



Implementing the Conjugate Gradient Method on a grid computer

Tijmen Collignon and Martin B. van Gijzen

Delft University of Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science,
Mekelweg 4, 2628 CD Delft, the Netherlands,
{t.p.collignon,m.b.vangijzen}@tudelft.nl,
Website: <http://ta.twi.tudelft.nl/nw/users/collignon/>

Abstract. We study two implementations of the Conjugate Gradient method for solving large sparse linear systems of equations on a heterogeneous computing grid, using GridSolve as grid middleware. 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. The computational work is divided into tasks which are dynamically distributed over the available resources using a resource-aware data partitioning strategy. We present numerical experiments that show lower computing times and better speed-up for the Chronopoulos/Gear variant. We also identify bottlenecks and suggest improvements to GridSolve.

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 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 the available non-dedicated hardware, and thus to make better use of the existing resources. This idea has given rise to the concepts of grid computing and of computational grids, see for example [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 at geographically different locations 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 nodes and the variability in network performance offer new algorithmic problems.

In this paper we study different implementations of the Conjugate Gradient (CG) method [3] on a computational grid. We use the GridSolve library [4] as grid middleware. 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 that has been proposed by Chronopoulos/Gear [5]. Our numerical experiments show that by minimising the number of synchronisation points and by careful load balancing a significant speed-up can also 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 and related issues, which includes a description of GridSolve, data management strategies, and specific implementation details. Section 3 contains some experimental results and in Sect. 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 [6] to simulate general fluid-structure interaction problems using grid computers. Examples of said problems are the swimming of fish or the airflow around wind turbine rotor blades. These simulations involve numerically solving 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. Such a system typically emerges from a finite difference discretisation of the Poisson equation with varying coefficients and Neumann boundary conditions,

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

Here, p and ρ represent the pressure and density respectively. For the purpose of this paper we confine ourselves to a two-dimensional square domain where p is zero on the boundary. Nevertheless, this problem still retains much of the characteristics of the original problem. The equation is discretised on a $k \times k$ grid using second order finite differences, resulting in the discrete Poisson equation,

$$\begin{aligned} \rho_{i+\frac{1}{2},j} p_{i+1,j} + \rho_{i,j+\frac{1}{2}} p_{i,j+1} - \left(\rho_{i+\frac{1}{2},j} + \rho_{i,j+\frac{1}{2}} + \rho_{i-\frac{1}{2},j} + \rho_{i,j-\frac{1}{2}} \right) p_{ij} + \\ \rho_{i-\frac{1}{2},j} p_{i-1,j} + \rho_{i,j-\frac{1}{2}} p_{i,j-1} = h^2 f_{ij}, \quad \text{for } 0 \leq i, j < k. \end{aligned} \quad (2)$$

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 (0.16.0) 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.

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, 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.

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

$$\frac{s_{\text{flops}} \times s_{\text{ncpu}}}{\frac{s_{\text{workload}}}{100} + 1.0}, \quad (8)$$

where s_{flops} is the speed of server s in flops determined by 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.

The main advantages of GridSolve are that it is easy to use, install, and maintain. It allows for easy 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, the 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 done before iterative solvers can be efficiently applied in Grid computing.

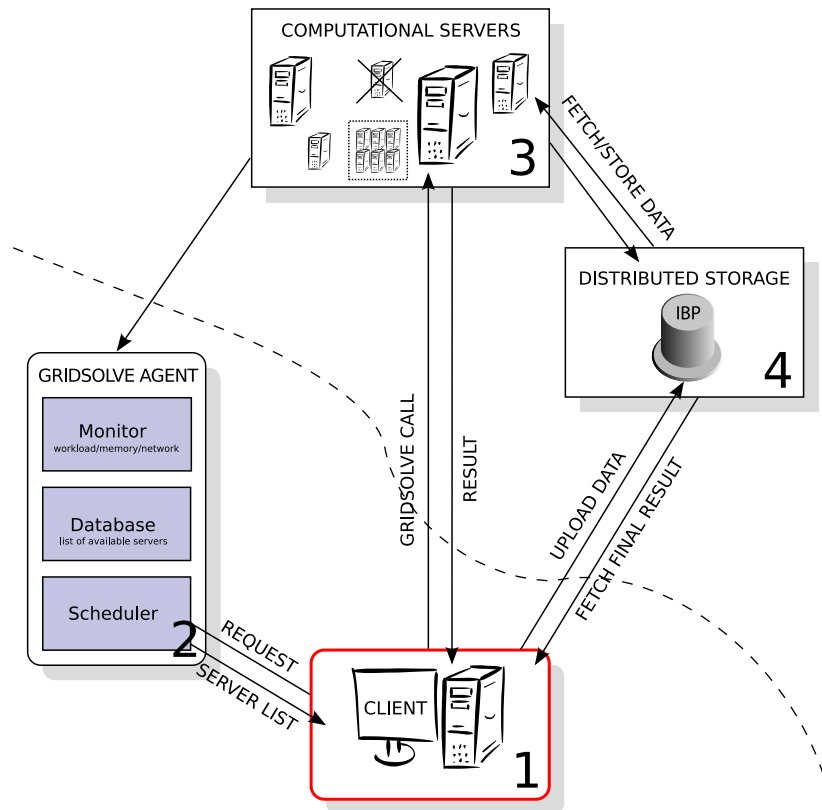


Fig. 1. Schematic overview of GridSolve. The dashed line symbolizes distance between client and servers.

2.3 Data management

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 is continuously being sent back and forth between the client and the server using a possible slow network connection. Also, any data that is read or generated locally during the execution of a task is 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 [14–18].

In GridSolve there is a partial solution to the first problem called the Distributed Storage Infrastructure (DSI). At the Logistical Computing and Inter-networking (LoCI) Laboratory of the University of Tennessee the IBP (Internet Backplane Protocol) middleware has been developed based on this approach [19]. 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 (see component (4) in Fig. 1). Using the DSI can be considered as programming for a shared memory model.

An approach similar to [15] in which the RPC model of NetSolve is extended to include communication between remote servers is currently being developed for GridSolve [20]. In the future we hope to use this extension and apply it to our problem.

2.4 Resource-aware load balancing

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

- 1: Agent partitions work based on available computational resources.
 - 2: Client sets initial values and uploads initial vectors such as b to the IBP data depot;
 Set $k = 0$.
 - 3: **while** CG not converged and $k < k_{\max}$ **do**
 - 4: Client assigns CG tasks to p servers and waits until tasks have completed.
 - 5: Client repartitions work if significant change in workload and/or computational resources has occurred.
 - 6: Set $k = k + 1$
 - 7: **end while**
 - 8: Client reads final answer from IBP depot.
-

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 being done in GridSolve. Instead, we need to use a slightly different approach. Suppose that the client wishes to use p servers to solve a linear system. The scheduler in the GridSolve agent has been enhanced so that it creates a simple (non-homogeneous) partitioning of the computational work over p servers using information about currently available resources. It then returns the partitioning and a list of 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 timestep of the iteration the client may decide to repartition the work and assign the work to different computational servers in the network accordingly.

2.5 Partitioning algorithm and CG schemes

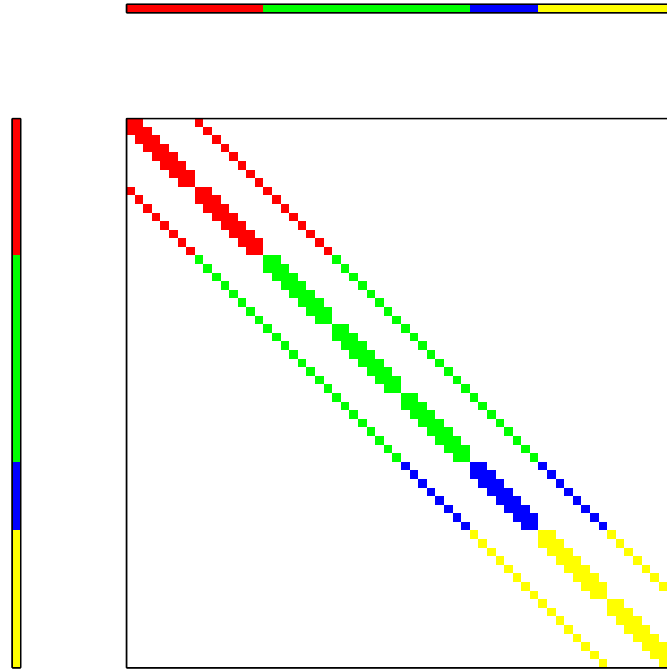


Fig. 2. Heterogeneous block-row partitioning $[2, 3, 1, 2]$ for $p = 4$ and $k = 8$.

As was shown earlier, the matrix A originating from our discretisation of the system (1) is a block tridiagonal matrix. Since each block-row roughly contains the same number of non-zeros, we chose a simple non-homogeneous block-row partitioning. An example of such a partitioning is illustrated in Fig.2 for $p = 4$ and $k = 8$. The input and output vectors are distributed in the same manner. More specifically, the effective speed of p servers is calculated using (8), together with the total flop count of a single CG iteration step. The size of each task is then determined accordingly.

The standard Conjugate Gradient method was implemented first and is shown in Algorithm 2. Jacobi preconditioning is used and there are two natural synchronisation barriers, namely the two inner products for computing α and ρ . When implementing this scheme in GridSolve, an extra synchronisation point is introduced due to technical implementation details. By simply rearranging certain terms this can be avoided.

Algorithm 2 Standard CG with Jacobi preconditioning; Task i with three sub-tasks.

Require: File handles to vectors $\mathbf{x}, \mathbf{r}, \mathbf{z}, \mathbf{p}, \mathbf{q}$ on IBP data depot and parameter k .

Ensure: $K := \text{diag}(A)$

```

1: // Each server  $i$  does the following.
2: Read  $\mathbf{r}^i$ 
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  $\mathbf{z}^i$ 
6: -SYNCHRONIZE- (Client sums  $\rho^i$ )
7: Read  $\mathbf{z}^i$  and  $\mathbf{p}^i$ 
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  $\mathbf{q}^i$ 
17: -SYNCHRONIZE- (Client sums  $\alpha^i$ )
18: Read  $\mathbf{x}^i, \mathbf{r}^i, \mathbf{p}^i$ , and  $\mathbf{q}^i$ 
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  $\mathbf{r}^i$ 
22: Check convergence; continue if necessary
23: Clients sets  $\rho_{\text{old}} = \rho$ 

```

To increase the granularity we have also implemented the Chronopoulos/Gear variant of CG [5, 21], which has a single synchronisation point, see Algorithm 3. Furthermore, this scheme introduces an additional $2n$ flops in each iteration step compared to the original scheme. Note that we employ matrix-free storage; each matrix element is recomputed when it is needed. Generating the matrix resulting from discretising the Poisson equation with varying diffusivity requires a significant amount of computation, increasing the granularity even further.

2.6 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 [22]. Although read operations by differ-

Algorithm 3 CG method with Jacobi preconditioning; 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 α and β .

Ensure: $K := \text{diag}(A)$ and initial values: Solve \mathbf{w} from $K\mathbf{w} = \mathbf{r}$; $\mathbf{s} := A\mathbf{v}$; $\rho := (\mathbf{r}, \mathbf{w})$; $\mu := (\mathbf{s}, \mathbf{w})$; $\alpha := \rho/\mu$.

- 1: // Each server i performs the following
 - 2: Read \mathbf{x}^i and \mathbf{p}^i .
 - 3: Depending on bandwidth of matrix read appropriate portions of vectors \mathbf{q} , \mathbf{r} , \mathbf{w} , and \mathbf{s} .
 - 4: Set $\mathbf{p}^i = \mathbf{w}^i + \beta\mathbf{p}^i$
 - 5: Set $\mathbf{q}^i = \mathbf{s}^i + \beta\mathbf{q}^i$
 - 6: Set $\mathbf{x}^i = \mathbf{x}^i + \alpha\mathbf{p}^i$
 - 7: Set $\mathbf{r}^i = \mathbf{r}^i - \alpha\mathbf{q}^i$
 - 8: Check convergence; continue if necessary
 - 9: Solve \mathbf{w}^i from $K\mathbf{w}^i = \mathbf{r}^i$
 - 10: Compute $\mathbf{s}^i = A\mathbf{w}^i$
 - 11: Compute $\rho^i = (\mathbf{r}^i, \mathbf{w}^i)$
 - 12: Compute $\mu^i = (\mathbf{s}^i, \mathbf{w}^i)$
 - 13: Update \mathbf{x}^i , \mathbf{r}^i , \mathbf{w}^i , \mathbf{p}^i , \mathbf{q}^i , and \mathbf{s}^i on the depot
 - 14: Return ρ^i and μ^i to client
 - 15: -SYNCHRONIZE-
 - 16: Client sums ρ^i and μ^i for all i
 - 17: Client sets $\beta = \rho/\rho_{\text{old}}$
 - 18: Client computes $\alpha = \rho/(\mu - \rho\beta/\alpha)$
 - 19: Client sets $\rho_{\text{old}} = \rho$
-

ent 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 hope to overcome this imbalance by using separate DSI files for each vector and letting each task update the vectors in a 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 is inadvertently overwritten. Specifically, we cannot guarantee that every task has finished reading the data from the previous iteration before other tasks have 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 [23] as a BLAS implementation which is used for the various `daxpy` and inner product operations. Each task recomputes its portion of the sparse matrix A every timestep and stores it using compressed row storage (CRS) format.

Although detection of convergence is an important issue in iterative methods, this is currently not implemented.

3 Experiments

In the previous sections we discussed several implementations and various suggestions for increasing the granularity. In this section we will perform several experiments and investigate the effect of these suggestions on the performance.

Our testbed is a local network of computers, which is a multi-user system consisting of different processors with 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 2.6.18. In order to perform controlled and repeatable experiments we decided on using idle processors and as a result the partitioning is fixed and homogeneous throughout the experiments. We measure the wall clock times of five CG steps for different values of n .

In the ideal case the IBP depot would be located on a dedicated node connected through a high-speed network to the computational nodes. Unfortunately such a resource was not available at the time of testing, and therefore we opted to start the depot on one of the dual core nodes. As a result we expect a huge communication overhead. The client and the agent are both running on a single core node while the servers are started on the remaining nodes. In a typical Grid environment the client program would be located on the user's desktop machine.

We differentiate between four implementations:

- (a) Standard CG without varying diffusivity using a single DSI file;
- (b) Chronopoulos/Gear scheme without varying diffusivity using a single DSI file;
- (c) Chronopoulos/Gear CG with varying diffusivity using a single DSI file; and
- (d) Chronopoulos/Gear CG with varying diffusivity using separate DSI files which are updated in a random order.

Figure 3(a) shows the total wall clock time of the first implementation for different values of n using up to seven servers. It also demonstrates that communication overhead is particularly an issue 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 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 four different implementations for $n = 4 \cdot 10^6$. Here we clearly see the decrease in execution times when using the Chronopoulos/Gear variant for a large number of servers. It is interesting to note that with implementation (b) and using three to four servers we observe a large drop in execution time. We do not have an explanation for this behaviour.

Note that despite our attempts to improve the granularity of the computation, we do not observe any significant improvements.

Although we observed that using separate DSI files for the vectors only improved the overall running time of the Chronopoulos/Gear scheme when using a small 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 are shown after five CG steps of Chronopoulos/Gear, broken down in both 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, Fig. 4(a) reveals increasing wall clock times for each subsequent task, which can be explained as follows. Although the client initiates a sequence of non-blocking calls, at the end of the first task the updates to the DSI file appears to block subsequent updates by other tasks. These figures also clearly reveal the amount of communication overhead.

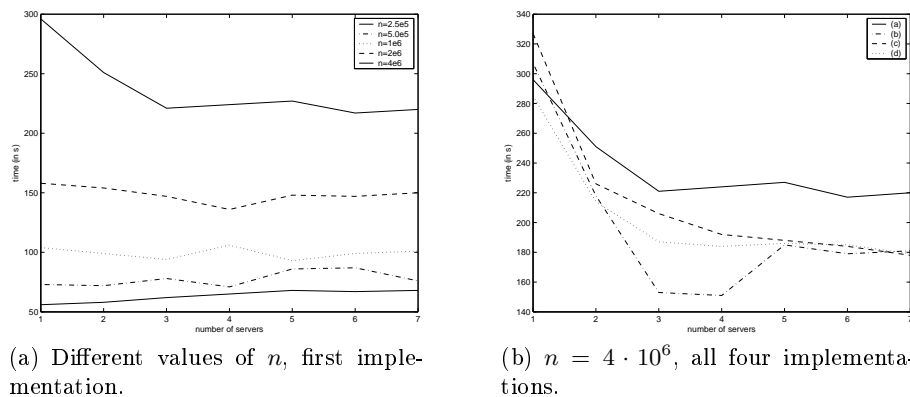


Fig. 3. Wall clock times of CG implementations in GridSolve.

4 Concluding remarks and future work

In this work we have described a case study where we have experimented with solving large sparse systems in parallel on Grid computers using a specific Grid middleware and an architecture-aware load balancing algorithm. A sparse iterative solver was implemented in GridSolve, which is mature grid middleware for accessing remote computational resources. Several suggestions for improving the granularity were given and implemented.

Our contribution is threefold. We have used a grid middleware to solve sparse linear systems with CG in a distributed manner. Using the middleware we have

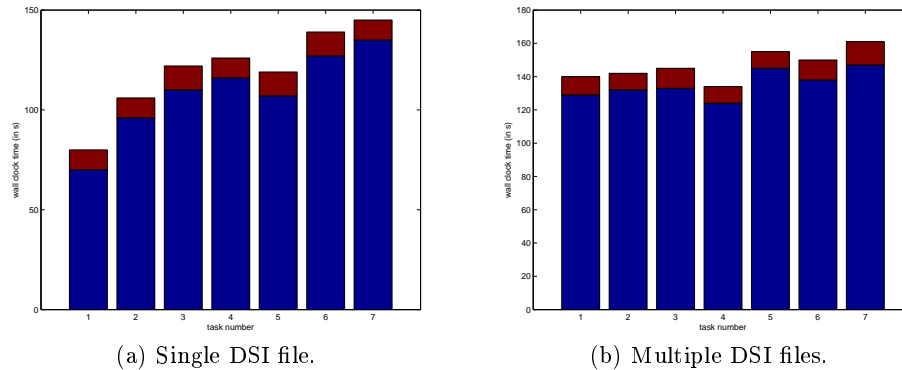


Fig. 4. Breakdown of wall clock time of tasks in communication (bottom part) and computation.

also implemented a simple but effective architecture-aware partitioning algorithm to divide the computational work. And finally, we have increased the granularity by using a custom version of CG that has a single synchronisation point.

Clearly our method is only suitable for solving very large systems. Furthermore, it is naturally suited for linear systems with specific classes of coefficient matrices, such as Poisson and Toeplitz matrices. This allows us to use matrix-free storage. Although we did not have a dedicated DSI depot at our disposal the results are promising and we expect that using such a dedicated resource will greatly improve the scalability of our approach.

There are many possible improvements and we will give some suggestions for future work. The current implementation of GridSolve forces us to use bridge communication. SmartGridSolve is an extension of GridSolve which is currently being developed. It will perform similarly to SmartNetSolve [15], allowing for communication between the computational servers as well as data persistence. By combining this with sophisticated (possibly weighted) hypergraph partitioning techniques such as used in Mondriaan [24] we hope to greatly improve our load balancing algorithm.

To further increase the granularity of the computations, we plan to use more sophisticated preconditioning techniques, such as block ILU and deflation.

Another possible improvement is incorporating network capabilities into the partitioning algorithm.

5 Acknowledgments

The authors would like to thank the GridSolve team for their prompt response pertaining to our questions. 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** (2002) 56–61.
3. Hestenes M. R., Stiefel E.: *Methods of conjugate gradients for solving linear systems*, Journal of Research of National Bureau Standards **49** (1952) 409–436.
4. Dongarra J., Li Y., Shi Z., Fike D., Seymour K., YarKhan A.: *Homepage of NetSolve/GridSolve* (2007). <http://icl.cs.utk.edu/netsolve/>, latest version of GridSolve is v0.16.0 (as of 2007-06-03).
5. Chronopoulos A. T., Gear C. W.: *S-step iterative methods for symmetric linear systems*, J. Comput. Appl. Math. **25** (1989) 153–168.
6. Mittal R., Iaccarino G.: *Immersed Boundary Methods*, Annual Review of Fluid Mechanics **37** (2005) 239–261.
7. YarKhan A., Seymour K., Sagi K., Shi Z., Dongarra J.: *Recent Developments in GridSolve*, International Journal of High Performance Computing Applications (IJHPCA) **20** (2006) 131–141.
8. 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.
9. Lee C., Nakada H., Tanimura Y.: *GridRPC Working Group* (2007) <https://forge.gridforum.org/projects/gridrpc-wg/>.
10. Caron E., Desprez F.: *DIET: A scalable toolbox to build network enabled servers on the Grid*, International Journal of High Performance Computing Applications **20** (2006) 335–352.
11. 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).
12. 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** (2003) 41–51.
13. Sato M., Boku T., Takahashi D.: *OmnirRPC: a Grid RPC system for parallel programming in cluster and Grid environment*, In: CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid, Washington, DC, USA, IEEE Computer Society (2003) 206–213.
14. Caron E., Del-Fabbro B., Desprez F., Jeannot E., Nicod J. M.: *Managing data persistence in network enabled servers*, Sci. Program. **13** (2005) 333–354.
15. Brady T., Konstantinov E., Lastovetsky A.: *SmartNetSolve: High level programming system for high performance Grid computing*, Rhodes Island, Greece, IEEE Computer Society (2006) CD-ROM/Abstracts Proceedings.
16. 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).

17. 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 (2007).
18. Desprez F., Jeannot E.: *Improving the GridRPC model with data persistence and redistribution*, In: ISPD'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 (ISPD/HeteroPar'04), Washington, DC, USA, IEEE Computer Society (2004) 193–200.
19. 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** (2002) 1773–1787.
20. Seymour K.: *Private correspondence* (2007).
21. Dongarra J. J., Duff I. S., Sorensen D. C., van der Vorst H. A.: *Numerical Linear Algebra for High Performance Computers*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1998).
22. Zheng Y., Bassi A., Beck M., Plank J. S., Wolski R.: *Internet Backplane Protocol: C API 1.4*, Technical report (2004).
<http://loci.cs.utk.edu/ibp/documents/IBPClientAPI.pdf>
23. Whaley R. C., Petitet A.: *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, *Software: Practice and Experience* **35** (2005) 101–121. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>
24. Vastenhouw B., Bisseling R. H.: *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, *SIAM Rev.* **47** (2005) 67–95.