requires a significant amount of computation, increasing granularity even further.

Preconditioning. The efficiency of iterative methods highly depends on the quality of the preconditioner, especially for very large systems. In parallel computing, efficient parallelisation of a sophisticated preconditioner is a difficult problem, but becomes even more so in the context of grid computing. We therefore choose two traditional preconditioners that do not require any communication in the parallel context, which are Jacobi (diagonal scaling) and block Jacobi. For the latter, the subdomains are solved accurately using standard CG with incomplete Cholesky preconditioning.

2.5. Implementation details.

In this section we will discuss some specific issues concerning the various implementations. In the normal operation of GridSolve in combination with DSI, if an input parameter of a task is a DSI file handle, the middleware automatically retrieves the relevant data from the IBP depot before the task is started on the server. For our purposes, a task needs full control over a DSI file, so instead we pass the DSI file handle explicitly.

Also, in the current implementation of IBP, reading and writing from and to the IBP depot are blocking operations (Zheng *et al.*, 2004). Although read operations by different tasks can be performed on the same DSI file concurrently, write operations cannot, even when the write regions do not overlap. In the Chronopoulos/Gear scheme, a task has to perform six write operations sequentially. Hence, if a single DSI file is used to store data, large communication imbalance may occur in this case. We try to overcome this imbalance by using separate DSI files for each vector and letting each task update the vectors in random order. By using the DSI functionality, it is also theoretically possible to interrupt the CG iteration process and restart at a later date, using possibly different computational resources.

At the end of each iteration step of the Chronopoulos/Gear variant, it may happen that DSI data are inadvertently overwritten. Specifically, we cannot guarantee that every task had finished reading the data from the previous iteration before other tasks updated the new data. We therefore use two different DSI files representing the previous and current data, and let the client swap the corresponding file handles at the end of each iteration step.

Furthermore, each server node in our experimental setup has ATLAS (Whaley and Petitet, 2005) as a BLAS implementation used for the various `axpy` and inner product operations. Each task recomputes its portion of the sparse coefficient matrix every iteration step and stores it using the Incremental Compressed Row Storage (ICRS) format. The implementation of this format in C is somewhat faster than that of CRS (Bisseling, 2004).

To avoid the additional overhead of communicating matrix elements, we used a matrix-free approach. This is naturally suited for linear systems with specific classes of coefficient matrices, such as Poisson and Toeplitz matrices.

3. Numerical experiments

3.1. Introduction.

In the previous sections we discussed several implementations and various suggestions for increasing granularity. In this section we will perform numerical experiments and investigate the effect of these suggestions on the performance. The experiments are divided into three parts. In the first part we investigate the difference between the standard CG method and the Chronopoulos/Gear variant and experiment with some features of the DSI mechanism. The implementation with the best results is then used for the remainder of the experiments. The second part describes experiments in a heterogeneous computing environment, and in the last part we conduct overall performance experiments using two types of preconditioning.

The residual at iteration step k is defined as $r_k = b - Ax_k$. As the starting vector we take $x_0 = \mathbf{0}$, and we use the termination criteria $||r_k||_2/||b||_2 < 10^{-6}$.

In parallel and distributed computing, speed-up is investigated by solving a problem of fixed size using an increasing number of nodes. In the context of grid computing, it is more natural to fix the problem size *per server* and investigate the scalability of the algorithm by adding more servers in order to solve bigger problems. We will analyse the algorithm using both approaches.

3.2. Target hardware.

In order to properly investigate the effectiveness of the proposed algorithm on grid hardware, two different grid testbeds are used: a local non-dedicated cluster with varying workloads and a dedicated multi-cluster of geographically separated clusters.

The first testbed is a local cluster of computers, which is a multi-user system consisting of nodes with different processors and dynamic workloads. The servers in the network are ten single core (AMD Athlon 64 Processor 3700 at 2.4GHz) and two dual core CPU nodes (Intel Core 2 CPU 6700 at 2.66GHz) with 3 GB and 8 GB of memory, respectively, and running Linux 2.6. In some cases we aim to perform controlled and repeatable experiments so we use idle processors and, as a result, the partitioning is fixed and homogeneous throughout these experiments. In most of the experiments we measure the wall clock times of five CG steps for different values of n.

The second testbed is the Distributed ASCI Supercomputer 3 (DAS-3), which is a cluster of five geographically separated clusters spread over four academic institutions in the Netherlands (Seinstra and Verstoep, 2007). The five sites are connected through SURFnet, which is a Dutch academic and research network. Four of the five local clusters are equipped with both Gigabit Ethernet interconnection and high speed Myri-10G interconnection. The TUD site only employs Gigabit Ethernet interconnection. Although each separate cluster is relatively homogeneous, the system as a whole can be considered heterogeneous.

More specific details on the five sites are given in Table 1, while Table 2 lists average roundtrip measurements between several DAS-3 sites on a lightly loaded network.

Table 1. Specific details on the five DAS-3 sites.

Site	Nodes	Speed	Network
VU	85	2.4 GHz	Myri–10G/GbE
LU	32	2.6 GHz	Myri–10G/GbE
UvA	41	2.2 GHz	Myri–10G/GbE
TUD	68	2.4 GHz	GbE
UvA–MN	46	2.4 GHz	Myri–10G/GbE

Table 2. Average roundtrip measurements (in ms) between several DAS-3 sites, with the exception of the TUD site.

	VU	LU	UvA	UvA–MN
VU	—	1.919	0.708	—
LU	1.920	—	1.246	—
UvA	0.707	1.242	—	0.039
UvA–MN	—	—	0.029	—

These facts show that a large amount of heterogeneity exists between the sites with respect to the computational resources and network capabilities, making DAS-3 a perfect testbed for grid computing.

On both testbeds, the client, the IBP server, and the agent are running on the same node while the servers are started on the remaining nodes. In a typical grid environment the client program would be located on the users' desktop machine. On the local cluster, the depot is started on a randomly chosen node, while on DAS-3, the depot is located on the head node of the VU site. As a result, we expect a significant communication overhead.

3.3. Preliminary testing. In the set of experiments on the local cluster we differentiate between the following three implementations:

(i) standard preconditioned CG using a single DSI file,

(ii) Chronopoulos/Gear scheme using a single DSI file, and

(iii) Chronopoulos/Gear CG using six separate DSI files for each vector which are manipulated in random order.

Jacobi preconditioning is used in every experiment and we fix the number of CG iterations to five. Figure 3(a) shows the total wall clock time of the second implementation for different values of n using up to eight servers. It also demonstrates that communication overhead is an issue particularly for small n, which is hardly surprising. In this case, using more servers does not result in improved execution times and even results in larger wall clock times due to the communication overhead. For large n, this implementation performs slightly better. The other implementations give similar results for small n, and we will therefore concentrate on results for large systems.

In Fig. 3(b) results are given of the three different implementations for $n = 4 \cdot 10^6$. Here we clearly see the improved performance of the Chronopoulos/Gear variants for a large number of servers. In other words, our attempts to improve granularity resulted in speed-up, albeit modest. Nevertheless, in the context of grid computing these are encouraging results.

Although we observe that using separate DSI files for the vectors only improved the overall running time of the Chronopoulos/Gear scheme for some number of servers, we noted that in this case the tasks finish at roughly the same time, in contrast to the case of using a single DSI file. This is illustrated in Fig. 4, where the wall clock times of the separate tasks for seven servers are shown after five CG steps of Chronopoulos/Gear, broken down in computation and communication. Note that Fig. 4(b) shows that by using separate DSI files the communication becomes more balanced, which is an encouraging result.

When using a single DSI file, we arrive at unbalanced wall clock times for each task, as shown in Fig. 4(a) which can be explained as follows. Although the client initiates a sequence of non-blocking calls, at the end of (in particular) the first task the updates to the DSI file appear to block subsequent updates by other tasks. These figures also clearly reveal the amount of the communication overhead.

For the remainder of the experiments we will use the third implementation, which uses separate DSI files for each vector.

3.4. Heterogeneous environment. In this section we perform heterogeneous experiments on the two testbeds. On the local cluster we artificially simulate workload, which affects the partitioning of the computational work. On DAS-3 no workload is simulated, but the differences in processor speed have a similar effect on the partitioning.

It is not trivial to perform repeatable experiments in a heterogeneous computing environment. Instead, we will give a brief qualitative analysis of the processes involved and present a typical execution of the resource-aware partitioning scheme and its effect on the CG iteration process.

On the local cluster, we artificially simulate varying workload by running a special process on each server. This process alternates between repeatedly performing a large matrix-vector multiplication and idling for a random number of seconds.

Note that the current resource-aware partitioning scheme is incompatible with block Jacobi preconditioning, because in this case the work done by each server is disproportional to the number of rows. As a result, Jacobi preconditioning is used for these experiments.

We fix the number of servers to four with approximately one million equations per server. In Fig. 5(a) the workload of each server is shown at the beginning of each

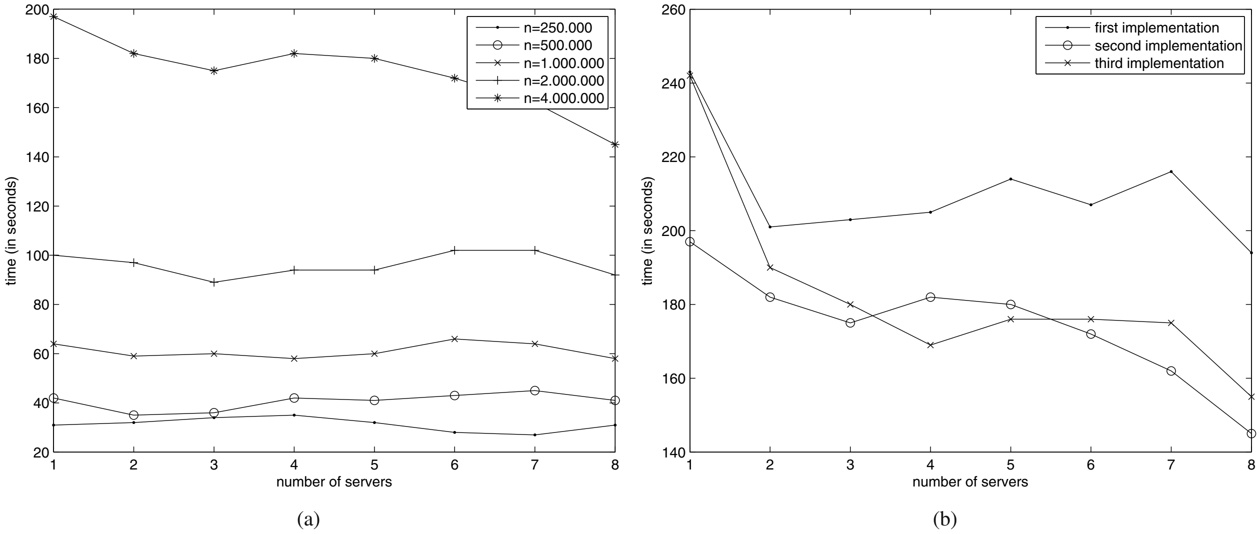


Fig. 3. Wall clock times of CG implementations in GridSolve on the local cluster: different problem sizes, implementation (ii), up to eight servers (a), $n = 4 \cdot 10^6$, all three implementations, up to eight servers (b).

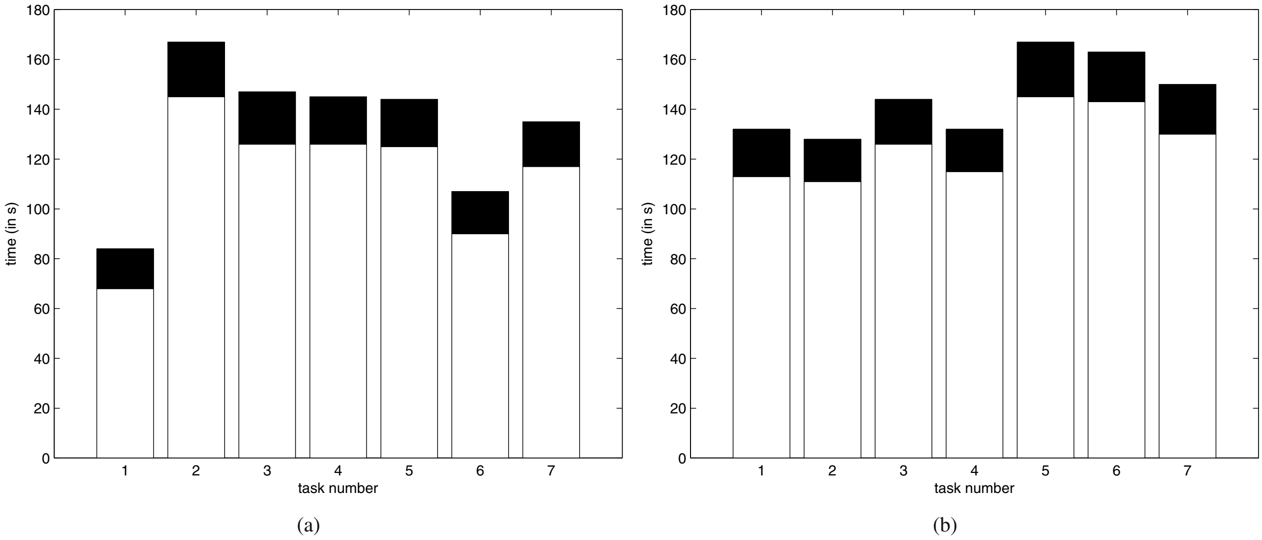


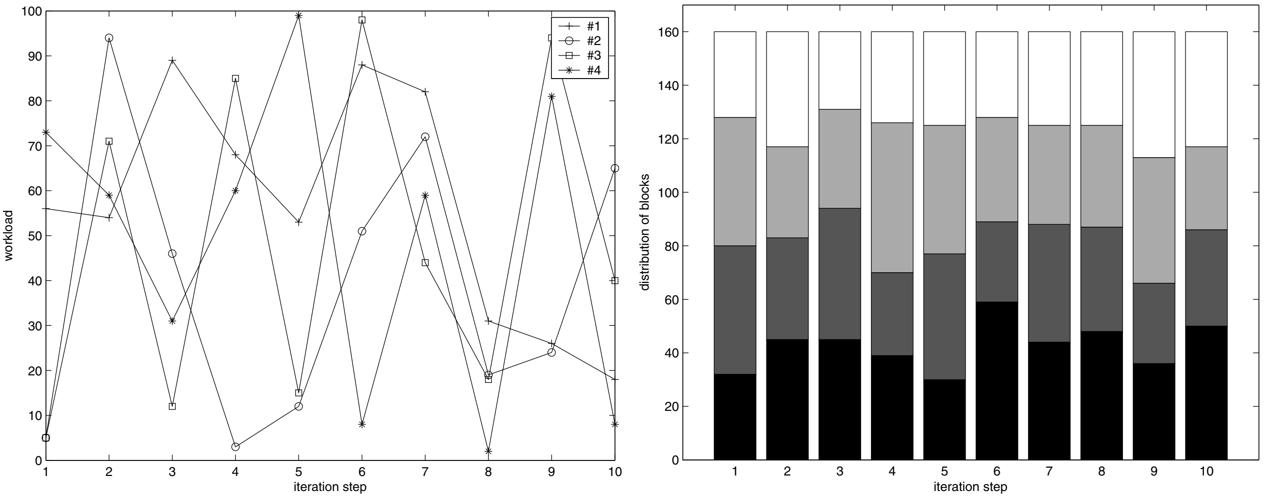
Fig. 4. Breakdown of wall clock time of tasks in communication (bottom part) and computation, for $n = 4 \cdot 10^6$ and using seven servers: utilising a single DSI file (a), utilising multiple DSI files (b).

iteration step as observed by the GridSolve agent. Figure 5(b) shows the corresponding distribution of the matrix and vector row blocks, where the tasks are numbered from top to bottom.

The graphs clearly show the effect of the varying workloads on the distribution. For example, at the sixth iteration step, Server 4 is only slightly occupied and, as a result, Task 4 has the largest size.

To investigate the effect of the partitioning strategy on the execution time of the algorithm in an artificial heterogeneous computing environment, we measured the wall clock times of five iteration steps using either homogeneous or heterogenous partitioning.

However, extensive experiments showed that the latter strategy only had a moderate effect on the total execution time. Due to the low computation to communication ratio, the current partitioning algorithm mostly affects the amount of communication of each task with the depot. As processor workload has barely any influence on these kinds of operations, the *total* execution time does not decrease significantly when using heterogeneous partitioning on heterogeneous computational resources.



(a)

(b)

Fig. 5. Heterogeneous experiments with the 3D bubbly flow problem (local cluster): workload (a), distribution (b).

There are two possible solutions within the current context. Either incorporate network bandwidth information in the partitioning algorithm, or try to increase the computation to communication ratio in combination with an appropriate computational resource-aware partitioning strategy.

Since DAS-3 is a dedicated machine, the nodes have zero workload. Therefore, the partitioning on DAS-3 is based solely on the heterogeneity in the processor speeds and is fixed throughout the whole iteration process. Not surprisingly, the effect on the wall clock times of five iteration steps is similar to that of the local cluster results.

3.5. Parallel performance. In the previous sections, we investigated various aspects of the algorithm separately. In this section, we present overall parallel performance results using two preconditioning techniques on both the local cluster and DAS-3, *without any workload*. First, we fix the problem size to $n = 120^3$ and investigate speed-up using up to six servers on the local cluster. Figure 6(a) shows the total wall clock time until convergence is obtained. In Fig. 7(a) speed-up results are given on DAS-3 for $n = 25^3$. A server is started on a randomly chosen node on each cluster of DAS-3. The client is located on the head node of the VU site.

Using a more sophisticated preconditioning technique like block Jacobi improves the computation to communication ratio and reduces the total number of CG iterations. However, it also negatively influences the manner in which the total number of CG iterations depends on the number of subdomains. This is in contrast to using diagonal scaling as a preconditioner, where the total number of iterations is independent of the number of subdomains.

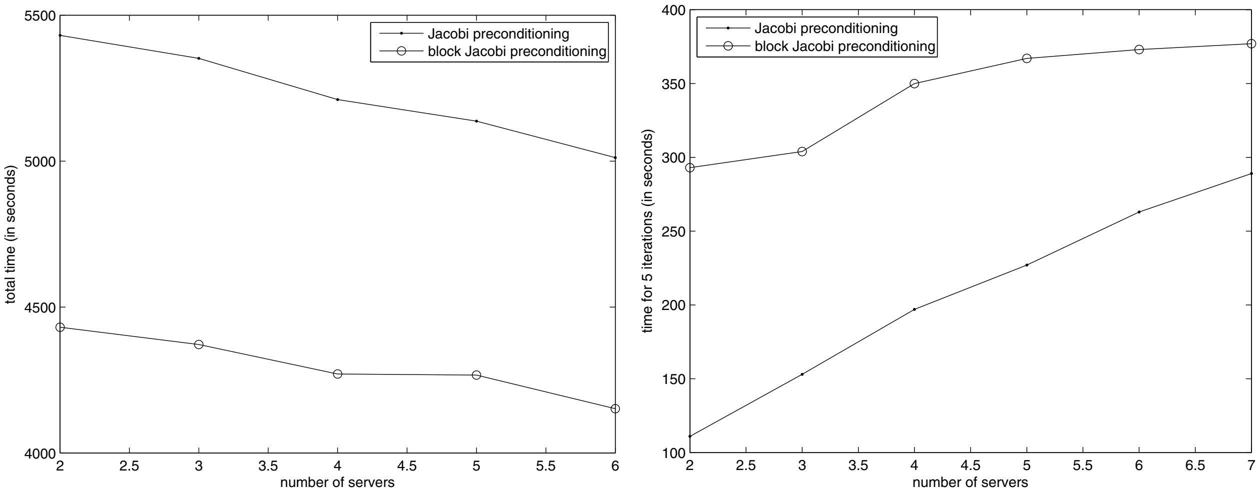
We investigate the scalability of the algorithm by setting the problem size per server to $1{,}000{,}000$ equations and performing *five CG steps* using both Jacobi and block Jacobi as a preconditioner. This experiment gives an indication on how fast the communication overhead grows. The results for the local cluster are given in Fig. 6(b), while Fig. 7(b) shows results for DAS-3.

Using Jacobi preconditioning results in a highly unfavourable computation to communication ratio, and the results show that the (communication) overhead grows quite fast, which is not a surprising result. On the other hand, block Jacobi preconditioning has a more favourable ratio, which is indicated by the reduced increase in execution time.

4. Concluding remarks and future work

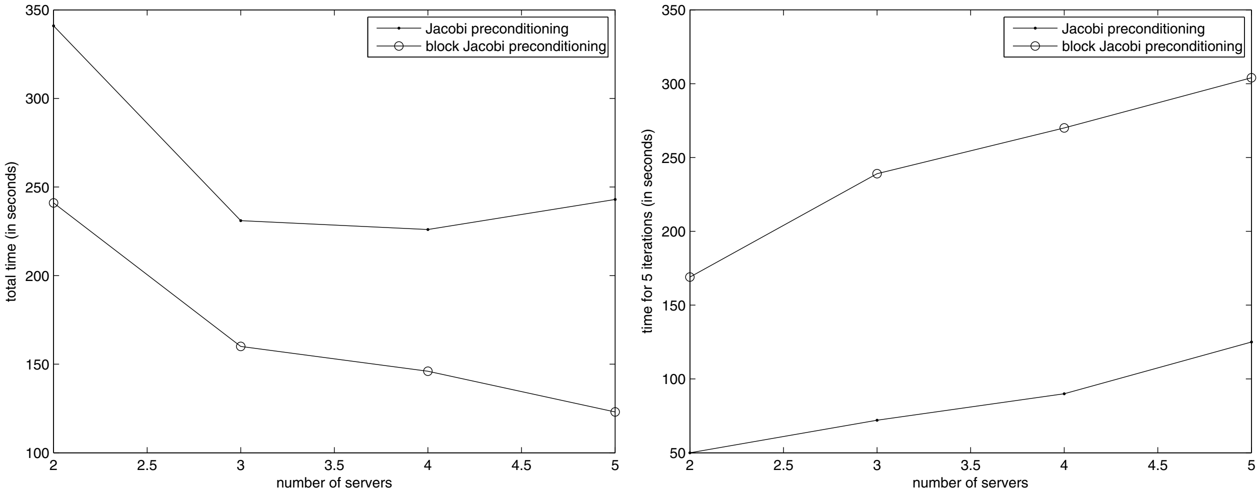
4.1. Conclusions. The efficient iterative solution of large sparse linear systems on aggregated computational resources is a difficult problem. While parallel implementation of iterative methods in the context of dedicated parallel computing is relatively well understood, the design of efficient iterative algorithms for the solution of large linear systems on non-dedicated and heterogeneous networks of computers is still in its infancy and much research is needed.

The key algorithmic constraint of CG methods in grid computing is the inner product. To be more specific, the computation of an inner product in parallel iterative algorithms induces a global synchronisation point. While such an operation can be highly complex even on dedicated and homogeneous parallel computers connected by a high-speed network, it may become the critical bottleneck in a non-dedicated and heterogeneous computing en-



(a) (b)

Fig. 6. Comparison of two preconditioning techniques (local cluster): speed-up experiments (a), scaled experiments (b).



(a) (b)

Fig. 7. Comparison of two preconditioning techniques (DAS-3): speed-up experiments (a), scaled experiments (b).

vironment. For example, within the context of GridSolve, the non-persistent data model forces us to transfer at each synchronisation point large amounts of data over a possibly unreliable and slow network connection.

In this work we described two implementations of the preconditioned conjugate gradient method using the mature grid middleware GridSolve. We evaluated the implementations on heterogeneous computing hardware, applied to a realistic test problem. Using the middleware we also implemented a simple architecture-aware partitioning algorithm to divide the computational work. Furthermore, by using multiple DSI files we have attempted to decrease the communication overhead within the bridge communication model currently used by GridSolve. And finally, we increased granularity by (i) using the Chronopoulos/Gear variant of CG which only has a single synchronisation point per iteration step and (ii) by devoting more work to the preconditioning phase.

We explored the current state-of-the-art in grid computing and iterative methods within the software constraints of the grid middleware GridSolve. The main bottlenecks—in both middleware and iterative methods—were identified, and algorithmic and software modifications for improving granularity were proposed and implemented, resulting in moderately improved performance and speed-up. Although the experimental results were suboptimal, they can be considered encouraging in the context of iterative solvers and grid computing.

4.2. Future work. Clearly, there is much room for improvement and we will give some suggestions for future work.

The current implementation of GridSolve forces us to use bridge communication. SmartGridSolve (Brady *et al*., 2008) is an extension of GridSolve, which performs similarly to SmartNetSolve (Brady *et al*., 2006), allowing for communication between the computational servers as well as data persistence. By combining this with sophisticated (possibly weighted) hypergraph partitioning techniques such as those used in *Mondriaan* (Vastenhouw and Bisseling, 2005), we hope to greatly improve our load balancing algorithm. Another possible improvement is incorporating information about network throughput into the partitioning algorithm.

Furthermore, possible hardware and software solutions to reduce the communication overhead include fast network connections to the IBP depot and using distributed IBP data depots (Beck *et al*., 2002).

The local subdomains in the block Jacobi preconditioner are solved accurately using another iterative method. In (Brakkee *et al*., 1997), interesting results were obtained with inaccurate subdomain solutions. Applying the same strategy to our application would require used a method that can handle a variable preconditioner, such as the flexible CG method (Axelsson, 1994). Efficient parallelisation of such a method on grid computers introduces additional difficulties. This is also subject of future work.

4.3. Related work. In metacomputing, using the appropriate middleware is of critical importance. Many different types of grid middleware exist and choosing the correct one depends on the application, target hardware, and (numerical) algorithm. The main reason for choosing GridSolve to solve the current application is two-fold: (i) the GridSolve middleware is specifically dedicated to numerical computations, and (ii) it allows easy access to remote computational resources.

Naturally, other choices exists for running parallel applications in heterogeneous computing environments. For example, MPICH-G2 is a reference MPI implementation that is designed for running MPI applications in grid environments (Karonis *et al*., 2003). It is a Globus-based library that extends MPICH. Another MPICH-based implementation that is tailored to heterogeneous networks of clusters is MPICH-Madeleine (Mercier, 2006). For real-world applications on grid hardware that use MPICH–G2, see (Wyrzykowski *et al.*, 2005; 2006; Boghosian *et al.*, 2006; Dong *et al.*, 2005; Mirghani *et al.*, 2005).

Grid middleware such as MPICH-G2 provides a convenient method of cross-site MPI-based communication, while the GridSolve middleware is based on the RPC model. Another key difference is that GridSolve utilises the DSI for (bridge) communication, whereas MPICH-G2 allows direct communication between the nodes.

In this paper we tried to realise the full potential of a *completely synchronous* parallel subspace method for solving large sparse linear systems on grid computers. Synchronous parallel iterative algorithms are methods where at each iteration step information is needed from the previous iteration step. As has been shown previously and exemplified by the experimental results, the fine-grain nature and potentially large number of synchronisations of the said methods raise many efficiency issues on grid computers and limit the applicability of this approach to large-scale problems.

An efficient and effective preconditioner is crucial for fast convergence of iterative methods. Such a preconditioner is, generally speaking, the most difficult part to parallelise, especially in heterogeneous environments as found in grid computing. Within the fully synchronous context of this paper, we maximised the amount of work that can be devoted to the employed preconditioning, *without* introducing additional synchronisation points.

Parallel *asynchronous* iterative algorithms exhibit features that are extremely well-suited for grid computing, such as a lack of synchronisation points and coarse-graininess. Unfortunately, they also suffer from slow (block Jacobi method-like) convergence rates. We propose using the said asynchronous methods as a coarse-grain preconditioner in a flexible subspace method, where the preconditioner is allowed to change in each iteration step. By combining a slowly converging asynchronous inner method and a fast converging synchronous outer method, we aim to reap the benefits and awards of both techniques.

Desynchronising the preconditioning phase in this manner has the advantage that: (i) the preconditioner can be easily and efficiently parallelised on grid computers, (ii) no additional synchronisation points are introduced, and (iii) by devoting the bulk of the computational effort to the preconditioner the computation to communication ratio can be improved significantly, while considerably reducing the number of expensive (outer) synchronisations. The resulting partially asynchronous inner-outer method is analysed in depth in (Collignon and van Gijzen, 2008; 2009), with promising experimental results.

Acknowledgment

The authors would like to thank the GridSolve team for their prompt response pertaining to our questions. They also thank Rob Bisseling for careful proof-reading of the manuscript and his valuable comments. The anonymous referees are gratefully acknowledged for their critical review of the manuscript and their suggestions, which greatly enhanced the presentation. The work of the first author was supported by the Delft Centre for Computational Science and Engineering (DCSE) within the framework of the project entitled on *Development of an Immersed Boundary Method, Implemented on Cluster and Grid Computers, Application to the Swimming of Fish*, conducted jointly with Barry Koren and Yunus Hassen from CWI.

An earlier version of this paper was presented at the Workshop on Computer Aspects of Numerical Algorithms, held in Wisła, Poland, on October 15–17, 2007 (Collignon and van Gijzen, 2007).

References

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M. and Werthimer, D. (2002). SETI@home: An experiment in public-resource computing, *Communications of the ACM* **45**(11): 56–61.

Axelsson, O. (1994). *Iterative Solution Methods*, Cambridge University Press, New York, NY.

Beck, M., Arnold, D., Bassi, A., Berman, F., Casanova, H., Dongarra, J., Moore, T., Obertelli, G., Plank, J., Swany, M., Vadhiyar, S. and Wolski, R. (2002). Middleware for the use of storage in communication, *Parallel Computing* **28**(12): 1773–1787.

Bisseling, R. H. (2004). *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, Oxford University Press, New York, NY.

Boghosian, B., Coveney, P., Dong, S., Finn, L., Jha, S., Karniadakis, G. and Karonis, N. (2006). Nektar, SPICE and Vortonics: Using federated grids for large scale scientific applications, *Workshop on Challenges of Large Applications in Distributed Environments (CLADE)/15th International Symposium on High Performance Distributed Computing (HPDC-15), Paris, France,* pp. 34–42.

Brady, T., Guidolin, M. and Lastovetsky, A. (2008). Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model, *9th IEEE/ACM International Conference on Grid Computing, Tsukuba, Japan,* pp. 49–56.

Brady, T., Konstantinov, E. and Lastovetsky, A. (2006). SmartNetSolve: High level programming system for high performance Grid computing, *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Rhodes Island, Greece*, (on CD-ROM).

Brakkee, E., Vuik, C. and Wesseling, P. (1997). Domain decomposition for the incompressible Navier–Stokes equations: Solving subdomain problems accurately and inaccurately, *in* R. Glowinski, J. Périaux and Z-C. Shi and O. Widlund (Eds), *Domain Decomposition Methods in Sciences and Engineering*, John Wiley & Sons, Chichester, pp. 443–451.

Caron, E., Del-Fabbro, B., Desprez, F., Jeannot, E. and Nicod, J.-M. (2005). Managing data persistence in network enabled servers, *Scientific Programming* **13**(4): 333–354.

Caron, E. and Desprez, F. (2006). DIET: A scalable toolbox to build network enabled servers on the grid, *International Journal of High Performance Computing Applications* **20**(3): 335–352.

Chronopoulos, A. T. and Gear, C. W. (1989). *S*–step iterative methods for symmetric linear systems, *Journal of Computational and Applied Mathematics* **25**(2): 153–168.

Collignon, T. P. and van Gijzen, M. B. (2007). Implementing the conjugate gradient method on a grid computer, *Proceedings of the International Multiconference on Computer Science and Information Technology, Wisła, Poland*, Vol. 2, pp. 527–540.

Collignon, T. P. and van Gijzen, M. B. (2008). Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner, *Technical report DUT 08–08*, Delft University of Technology, Delft.

Collignon, T. P. and van Gijzen, M. B. (2010). Parallel scientific computing on loosely coupled networks of computers, *in* B. Koren and C. Vuik (Eds), *Advanced Computational Methods in Science and Engineering*, Lecture Notes in Computational Science and Engineering, Vol. 71, Springer, Berlin, pp.79–106.

Desprez, F. and Jeannot, E. (2004). Improving the GridRPC model with data persistence and redistribution, *ISPDC '04: Proceedings of the 3rd International Symposium on Parallel and Distributed Computing/3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04), Cork, Ireland*, pp. 193–200.

Dong, S., Karniadakis, G. E. and Karonis, N. T. (2005). Cross-site computations on the TeraGrid, *Computing in Science and Engineering* **7**(5): 14–23.

Dongarra, J. J., Duff, I. S., Sorensen, D. C. and van der Vorst, H. A. (1998). *Numerical Linear Algebra for High Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA.

Dongarra, J., Li, Y., Shi, Z., Fike, D., Seymour, K. and YarKhan, A. (2007). Homepage of NetSolve/GridSolve, `http://icl.cs.utk.edu/netsolve/`.

Foster, I. and Kesselman, C. (2004). *The Grid: Blueprint for a New Computing Infrastructure, 2nd Edn.*, Morgan Kaufman Publishers, San Fransisco, CA.

Hestenes, M. R. and Stiefel, E. (1952). Methods of conjugate gradients for solving linear systems, *Journal of Research of National Bureau Standards* **49**(6): 409–436.

Karonis, N. T., Toonen, B. and Foster, I. (2003). MPICH-G2: A grid-enabled implementation of the message passing interface, *Journal of Parallel and Distributed Computing* **63**(5): 551–563.

Lastovetsky, A., Zuo, X. and Zhao, P. (2005). A non-intrusive and incremental approach to enabling direct communications in RPC-based grid programming systems, *Technical report*, UCD School of Computer Science and Informatics, Dublin, `http://www.csi.ucd.ie/content/non-intrusive-and-incremental-approach`

-enabling-direct-communications-rpc -based-grid-program.

Lee, C., Nakada, H. and Tanimura, Y. (2007). GridRPC Working Group, http://forge.ogf.org/sf/projects/gridrpc-wg/.

Mercier, G. (2006). MPICH-Madeleine. An MPI implementation for heterogeneous clusters of clusters, http://runtime.futurs.inria.fr/mpi/.

Mirghani, B., Tryby, M., Baessler, D., Karonis, N., Ranhthan, R. and Mahinthakumar, K. (2005). Development and performance analysis of a simulation-optimization framework on TeraGrid Linux clusters, *6th LCI International Conference on Linux Clusters: The HPC Revolution 2005, Chapel Hill, NC, USA*.

Mittal, R. and Iaccarino, G. (2005). Immersed boundary methods, *Annual Review of Fluid Mechanics* **37**: 239–261.

Peskin, C. (2002). The immersed boundary method, *Acta Numerica* **11**: 479–517.

Sato, M., Boku, T. and Takahashi, D. (2003). OmniRPC: A Grid RPC system for parallel programming in cluster and Grid environment, *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan*, pp. 206–213.

Seinstra, F. J. and Verstoep, K. (2007). DAS–3: The distributed ASCI supercomputer 3, http://www.cs.vu.nl/das3/.

Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C. and Casanova, H. (2002). Overview of GridRPC: A remote procedure call API for grid computing, *GRID '02: Proceedings of the 3rd International Workshop on Grid Computing, Baltimore, MD, USA*, pp. 274–278.

Seymour, K., YarKhan, A., Agrawal, S. and Dongarra, J. (2005). NetSolve: Grid enabling scientific computing environments, *in* L. Grandinetti (Ed.), *Grid Computing and New Frontiers of High Performance Processing*, Elsevier, New York, NY.

Tanaka, Y., Nakada, H., Sekiguchi, S., Suzumura, T. and Matsuoka, S. (2003). Ninf-G: A reference implementation of RPC-based programming middleware for Grid computing, *Journal of Grid Computing* **1**(1): 41–51.

Tang, J. M. and Vuik, C. (2007a). On deflation and singular symmetric positive semi-definite matrices, *Journal of Computational and Applied Mathematics* **206**(2): 603–614.

Tang, J. and Vuik, C. (2007b). Efficient deflation methods applied to 3-D bubbly flow problems, *Electronic Transactions on Numerical Analysis* **26**: 330–349.

van der Pijl, S., Segal, A., Vuik, C. and Wesseling, P. (2005). A mass-conserving level-set method for modelling of multi-phase flows, *International Journal for Numerical Methods in Fluids* **47**: 339–361.

van Kan, J. (1986). A second-order accurate pressure correction scheme for viscous incompressible flow, *SIAM Journal on Scientific and Statistical Computing* **7**(3): 870–891.

Vastenhouw, B. and Bisseling, R. H. (2005). A two-dimensional data distribution method for parallel sparse matrix-vector multiplication, *SIAM Review* **47**(1): 67–95.

Whaley, R. C. and Petitet, A. (2005). Minimizing development and maintenance costs in supporting persistently optimized BLAS, *Software: Practice and Experience* **35**(2): 101–121.

Wyrzykowski, R., Meyer, N., Olas, T., Kuczynski, L., Ludwiczak, B., Czaplewski, C. and Oldziej, S. (2009). Meta–computations on the CLUSTERIX grid, *in* B. Kågström, E. Elmroth, J. Dongarra and J. Wasniewski (Eds), *Applied Parallel Computing: State of the Art in Scientific Computing. 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers,* Lecture Notes in Computer Science, Vol. 4699, Springer, Berlin/Heidelberg, pp. 489–500.

Wyrzykowski, R., Meyer, N. and Stroinski, M. (2005). Concept and implementation of CLUSTERIX: National cluster of linux systems, *6th LCI International Conference on Linux Clusters: The HPC Revolution 2005, Chapel Hill, NC, USA*.

YarKhan, A., Seymour, K., Sagi, K., Shi, Z. and Dongarra, J. (2006). Recent developments in GridSolve, *International Journal of High Performance Computing Applications (IJHPCA)* **20**(1): 131–141.

Zheng, Y., Bassi, A., Beck, M., Plank, J. S. and Wolski, R. (2004). Internet Backplane Protocol: C API 1.4, *Technical report,* Department of Computer Science, University of Tennessee, Knoxville, TN.

Zuo, X. and Lastovetsky, A. (2007). Experiments with a software component enabling NetSolve with direct communications in a non-intrusive and incremental way, *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, USA*.

Tijmen P. Collignon obtained his M.Sc. degree in scientific computing at Utrecht University in 2006. Currently he is a Ph.D. student in the numerical analysis research group of the Delft Institute of Applied Mathematics at the Delft University of Technology in the Netherlands. His main research areas are numerical linear algebra and grid computing.

Martin B. van Gijzen received his M.Sc. degree in applied mathematics in 1989, and his Ph.D. in 1994, both from the Delft University of Technology in the Netherlands. From 1994 until 1996 he was a research associate at Utrecht University, and from 1997 until 2001 he was a project leader of several European research projects on underwater communication at the TNO Physics and Electronics Laboratory. In 2001 he moved to France to become a senior scientist in the parallel computing group at CERFACS in Toulouse. In 2004 he returned to the Delft University of Technology, where he is an associate professor. His research areas are numerical linear algebra and high performance computing.

Received: 19 February 2009
Revised: 26 August 2009