DELFT UNIVERSITY OF TECHNOLOGY

REPORT 08-08

Solving Large Sparse Linear Systems Efficiently on Grid Computers using an Asynchronous Iterative Method as a Preconditioner

T. P. Collignon and M. B. van Gijzen

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2008

Copyright © 2008 by Delft Institute of Applied Mathematics Delft, The Netherlands.

No part of the Journal may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission from Delft Institute of Applied Mathematics, Delft University of Technology, The Netherlands.

Solving Large Sparse Linear Systems Efficiently on Grid Computers using an Asynchronous Iterative Method as a Preconditioner

Abstract

In this paper we describe an efficient iterative algorithm for solving large sparse linear systems on Grid computers and review some of its advantages and disadvantages. The algorithm is a combination of a synchronous flexible outer iterative method and a coarse–grain asynchronous inner iterative method as a preconditioner. We present results of a complete implementation using mature Grid middleware, applied to a 3D convection–diffusion problem. Experiments are performed in a heterogeneous computing environment.

Key words. Parallel and Distributed Computing, Numerical Algorithms for CS&E, Grid Computing (middleware, algorithms, performance evaluation), Sparse linear systems, Preconditioning.

1 Introduction

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

An obvious strategy for providing cheap parallel computing power is to use the available non–dedicated hardware, and thus to make better use of existing resources. This idea has given rise to the concept of Grid computing [1]. 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 and/or clusters at geographically separated sites that are connected via the Internet. This approach has proven to be successful for embarrassingly parallel applications where the tasks do not require interprocessor communication, as exemplified by the well–known SETI@home project [2].

For the 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 servers and erratic network behaviour present new algorithmic challenges.

In this paper we describe an efficient iterative method for solving large linear systems that is designed to exploit the characteristics of both Grid computing and parallel computing. More specifically, it is a combination of the flexible iterative method GMRESR [3], and an asynchronous iterative method [4] as preconditioner.

Much work has been done on applying asynchronous iterative algorithms to solve linear systems on Grid computers, but the slow block Jacobi–like convergence rate of asynchronous methods limits the applicability of such approaches [4, 5]. By using an asynchronous method as a coarse–grain preconditioner in a flexible iterative method, we expect to improve overall convergence rate and to extend the range of applications.

A full implementation of the algorithm using the Grid middleware GridSolve [6, 7] will be presented. Depending on the application, the outer iteration may be performed either sequentially on a single master node or in parallel. The asynchronous preconditioner is implemented in a coarse–grained parallel manner. Numerical experiments are performed on a local heterogeneous cluster applied to a realistic test problem.

2 Sparse linear solvers in Grid environments

This section starts by giving a brief overview of the complete algorithm. Then, in Sect. §2.2 the Grid middleware GridSolve is discussed describing the various components along with some of its advantages and disadvantages. Section §2.3 describes specific details pertaining to the parallel implementation in GridSolve. Finally, Sect. §2.4 reviews some of the advantages and disadvantages of using an asynchronous iterative method as a preconditioner.

2.1 GMRESR with asynchronous preconditioning

We are interested in designing efficient iterative methods for solving large sparse linear systems,

$$Ax = b, \quad \text{with a non–symmetric, non–singular matrix } A, \tag{1}$$

on heterogeneous networks of computers. Parallel asynchronous iterative methods possess several characteristics that are perfectly suited for Grid computing, such as lack of synchronisation points [8]. Unfortunately, they also have significant drawbacks, such as slow convergence rates [4]. We propose to use an asynchronous method as a coarse–grain preconditioner in a flexible iterative method, in which the preconditioner is allowed to change in each iteration step. By using a slowly converging asynchronous method as a preconditioner in a fast converging flexible method we expect to achieve overall fast convergence.

We choose GMRESR as a flexible method, partly because the orthogonalisation process can be easily truncated, which is essential for practical implementations. The truncated variant of GMRESR is shown in Alg. 1. The preconditioning step in the second line

Algorithm 1 GMRESR (truncated version)

INPUT: Parameters $m, \epsilon_{\text{in}}, j_{\text{in}}, T_{\max}$; Initial guess x_0; Set $r_0 = b - Ax_0$.
OUTPUT: Approximate solution to $Ax = b$.

1: **for** $k = 0, 1, \ldots,$ until convergence **do**
2: Evaluate $u = \mathcal{M}(r_k, \epsilon_{\text{in}}/j_{\text{in}}, T_{\max})$;
3: Compute $c = Au$;
4: Compute $[c_k, u_k] = \text{orthonorm}(c, u, c_i, u_i, k, m)$;
5: Compute $\gamma = c_k^\top r_k$;
6: Update $x_{k+1} = x_k + \gamma u_k$;
7: Update $r_{k+1} = r_k - \gamma c_k$;
8: **end for**

Algorithm 2 Block Jacobi iteration on p processors.

OUTPUT: $u = \mathcal{M}(r, \epsilon_{\text{in}}/j_{\text{in}}, T_{\max})$
1: Initialize $u(0)$;
2: **for** $t = 1, 2, \ldots,$ until $T_{\max}$ **do**
3: **for** $i = 1, 2, \ldots, p$ **do**
4: Approximate $A_{ii} u_i(t) = r_i - \sum_{j=1, j \neq i}^{p} A_{ij} u_j(t-1)$;
5: **end for**
6: **end for**

computes some approximate solution for $Au = r_k$ in each outer iteration step k and is performed by an asynchronous iterative method. The obtained search direction is then orthogonalised against m previous search directions. Note that it is our intention that the bulk of the computational work is performed by the preconditioner.

Asynchronous algorithms generalise simple iterative methods such as the classical block Jacobi iteration [9], shown in Alg. 2. To compute an approximation to $Au = r$ using p processors, the coefficient matrix, the solution vector, and the right–hand side are partitioned into blocks as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix}, \quad u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_p \end{bmatrix}, \quad r = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_p \end{bmatrix}. \tag{2}$$

In the standard synchronous Jacobi iteration process, the processors operate in parallel on their part of the vector $u(t)$, followed by a synchronisation point at each iteration step t. In our asynchronous algorithm, a processor computes $u_i(t)$ using information that is available at that particular time. As a result, each separate block Jacobi iteration process may use out–of–date information, but the lack of synchronisation points and the reduction of communication can potentially result in improved parallel performance.

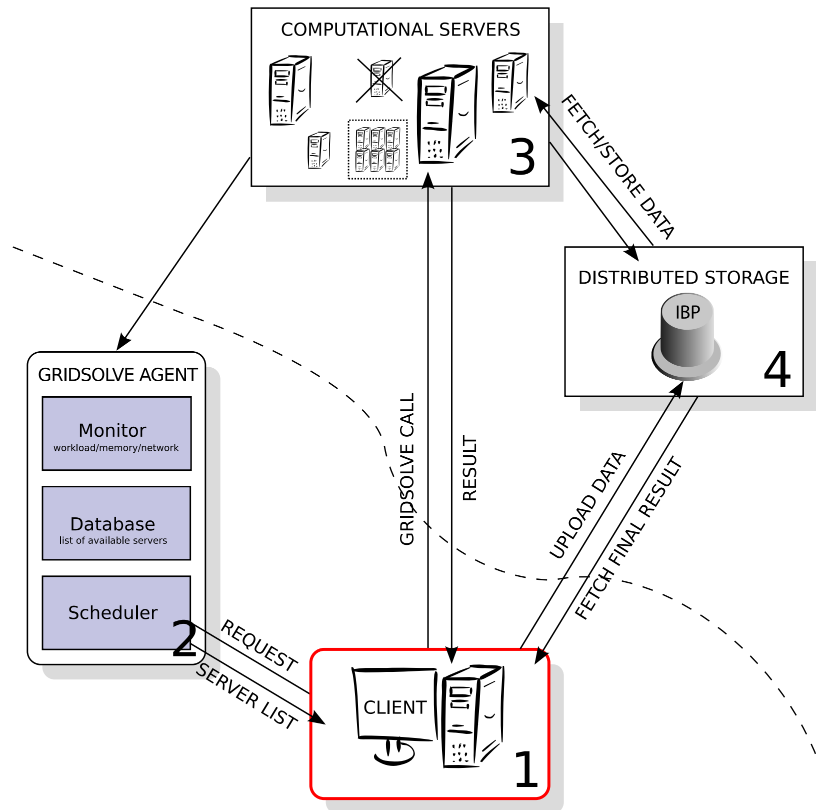


Figure 1: Schematic overview of GridSolve. The dashed line symbolises (geographical) distance between the client and servers.

Note that in practical implementations, the inner system in line 4 of Alg. 2 is often solved (possible inaccurately) by some other iterative method.

2.2 Brief description of GridSolve

GridSolve (GS) is a distributed programming system which uses a client–server model for solving complex problems remotely on global networks. It is an instantiation of the GridRPC model, a standard for a Remote Procedure Call (RPC) mechanism on Grid computers [10]. The GridRPC Application Programming Interface (API) is defined within the Global Grid Forum [11]. Other projects that implement the GridRPC API are DIET [12], NetSolve [13], Ninf–G [14], and OmniRPC [15].

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 current 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 easily added or modified without restarting the server.

A single GridSolve agent (component #2 in Fig. 1) 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, 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 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.

The main advantages of GridSolve are that it is easy to use, install, maintain, and that it is a standard for programming on Grid environments. Nevertheless, the current implementation has several limitations. For example, the remote servers cannot communicate directly. 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 are read or generated locally during the execution of a task is lost after it completes. Several strategies such as data persistence and data redistribution have been proposed to tackle these deficiencies for different implementations of the GridRPC API [16, 17, 18, 19, 20]. Furthermore, a proposal for a Data Management API within the GridRPC is currently being developed.

In GridSolve there is a partial solution to the data management 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 [21]. 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 can be sent to the server and the task can manipulate data on the IBP depot (component #4). Using the DSI can be considered as programming for a shared memory model.

2.3 Parallel implementation details

The coarse–grain nature of the asynchronous preconditioning iteration allows for efficient implementation on a Grid computer. On the other hand, the relatively fine–grain nature of the rest of the operations in the outer iteration (i.e., the matrix–vector multiplication, orthogonalisation, and vector operations) may present parallel efficiency issues in the current context.

Algorithm 3 Asynchronous block Jacobi iteration task for each server i.

OUTPUT: $u = \mathcal{M}(r_k, \epsilon_{\text{in}}/j_{\text{in}}, T_{\max})$

1: Read r_i from IBP depot; Set $t = 0$; Set $u_i^{(0)} = 0$;
2: Perform ILU decomposition of A_{ii};
3: **while** $t_{\text{elapsed}} < T_{\max}$ **do**
4: Read $\tilde{u}^{(t)}$ from IBP depot if $t \neq 0$;
5: Compute $v_i^{(t)} = r_i - \sum_j A_{ij} u_j^{(t)}$;
6: Solve $A_{ii} x_i^{(t)} = v_i^{(t)}$ with IDR(s) using j_{in} steps and/or with accuracy ϵ_{in};
7: Set $u_i^{(t+1)} = u_i^{(t)} + x_i^{(t)}$;
8: Write $u_i^{(t+1)}$ to IBP depot;
9: Set $t = t + 1$;
10: **end while**

Performing the outer iteration on a single node will eventually become a bottleneck, either in memory or computational work. As previously mentioned, the bulk of the computational work is to be performed by the asynchronous preconditioning. This implies that in order to reduce the number of outer iterations, more work may be devoted to the preconditioning by either (i) adding more servers or by (ii) increasing the time spent on each preconditioning step.

Nevertheless, for extremely large problems such an approach may become impractical. A parallel outer iteration was therefore implemented using techniques described in [22]. Currently, the matrix is partitioned using a homogeneous one–dimensional block–row distribution, both in the preconditioning iteration and in the outer iteration. The vectors are distributed accordingly. What follows are various implementation issues pertaining to performing the outer iteration in sequential or parallel.

2.3.1 Sequential outer loop.

All of the operations — with exception of the preconditioning iteration — are performed on the client machine. There is a single GridSolve task for the preconditioning step, which implies that there is a single global synchronisation point in each outer iteration step. The client machine begins by updating the complete residual on the IBP data depot. Algorithm 3 shows the specific steps performed by each server i in the preconditioning phase.

At the beginning of task i, the appropriate portion of the residual is read and the task starts iterating on its portion of u. At the end of each block Jacobi iteration step, the server updates the relevant portion(s) of u (i.e., the overlap) on the IBP depot. This process continues until some appropriate criterion is met, which is currently related to a simple time limit. Each process then writes its part of u to the IBP depot and the complete vector u is read by the client machine. The obtained search direction is then used to compute the new iterate and residual. This procedure is repeated until convergence.

Algorithm 4 Classical Gram–Schmidt

OUTPUT: $[c_k, u_k] = \text{orthonorm}(c, u, c_i, u_i, k, m)$;

1: Compute $\beta = c^\top c$;
2: **for** $i = \max(0, k-m), \ldots, k-1)$ **do**
3: Compute $\alpha_i = c^\top c_i$;
4: **end for**
5: Compute $\beta = \sqrt{\beta - \sum_{i=1}^{k-1} \alpha_i^2}$;
6: Compute $c_k = \beta^{-1} \left(c - \sum_{i=1}^{k-1} \alpha_i c_i \right)$;
7: Compute $u_k = \beta^{-1} \left(u - \sum_{i=1}^{k-1} \alpha_i u_i \right)$;

2.3.2 Parallel outer loop.

In this case, no bulk computation and communication is done by the client machine. The only data that is communicated between the client and the computational nodes are the results of the (partial) inner products.

In a parallel context, choosing an appropriate orthogonalisation procedure becomes essential. The classical Gram–Schmidt algorithm (CGS) applied to our case is shown in Alg. 4 and has good parallel properties. However, it may suffer from numerical instabilities. This may be remedied by using a selective reorthogonalisation procedure [23, 24].

By combining operations as much as possible, three distinct GridSolve tasks can be constructed, giving three synchronisation points per outer iteration step. The first task consists of two main operations: updating the iterate and residual and performing the asynchronous Jacobi iterations. The second GridSolve task has two operations: computing the local matrix–vector product and performing the first phase of the CGS algorithm. The third and last GridSolve task performs the second phase of CGS and stores the newly computed search directions.

A disadvantage of this approach is that every GridSolve task should be performed on reliable hardware. That is, should any of the tasks fail, it is likely that important intermediate information is lost, halting the entire outer iteration process. An obvious solution is to create a fourth GridSolve task solely for updating the iterate and residual. The tasks that perform the preconditioning iteration may then be executed on resources that are less reliable.

The client node is idle most of the time during the preconditioning iteration. To avoid this, it may be included in the computational nodes.

2.4 Discussion

A good preconditioner is crucial for rapid convergence of iterative methods. Generally speaking, efficient parallelisation of a preconditioner is a difficult problem, particularly in Grid environments. Asynchronous iterative methods exhibit features that are perfectly

suitable for these environments, such as: easy to parallelise, coarse–grain, overlapping communication with computation, and no synchronisation points. Also, if we choose to use a fixed number of seconds to perform the preconditioning step, there is no need for a — possibly complicated and expensive — convergence detection algorithm.

Nevertheless, asynchronous iterative method have several disadvantages. For example, they exhibit slow (block Jacobi–type) convergence rates, iterations performed on the basis of outdated information can be non–effective and possibly counterproductive, finding effective inner iteration stopping criteria is not trivial, and implementation of an efficient convergence detection algorithm is difficult.

By using the asynchronous iteration as a preconditioner in a flexible iterative method, we obtain an algorithm that is partially fault tolerant. In the preconditioning phase, each server iterates on a specific part of the new search direction. In heterogeneous computing environments, servers may become unavailable at any time, resulting in loss of computed data. If the asynchronous method is used to solve the main linear system, such an event would be disastrous. In our case, the outer iteration process will slow down in the worst case, but is otherwise unaffected.

Asynchronism in iterative methods allows for efficient overlapping of communication by computation. However, this does not automatically exclude the issue of load balancing. To avoid significant desynchronisation of the Jacobi iterations in heterogeneous environments, the problem of efficiently partitioning the computational work becomes equally important. Our algorithm has another advantage in this respect. We can use a simple static partitioning scheme during the preconditioning iteration and choose to repartition (if necessary) each outer iteration step. Any load imbalance that may have occurred during the preconditioning will then automatically be resolved.

The complete algorithm allows for highly recursive iteration schemes. For example, it would be possible to solve a sub–block in parallel on a dedicated cluster.

Another issue is tuning. There are many different parameters which have a significant effect on the performance of the complete iteration process, and finding the ideal parameters for a specific application may be a difficult issue.

There is an important question regarding trade–off. That is, finding the ideal time spend on preconditioning is highly problem dependent. Furthermore, it may be advantageous to vary the amount of preconditioning in each iteration step. In this case some form of convergence detection may become necessary. At the moment we use a fixed number of seconds for each preconditioning step. A related issue is how to accurately measure the effectiveness of the preconditioning step.

The preconditioning operator varies in each outer iteration step which forces us to use a flexible method. In some cases this can introduce additional overhead in the outer iteration, such as with flexible Conjugate Gradients [25, 26].

3 Numerical simulations

We have conducted several experiments solving the following form of the 3D convection–diffusion problem,

$$\begin{cases} -\nabla^2 \mathbf{u} + (2\mathbf{p} \cdot \nabla)\mathbf{u} = f(\mathbf{u}), & \mathbf{u} \in \mathbf{\Omega}, \\ \mathbf{u} = 0, & \mathbf{u} \in \partial\mathbf{\Omega}, \end{cases} \tag{3}$$

where $\mathbf{p} = (1, 2, 3)$. Discretisation by the finite difference scheme with a seven point stencil on a uniform $n_x \times n_y \times n_z$ mesh results in a sparse linear system of equations $Ax = b$ where A is of order $n = n_x n_y n_z$. Centered differences are used for the first derivatives. The grid points are numbered using the standard (lexicographic) ordering, resulting in a block pentadiagonal coefficient matrix.

3.1 Experimental setup

The experiments are performed using a local cluster, which is a multi–user system. It is moderately heterogeneous in design, consisting of twelve nodes: six Intel 2.20 GHz machines, two Intel 2.66 GHz machines, and four AMD Athlon 2.20 GHz machines. It will be used for heterogeneous experiments in a real–world setting. The nodes are equipped with memory in the range of 2–4 GB and the cluster is interconnected through 100 MB/s Ethernet links.

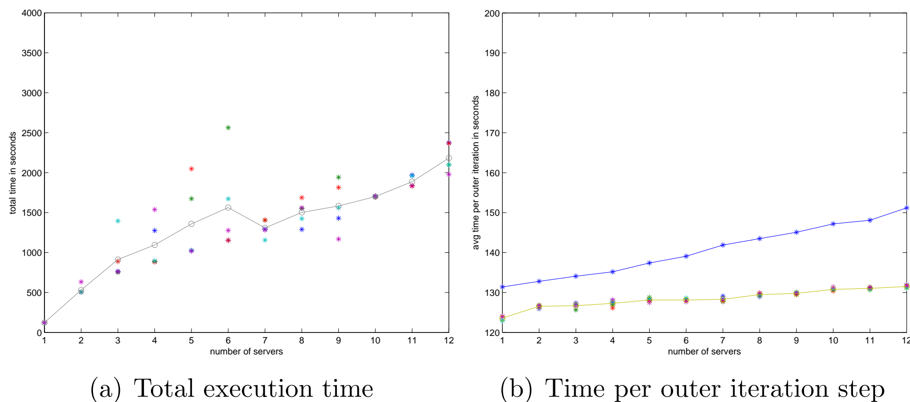
The IBP depot is started on one of the nodes in the cluster. Also, instead of letting the GridSolve agent assign the tasks, the client randomly allocates tasks to servers. The Jacobi sweeps are performed for a fixed number of seconds $T_{\max} = 120$s and we use matrix–free storage. The inner iterations are solved (inaccurately) with relative tolerance $\epsilon_{\text{in}} = 10^{-4}$ using the recent Krylov method IDR(s) taking $s = 4$ and preconditioned with ILU [27].

In the context of Grid computing, it is 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. For each experiment, we take $n_x = n_y = n_z$ such that the number of equations of unknowns per server is approximately 250,000.

Using the approach discussed previously, the outer iteration is performed either sequentially on the client machine or in parallel. The complete linear system is solved with relative residual $\epsilon = 10^{-8}$. To limit memory requirements, the truncation parameter is kept small ($m = 5$).

3.2 Experimental heterogeneous results

The experiments are performed on a typical work day, during which other users perform their computations on various combinations of nodes. The diversity of the hardware and the varying workload of the servers both add to the heterogeneity of the computational environment. The goal of these experiments is to provide results from a real–world setting.



(a) Total execution time

(b) Time per outer iteration step

Figure 2: Large heterogeneous cluster with 250,000 equations per server.

Table 1: Results of heterogeneous experiments.

number of servers	number of outer iterations	avg. comm/comp ratio server
1	1	n/a
2	$4, 5$	$6.3 \cdot 10^{-3}$
3	$6, 7, 11$	$2.3 \cdot 10^{-2}$
4	$7, 10, 12$	$3.2 \cdot 10^{-2}$
5	$8, 13, 16$	$4.8 \cdot 10^{-2}$
6	$9, 10, 13, 20$	$7.6 \cdot 10^{-2}$
7	$9, 10, 11$	$1.9 \cdot 10^{-1}$
8	$10, 11, 12, 13$	$1.6 \cdot 10^{-1}$
9	$9, 11, 12, 14, 15$	$2.0 \cdot 10^{-1}$
10	13	$3.1 \cdot 10^{-1}$
11	$14, 15$	$3.9 \cdot 10^{-1}$
12	$15, 16, 18$	$6.0 \cdot 10^{-1}$

Throughout the day, five executions of the algorithm were performed, each time using a different set of servers. Figure 2 shows results obtained using up to twelve servers (i.e., for problem sizes between 250,000 and three million). In Fig. 2(a) the total execution time of the outer iteration process is shown. The average execution times for each number of servers are connected with straight lines. Not surprisingly, the method converges in one outer iteration step when using a single server.

Table 1 shows the number of outer iterations with increasing servers. It also lists the average communication/computation ratio per server, for the complete outer iteration process. Although both the parallel and sequential implementation of the outer loop were used in the experiments, we only present results for the sequential outer iteration. The reason is that the additional overhead caused by the outer iteration is small compared to the total execution time.

For a small number of servers the spread in execution time is rather large, which may be explained as follows. Firstly, the heterogeneity of the computational environment can have an increased impact and secondly, the total number of outer iterations is relatively small. This may potentially cause large fluctuations in the number of outer iterations. Vice versa, for a large number of servers the effects of these properties are averaged.

The results show that the execution time increases, but this can be attributed almost completely to the increase in the number of outer iterations. Furthermore, increasing the problem size by adding servers has the following adverse consequences.

1. The coefficient matrix becomes increasingly ill–conditioned; and

2. the number of subdomains in asynchronous block Jacobi increases.

These two effects have a negative impact on the number of outer iterations. The first consequence is inherent to the problem and the second effect applies to all block Jacobi–type preconditioners. A possible third consequence is that the average number of Jacobi sweeps per server decreases due to increased communication. However, this was not observed in our experiments.

Factors that do have a large impact on the effectiveness of the preconditioner are the heterogeneity of the hardware and the differences in work load. Using the current computational environment and the aforementioned parameters the number of Jacobi sweeps on a server during a preconditioning step ranged between approximately 120 for a fully dedicated server and 30 for a fully occupied server.

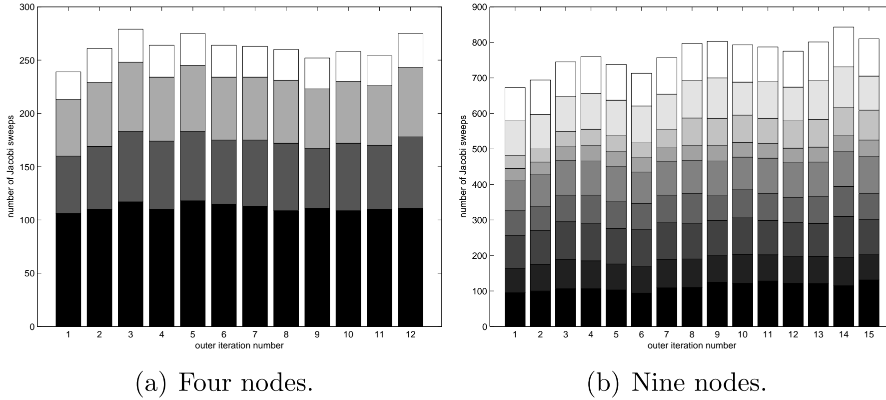
Despite these highly unfavorable conditions there is only a limited increase in total computing time.

Keeping the problem size per server fixed implies that — in the ideal case where overhead is negligible — the execution time per outer iteration remains constant. This is demonstrated in Fig. 2(b), were we show the average times per outer iteration step, for the sequential (bottom) and parallel (top) implementations. It indicates that for the sequential outer loop the overhead is rather small. Also, the increase in overhead due to the additional work in the outer loop and the (GridSolve) communication overhead is quite limited. For the parallel outer iteration a single execution is listed. Clearly, the overhead has increased and grows more rapidly compared to the sequential outer iteration.

For illustrative purposes, Fig. 3(a) and Fig. 3(b) demonstrate the effect of varying workload and heterogeneity of hardware on a typical execution of the algorithm. It shows the number of Jacobi iterations performed by each server per outer iteration step, using four and nine servers respectively. Certain nodes exhibit an increased workload and its effect is clearly visible.

4 Conclusions and future work

Although the idea of using a coarse–grain asynchronous iterative method as a preconditioner is quite natural, there have not been any implementations and experimental results



(a) Four nodes.

(b) Nine nodes.

Figure 3: Jacobi sweeps performed by each server during outer iteration steps.

in Grid environments — to the best of our knowledge. Previous approaches that solely use asynchronous iterative methods to solve linear systems on heterogeneous networks of computers appear limited in scope and applicability.

Our contribution is two–fold. We have described in detail an iterative algorithm that is designed to combine the strengths of both Grid computing and cluster computing to solve large linear systems. We believe it has the potential to be highly effective in performing large–scale numerical simulations in various fields of science. Furthermore, we have presented a fully working implementation using standardised Grid middleware, applied to a realistic test problem.

Valuable numerical experiments were performed under real–world conditions and we believe that the obtained results are promising in the context of sparse iterative solvers and Grid computing.

Naturally, there is much room for improvement. Increasing the effectiveness of the preconditioner may be done in various ways. Block Jacobi methods are closely related to domain decomposition techniques and the large number of subdomains would suggest that some form of coarse grid correction may be appropriate. Furthermore, communication is currently done through a single memory depot. We hope to reduce communication overhead by using Grid middleware that allows for direct communication between the servers.

Also, the work performed by the preconditioner is currently being divided equally over the available servers. In an extremely heterogeneous computing environment, such an approach could lead to significant desynchronisation of the block Jacobi processes and deteriorate the effectiveness of the preconditioner. We plan to use a form of resource–aware partitioning technique such as described in [22] to divide the work according to the currently available hardware.

Another important research question is how much time one should devote to the preconditioning. Further analysis is planned.

Acknowledgments

The authors would like to thank the GridSolve team for their prompt response pertaining to our questions and greatly appreciate the constructive criticism of the anonymous referees. This work was supported by the Delft Centre for Computational Science and Engineering within the framework of the project entitled *Development of an Immersed Boundary Method, Implemented on Cluster and Grid Computers, Application to the Swimming of Fish*.

References

[1] Foster, I., Kesselman, C.: The Grid: Blueprint for a new Computing Infrastructure. second edn. Morgan Kaufman Publishers (2004)

[2] Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public–resource computing. Commun. ACM **45**(11) (2002) 56–61

[3] van der Vorst, H., Vuik, C.: GMRESR: a family of nested GMRES methods. Num. Lin. Alg. Appl. **1**(4) (1994) 369–386

[4] Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentic Hall, Englewood Cliffs NJ (1989)

[5] Contassot-Vivier, S., Couturier, R., Denis, C., Jézéquel, F.: GREMLINS: GRid Efficient Methods for LINears Systems. Technical report (January 2007)

[6] Dongarra, J., Li, Y., Shi, Z., Fike, D., Seymour, K., YarKhan, A.: Homepage of NetSolve/GridSolve (2007)

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

[8] Bahi, J.M., Contassot-Vivier, S., Couturier, R.: Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. Parallel Comput. **31**(5) (2005) 439–461

[9] Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)

[10] Seymour, K., Nakada, H., Matsuoka, S., Dongarra, J., Lee, C., Casanova, H.: Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In: GRID '02: Proceedings of the Third International Workshop on Grid Computing, London, UK, Springer–Verlag (2002) 274–278

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

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

[13] Seymour, K., YarKhan, A., Agrawal, S., Dongarra, J.: NetSolve: Grid enabling scientific computing environments. In Grandinetti, L., ed.: Grid Computing and New Frontiers of High Performance Processing. Elsevier (2005)

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

[15] Sato, M., Boku, T., Takahashi, D.: OmniRPC: a Grid RPC system for parallel programming in cluster and Grid environment. In: CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society (2003) 206–213

[16] Caron, E., Del-Fabbro, B., Desprez, F., Jeannot, E., Nicod, J.M.: Managing data persistence in network enabled servers. Sci. Program. **13**(4) (2005) 333–354

[17] Brady, T., Konstantinov, E., Lastovetsky, A.: SmartNetSolve: High level programming system for high performance Grid computing, Rhodes Island, Greece, IEEE Computer Society (25-29 April 2006 2006) CD-ROM/Abstracts Proceedings.

[18] Lastovetsky, A., Zuo, X., Zhao, P.: A non–intrusive and incremental approach to enabling direct communications in RPC–based grid programming systems. Technical report (2006)

[19] Zuo, X., Lastovetsky, A.: Experiments with a software component enabling NetSolve with direct communications in a non–intrusive and incremental way. In: Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, California, USA, IEEE Computer Society (26-30 March 2007 2007)

[20] Desprez, F., Jeannot, E.: Improving the GridRPC model with data persistence and redistribution. In: ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC/HeteroPar'04), Washington, DC, USA, IEEE Computer Society (2004) 193–200

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

[22] Collignon, T.P., van Gijzen, M.B.: Implementing the Conjugate Gradient Method on a grid computer. In: Proceedings of the International Multiconference on Computer Science and Information Technology, Volume 2, October 15–17, 2007, Wisla, Poland. (2007) 527–540

[23] Daniel, J., Gragg, W.B., Kaufman, L., Stewart, G.W.: Reorthogonalization and stable algorithms for updating the Gram–Schmidt QR factorization. Mathematics of Computation **30** (1976) 772–795

[24] Björck, Å.: Solving linear least squares problems by Gram–Schmidt orthogonalization. BIT **7** (1967) 1–21

[25] Notay, Y.: Flexible conjugate gradients. SIAM Journal on Scientic Computing **22** (2000) 1444–1460

[26] Simoncini, V., Szyld, D.B.: Flexible inner–outer Krylov subspace methods. SIAM J. Numer. Anal. **40**(6) (2002) 2219–2239

[27] Sonneveld, P., van Gijzen, M.B.: IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems. Technical report, Delft University of Technology, Delft, the Netherlands (2007) DUT Report 07–07.