DELFT UNIVERSITY OF TECHNOLOGY

REPORT 08-12

Parallel Scientific Computing on Loosely Coupled Networks of Computers

Tijmen Collignon and Martin B. van Gijzen

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2008

Parallel Scientific Computing on Loosely Coupled Networks of Computers

Tijmen P. Collignon and Martin B. van Gijzen

Abstract Efficiently solving large sparse linear systems on loosely coupled networks of computers is a rich and vibrant field of research. The induced heterogeneity and volatile nature of the aggregated computational resources present numerous algorithmic challenges. Designing efficient numerical algorithms for said systems is a complex process that brings together many different scientific disciplines. This book chapter is divided into two distinct parts. The purpose of the first half (Sect. §2–§4) is to give a bird's view of the issues pertaining to designing efficient numerical algorithms for *Grid computing*. It kicks off by clearly stating the problem and exposing the various bottlenecks, subsequently followed by the presentation of potential solutions. Thus, the stage is set and Sect. §3 proceeds by detailing *classical* iterative solution methods, along with the concept of *asynchronism*, which is a highly favorable quality in the context of Grid computing. The first half is wrapped up by explaining how asynchronism can be introduced into faster but more complicated *subspace methods*. The general idea is that by using an asynchronous method as a *preconditioner*, the best of both worlds can be combined. The advantages and disadvantages of this approach are discussed in minute detail. The second half (Sect. §5) contains discussions on the various intricacies related to *implementing* the proposed algorithm on Grid computers. Section §6 gives some concluding remarks along with suggestions for further reading.

T. P. Collignon
Delft University of Technology, Delft Institute of Applied Mathematics and J. M. Burgerscentrum, Mekelweg 4, 2628 CD Delft, the Netherlands, e-mail: t.p.collignon@tudelft.nl

M. B. van Gijzen
Delft University of Technology, Delft Institute of Applied Mathematics and J. M. Burgerscentrum, Mekelweg 4, 2628 CD Delft, the Netherlands, e-mail: m.b.vangijzen@tudelft.nl

1 Introduction

Solving extremely large sparse linear systems of equations is the computational bottleneck in a wide range of scientific applications. Examples include airflow around wind turbine rotor blades, weather prediction, options pricing, and search engines. Although the computing power of a single processor continues to grow, fundamental physical laws place severe limitations on sequential processing. This fact accompanied by an ever increasing demand for more realistic simulations has intensely stimulated research in the field of *parallel and distributed computing*. By combining the power of multiple processors and sophisticated numerical algorithms, simulations can be performed that perfectly imitate physical reality.

Traditional parallel processing was and is currently performed using sophisticated supercomputers, which typically consist of thousands of identical processors linked by a high–speed network. They are often purpose–built and highly expensive to operate, maintain, and expand.

A poor man's alternative to massive supercomputing is to exploit existing nondedicated hardware for performing parallel computations. With the use of cost–effective commodity components and freely available software, cheap and powerful parallel computers can be built. The *Beowulf* cluster technology is a good example of this approach [49]. A major advantage of such technology is that resources can easily be replaced and added. However, this introduces the problem of dealing with *heterogeneity*, both in machine architecture and in network capabilities. The problem of efficiently *partitioning* the computational work became an intense topic of research.

The nineties of the previous century ushered in the next stage of parallel computing. With the advent of the Internet, it became viable to connect geographically separate resources — such as individual desktop machines, local clusters, and storage space — to solve very large–scale computational problems. In the mid–1990s the SETI@home project was conceived, which has established itself as the prime example of a so–called *Grid computing* project. It currently combines the computational power of millions of personal computers from around the world to search for extraterrestrial intelligence by analysing massive quantities of radio telescope data [1].

In analogy to the Electric Grid, the driving philosophy behind Grid computing is to allow individual users and large organisations alike to access *casu quo* supply computational resources without effort by plugging into the Computational Grid. Much research has been done in Grid software and Grid hardware technologies, both by the scientific community and industry [29].

The fact that in Grid computing resources are geographically separated implies that *communication* is less efficient compared to dedicated parallel hardware. As a result, it is naturally suited for so–called *embarrassingly parallel* applications where the problem can be broken up easily and tasks require little or no interprocessor communication. An example of such an application is the aforementioned SETI@home project.

For the numerical solution of linear systems of equations, matters are far more complicated. One of the main reasons is that inter–task communication is both un-

avoidable and abundant. 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 resources and the variability in network performance present numerous algorithmic challenges. This book chapter highlights the key difficulties in designing such algorithms and strives to present efficient solutions.

One of the latest trends in parallel processing is *Cell* or *GPU computing*. Modern gaming consoles and graphics cards employ dedicated high–performance processors for specialised tasks, such as rendering high–resolution graphics. In combination with their inherent parallel design and cheap manufacturing process, this makes them extremely appropriate for parallel numerical linear algebra [60]. The Folding@Home project is a striking example of an embarrassingly parallel application where the power of many gaming consoles is used to simulate protein folding and other molecular dynamics [28].

Nowadays, multi–core desktop computers with up to four cores are becoming increasingly mainstream. An obvious application is the field of parallel scientific computing. Furthermore, many existing user software such as graphics editors and computer games cannot benefit from these additional resources effectively. Such software often needs to be rewritten from scratch and this has also become an intensive topic of research.

The book chapter is divided into two distinct parts. The purpose of the first half (Sect. §2–§4) is to give a bird's view of the issues pertaining to designing efficient numerical algorithms for Grid computing and is aimed at a general audience. The second half (Sect. §5) deals with more advanced topics and contains detailed discussions on the issues related to implementing said algorithm on Grid computers. Section §6 gives some concluding remarks along with suggestions for further reading.

2 The problem

Large systems of linear equations arise in many different scientific disciplines, such as physics, computer science, chemistry, biology, and economics. Their efficient solution is a rich and vibrant field of research with a steady supply of important results. As the demand for more realistic simulations continues to grow, the use of direct methods for the solution of linear systems becomes increasingly infeasible. This leaves iterative methods as the only practical alternative.

The main characteristic of such methods is that at each iteration step, information from one or more previous iteration steps is used to find an increasingly accurate approximation to the solution. Although the design of new iterative algorithms is a very active field of research, physical restrictions such as memory capacity and computational power will always place limits on the type of problem that can be solved on a single processor.

Table 1 Parallel and distributed computing on cluster and Grid hardware.

Cluster computing	Grid computing
local–area–networks	wide–area networks
dedicated	non–dedicated
special–purpose hardware	aggregated resources
fast network	slow connections
synchronous communication	asynchronous communication
fine–grain	coarse–grain
homogeneous	heterogeneous
reliable resources	volatile resources
static environment	dynamic environment

The obvious solution is to combine the power of multiple processors in order to solve larger problems. This automatically implies that memory is also distributed. Combined with the fact that iterations may be based on previous iterations, this suggests that some form of *synchronisation* between the processors has to be performed.

Accumulating resources in a local manner is typically called cluster computing. Neglecting important issues such as heterogeneity, this approach ultimately has the same limitations as with sequential processing: memory capacity and computational power. The next logical step is to combine computational resources that are geographically separated, possible spanning entire continents. This idea gives birth to the concept of Grid computing. Ultimately, the price that needs to paid is that of synchronisation.

Table 1 lists some of the classifications that may be associated with cluster and Grid computing, respectively. In real life, things are not as clear–cut as the Table might suggest. For example, a cluster of homogeneous and dedicated clusters connected by a network is considered a Grid computer. Vice versa, a local cluster may consist of computers that have varying workload, making the annotations 'dedicated' and 'static environment' unwarranted.

The high cost of global synchronisation is not the only algorithmic hurdle in designing efficient numerical algorithms for Grid computing. In Tab. 2 the main problems are listed, along with possible solutions. Clearly there are many aspects that need to addressed, requiring substantial expertise from a broad spectrum of mathematical disciplines.

When designing numerical algorithms for general applications, a proper balance should be struck between *robustness* (consistent performance with few parameters) and *efficiency* (optimal scalability, both algorithmic and parallel). At the risk of trivialising these two highly important issues, the ultimate numerical algorithm wishlist for Grid computing contains the following additional items: coarse–grain, asynchronous communications, minimal number of synchronisation points, resource–aware, dynamic, and fault tolerant. The ultimate challenge is to devise an algorithm that exhibits all of these eight features.

Table 2 Main difficulties and possible solutions associated with designing efficient numerical algorithms in Grid computing.

Difficulties and challenges	Possible solutions
– **Frequent synchronisation.** One of the reasons for synchronisation is global reduction. Compared to the overhead, the data that is being exchanged is relatively small, making this an extremely expensive operation in Grid environments. The most important example is the computation of an inner product.	– **Coarse–grained.** Communication is expensive, so the amount of computation should be large in comparison to the amount of communication. – **Asynchronous communication.** Tasks should not have to wait for specific information from other tasks to become available. That is, the algorithm should be able to incorporate any newly received information immediately. – **Minimising synchronisation points.** Many iterative algorithms can be modified in such a manner that the number of synchronisation points is reduced. These modifications include rearrangement of operations [15], truncation strategies [50], and the type of reorthogonalisation procedure [21].
– **Heterogeneity.** Resources from many different sources may be combined, potentially resulting in a highly heterogeneous environment. This can apply to machine architecture, network capabilities, and memory capacities.	– **Resource–aware.** When dividing the work, the diversity in computational hardware should be reflected in the partitioning process. Techniques from graph theory are extensively used here [52].
– **Volatility.** Large fluctuations can occur in things like processor workload, processor availability, and network bandwidth. A huge challenge is how to deal with failing network connections or computational resources.	– **Dynamic.** Changes in the computational environment should be detected and accounted for, either by repartitioning the work periodically or by using some type of diffusive partitioning algorithm [52]. – **Fault tolerant.** The algorithm should somehow be (partially) resistant to failing resources in the sense that the iteration process may stagnate in the worst case, but not break down.

3 The basics: iterative methods

The goal is to efficiently solve a large algebraic linear system of equations,

$$Ax = b, \tag{1}$$

on large heterogeneous networks of computers. Here, A denotes the coefficient matrix, b represents the right–hand side vector, and x is the vector of unknowns.

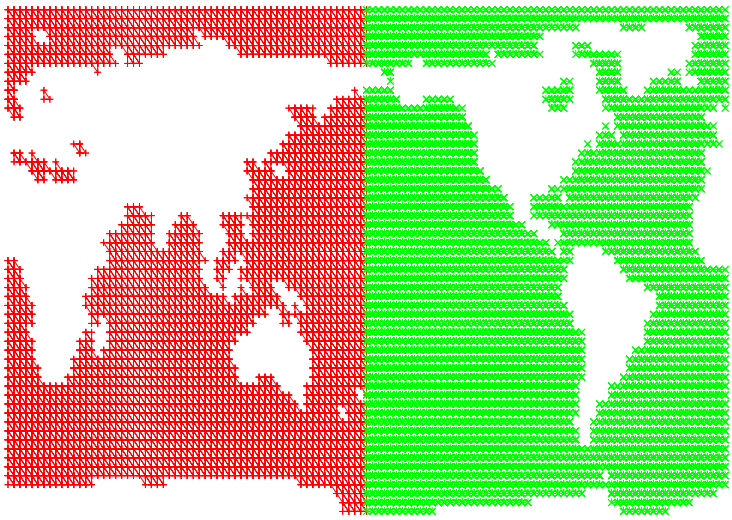

Fig. 1 Depiction of the oceans of the world, divided into two separate computational subdomains.

3.1 Simple iterations

Given an initial solution $x^{(0)}$, the classical iteration for solving the system (1) is

$$x^{(t+1)} = x^{(t)} + M^{-1}(b - Ax^{(t)}), \qquad t = 0, 1, \ldots, \tag{2}$$

where M^{-1} serves as an approximation for A^{-1}. For practical reasons, inverting the matrix M should be cheap and this is reflected in the different choices for M. The simplest option would be to choose the identity matrix for M, which results in the *Richardson* iteration. Another variant is the *Jacobi* iteration, which is obtained by taking for M the diagonal matrix having entries from the diagonal of A. Choices that in some sense better approximate the matrix A naturally result in methods that converge to the solution in less iterations. However, inverting the matrix M will be more expensive and it is clear that some form of trade–off is necessary.

The iteration (2) can be generalised to a block version, which results in an algorithm closely related to *domain decomposition* techniques [46]. One of the earliest variants of this method was introduced as early as 1870 by the German mathematician Hermann Schwarz. The general idea is as follows. Most problems can be divided quite naturally into several smaller problems. For example, problems with complicated geometry may be divided into subdomains with a geometry that can be handled more easily, such as rectangles or triangles.

Consider the physical domain Ω shown in Fig. 1. The objective is to solve some given equation on this domain. For illustrative purposes, the domain is divided into two subdomains Ω_1 and Ω_2. The matrix, the solution vector, and the right–hand side are partitioned into blocks as follows:

Algorithm 1 Block Jacobi iteration for solving $Ax = b$.

OUTPUT: Approximation of $Ax = b$;
1: Initialize $x^{(0)}$;
2: **for** $t = 0, 1, \ldots,$ until convergence **do**
3: **for** $i = 1, 2, \ldots, p$ **do**
4: Solve $A_{ii}x_i^{(t+1)} = b_i - \sum_{j=1, j\neq i}^{p} A_{ij}x_j^{(t)}$;
5: **end for**
6: **end for**

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{3}$$

The two matrices on the main diagonal of A symbolise the equation on the subdomains themselves, while the coupling between the subdomains is contained in the off–diagonal matrices A_{12} and A_{21}.

Block Jacobi generalises standard Jacobi by taking for M the block diagonal elements, giving

$$M = \begin{bmatrix} A_{11} & \varnothing \\ \varnothing & A_{22} \end{bmatrix}. \tag{4}$$

This results in the following two iterations for the first and second domain respectively,

$$\begin{cases} x_1^{(t+1)} = x_1^{(t)} + A_{11}^{-1}\left(b_1 - A_{11}x_1^{(t)} - A_{12}x_2^{(t)}\right); \\ x_2^{(t+1)} = x_2^{(t)} + A_{22}^{-1}\left(b_2 - A_{21}x_1^{(t)} - A_{22}x_2^{(t)}\right), \end{cases} \quad t = 0, 1, \ldots. \tag{5}$$

On a parallel computer, these iterations may be performed independently for each iteration step t. This is followed by a synchronisation point where information is exchanged between the processors. Algorithm 1 shows the general case for p processors and/or subdomains. An extra complication is that the block matrices located on the diagonal need to be inverted. In most cases these matrices have the same structure as the complete matrix. Therefore, systems involving these matrices are usually solved using some other iterative method, possibly block Jacobi. Another important issue is how accurately these systems should be solved.

3.2 Impatient processors: asynchronism

Parallel asynchronous algorithms can be considered as a generalisation of simple iterative methods such as the aforementioned block Jacobi method. Instead of exchanging the most recent information with other processes at each iteration step, an asynchronous algorithm performs their iterations based on information that is avail-

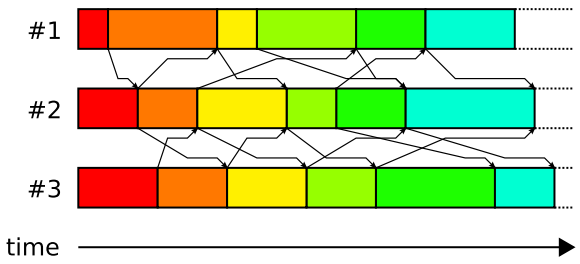

Fig. 2 Time line of a certain type of asynchronous algorithm, showing three (Jacobi) processes. Newly computed information is send at the end of each iteration step and newly received information is used only at the start of each iteration. The schematic is inspired by [3].

able at that particular time. Therefore, the iteration counter t loses it global meaning. The classification *asynchronous* pertains to the type of communication.

In Fig. 2 a schematic is given which illustrates some of the important features of a particular type of asynchronous algorithm. Time is progressing from left to right and communication between the three (Jacobi) iteration processes is denoted by arrows. The erratic communication is expressed by the varying length of the arrows. At the end of an iteration step of a particular process, locally updated information is sent to its neighbour(s). Vice versa, new information may be received multiple times during an iteration. However, only the most recent information is included at the start of the next iteration step. Other kinds of asynchronous communication are possible [4, 5, 19, 31, 36]. For example, there exists asynchronous iterative methods that immediately incorporate newly received information.

Thus, the execution of the processes does not halt while waiting for new information to arrive from other processes. As a result, it may occur that a process does not receive updated information from one of its neighbours. Another possibility is that received information is outdated in some sense. Also, the duration of each iteration step may vary significantly, caused by heterogeneity in computer hardware and network capabilities, and fluctuations in things like processor workload and problem characteristics.

Some of the main advantages of parallel asynchronous algorithms are summarised in the following list.

- *Reduction of the synchronisation penalty.* No global synchronisation is performed, which may be extremely expensive in a heterogeneous environment.
- *Efficient overlap of communication with computation.* Erratic network behaviour may induce complicated communication patterns. Computation is not stalled while waiting for new information to arrive and more Jacobi iterations can be performed.
- *Coarse–graininess.* Techniques from domain decomposition can be used to effectively divide the computational work and the lack of synchronisation results in a highly attractive computation/communication ratio.

In extremely heterogeneous computing environments, these features can potentially result in improved parallel performance. However, no method is without disadvantages and asynchronous algorithms are no exception. The following list gives some idea on the various difficulties and possible bottlenecks.

- *Suboptimal convergence rates.* Block Jacobi–type methods exhibit slow convergence rates. Furthermore, if no synchronisation is performed whatsoever, processes perform their iterations based on potentially outdated information. Consequently, it is conceivable that important characteristics of the solution may propagate rather slowly throughout the domain.
- *Non–trivial convergence detection.* Although there are no synchronisation points, knowing when to stop may require a form of global communication at some point.
- *Partial fault tolerance.* If a particular Jacobi process is terminated, the complete iteration process will effectively break down. On the other hand, a process may become unavailable due to temporary network failure. Although this would delay convergence, the complete convergence process would eventually finish upon reinstatement of said process.
- *Importance of load balancing.* In the context of asynchronism, dividing the computational work efficiently may appear less important. However, significant desynchronisation of the iteration processes may negatively impact convergence rates. Therefore, some form of (resource–aware) load balancing could still be appropriate.

4 Acceleration: subspace methods

The major disadvantage of block Jacobi–type iterations — either synchronous or asynchronous — is that they suffer from slow convergence rates and that they only converge under certain strict conditions. These methods can be improved significantly as follows. Using a starting vector x_0 and the initial residual $r_0 = b - Ax_0$, iteration (2) may be rewritten as

$$Mu_k = r_k, \quad c_k = Au_k, \quad x_{k+1} = x_k + u_k, \quad r_{k+1} = r_k - c_k, \quad k = 0, 1, \ldots. \tag{6}$$

Instead of finding a new approximation using information solely from the previous iteration, *subspace methods* operate by iteratively constructing some special subspace and extracting an approximate solution from this subspace. The key difference is that information is used from several previous iteration steps, resulting in more efficient methods. This is accomplished by performing (non–standard) projections, which suggests that inner products need be to computed. As mentioned before, in the context of Grid computing this is an expensive operation and should be avoided as much as possible.

Some popular subspace methods are: the Conjugate Gradient method, GCR, GMRES, Bi–CGSTAB, and IDR(s) [27, 33, 40, 47, 57]. Roughly speaking, these methods differ from each other in the way they exploit certain properties of the underlying linear system. Purely for illustrative purposes, the Conjugate Gradient method is listed in Alg. 2, which is designed for symmetric systems. The four main building blocks of a subspace method can be identified as follows.

Algorithm 2 The preconditioned Conjugate Gradient method.

INPUT: Choose x_0; Compute $r_0 = b - Ax_0$;
OUTPUT: Approximation of $Ax = b$;

for $k = 1, 2, \ldots,$ until convergence **do**
  Solve $Mz_{k-1} = r_{k-1}$;
  Compute $\rho_{k-1} = (r_{k-1}, z_{k-1})$;
  if $k = 1$ **then**
    Set $p_1 = z_0$;
  else
    Compute $\beta_{k-1} = \rho_{k-1}/\rho_{k-2}$;
    Set $p_k = z_{k-1} + \beta_{k-1} p_{k-1}$;
  end if
  Compute $q_k = Ap_k$;
  Compute $\alpha_k = \rho_{k-1}/(p_k, q_k)$;
  Set $x_k = x_{k-1} + \alpha_k p_k$;
  Set $r_k = r_{k-1} - \alpha_k q_k$;
end for

1. *Vector operations.* These include inner products and vector updates. Note that classical methods lack inner products.
2. *Matrix–vector multiplication.* This is generally speaking the most computationally intensive operation per iteration step. Therefore, the total number of iterations until convergence is a measure for the cost of a particular method.
3. *Preconditioning phase.* The matrix M in the iteration (6) is sometimes viewed as a *preconditioner*. The ancient and secret art of preconditioning is to find the optimal trade–off between the cost of solving systems involving M and the effectiveness of the newly obtained update u_k. That is, an effective but costly preconditioner will reduce the number of (outer) iterations, but the cost of solving said systems may be too large. Vice versa, applying some cheap preconditioner may be fast, but the resulting number of outer iterations may increase rapidly.
4. *Convergence detection.* Choosing an appropriate halting procedure is not entirely trivial. This has two main reasons: (i) the residual r_k that is computed does not need to resemble the actual residual $b - Ax_k$, and (ii) computing the norm of the residual requires an inner product.

For most applications, finding an efficient preconditioner is more important than the choice of subspace method and it may be advantageous to put much effort in the preconditioning step. A popular choice is to use so–called *incomplete factorisations* of the coefficient matrix as preconditioners, e.g., ILU and Incomplete Cholesky. Another well–known strategy is to approximate the solution to $A\varepsilon = r$ by performing one or more iteration steps of some iterative method, such as block Jacobi or IDR(s). Algorithms that use such a strategy are known as *inner–outer* methods.

A direct consequence of the latter approach is that the preconditioning step may be performed *inexactly*. Unfortunately, most subspace methods can potentially break down if a different preconditioning operator is used in each iteration step. An example is the aforementioned preconditioned Conjugate Gradient method. Methods

that can handle a varying preconditioner are called *flexible*, e.g., GMRESR [56], FGMRES [38], and flexible Conjugate Gradients [2, 37, 44]. A major disadvantage of some flexible methods is that they can incur additional overhead in the form of inner products.

4.1 Hybrid methods: best of both worlds

The potentially large number of synchronisation points in subspace methods make them less suitable for Grid computing. On the other hand, the improved parallel performance of asynchronous algorithms make them perfect candidates.

To reap the benefits and awards of both techniques, the authors propose in [17] to use an asynchronous iterative method as a preconditioner in a flexible iterative method. By combining a slow but coarse–grain asynchronous preconditioning iteration with a fast but fine–grain outer iteration, it is believed that high convergence rates may be achieved on Grid computers.

For their particular application the flexible method GMRESR is used as the outer iteration and asynchronous block Jacobi as the preconditioning iteration. The proposed combined algorithm exhibits many of the features that are on the algorithmic wishlist given in Sect. §2. These include the following items.

- *Coarse–grained.* The asynchronous preconditioning iteration can be efficiently performed on Grid hardware with the help of domain decomposition techniques.
- *Minimal amount of synchronisation points.* When using this approach, a distinction has to be made between global and local synchronisation points. Global synchronisation occurs when information is exchanged between the preconditioning iteration and the outer iteration, whereas local synchronisation only takes place within the outer iteration process. By investing a large amount of time in the preconditioning iteration, the number of expensive global synchronisations can be reduced to a minimum. Subsequently, the number of outer iterations also diminishes, reducing the number of local synchronisation points.
- *Multiple instances of asynchronous communication.* Within the preconditioning iteration asynchronous communication is used, allowing for efficient overlap of communication with computation. Furthermore, the outer iteration process does not need to halt while waiting for a new update u to arrive. It may continue to iterate until a new complete update can be incorporated.
- *Resource–aware and dynamic.* A simple static partitioning scheme may be used for the preconditioner and repartitioning can be performed each outer iteration step. Any load imbalance that may have occurred during the preconditioning iteration will then automatically be resolved.
- *Increased fault tolerance.* In the preconditioning phase, each server iterates on a unique part of the vector u. In heterogeneous computing environments, servers may become temporarily unavailable or completely disappear at any time, potentially resulting in loss of computed data. If the asynchronous process is used to

solve the main linear system, these events would either severely hamper convergence or destroy convergence completely. Either way, by using the asynchronous iteration as a preconditioner — assuming that the outer iteration is performed on reliable hardware — the whole iteration process may temporarily slow down in the worst case, but is otherwise unaffected.

In addition, the proposed algorithm has several highly favorable properties.

- *No expensive asynchronous convergence detection.* By spending a fixed amount of time on preconditioning in each outer iteration step, there is no need for a — possibly complicated and expensive — convergence detection algorithm in the asynchronous preconditioning iteration.
- *Highly flexible and extendible iteration scheme.* The algorithm allows for many different implementation choices. For example, highly recursive iteration schemes may be used. That is, it could be possible to solve a sub–block from a block Jacobi iteration step in parallel on some distant non–dedicated cluster. Another possibility is that the processors that perform the preconditioning iteration do not need to be equal to the nodes performing the outer iteration.
- *The potential for efficient multi–level preconditioning.* The spectrum of a coefficient matrix is the set of all its eigenvalues. Generally speaking, the speed at which a problem is iteratively solved depends on three key things: the iterative method, the preconditioner, and the spectrum of the coefficient matrix. The second and third component are closely related in the sense that a good preconditioner should transform (or *precondition*) the linear system into a problem that has a more favorable spectrum. Many important large–scale applications involve solving linear systems that have highly unfavorably spectra, which consist of many large and many small eigenvalues. The large eigenvalues can be efficiently handled by the asynchronous iteration. On the other hand, the small and more difficult eigenvalues require advanced preconditioners, which can be neatly incorporated in the outer iteration. In this way, both small and large eigenvalues may be efficiently handled by the combined preconditioner. This is just one example of the possibilities.

Naturally, the algorithm is not perfect and there are several potential draw–backs. The main bottlenecks are as follows.

- *Robustness issues.* There are several parameters which have a significant impact on the performance of the complete iteration process. Determining the optimal parameters for a specific application may be a difficult issue. For example, 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.
- *Algorithmic and parallel efficiency issues.* The preconditioning operator varies in each outer iteration step. In most cases this implies that a flexible method has to be used, which can introduce additional overhead in the outer iteration. In order to avoid potential computational bottlenecks, the outer iteration has to be performed in parallel as well. In addition, it is well–known that block Jacobi–type

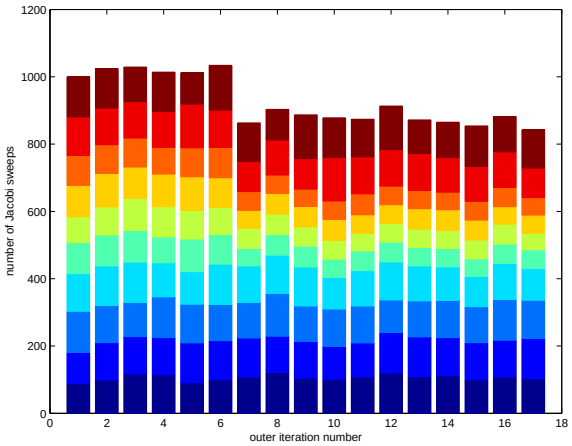

Fig. 3 This experiment is performed using ten servers on a large heterogeneous and non–dedicated local cluster during a normal working day. The Figure shows the number of Jacobi iterations — broken down for each server — during each outer iteration step. Here, a fixed amount of time is devoted to each preconditioning step. After the sixth outer iteration several nodes begin to exhibit an increased workload, its effect clearly noticeable.

methods are slowly convergent for a large number of subdomains. In the current context of large–scale scientific computing, this problem needs to be addressed as well.

Despite these crucial issues, the proposed algorithm has the potential to be highly effective in Grid computing environments.

4.2 Some experimental results

In order to give a rough idea on the effect a heterogeneous computing environment may have on the performance of the proposed algorithm, two illustrative experiments will be discussed. Figure 3 shows the effect heterogeneity can have on the number of Jacobi iterations performed by each server. The effect of the variability in computational environment on the amount of work is clearly visible.

The second experiment illustrates the potential gain of desynchronising part of a subspace method, i.e., in this case the preconditioner. In Fig. 4 some problem is solved using both an asynchronous and a synchronous preconditioner. For this particular application, the use of asynchronous preconditioning nearly cuts the total computing time in half.

These experiments conclude the first and general part of the chapter. The second part of the chapter contains more advanced topics and deals with specific implementation issues.

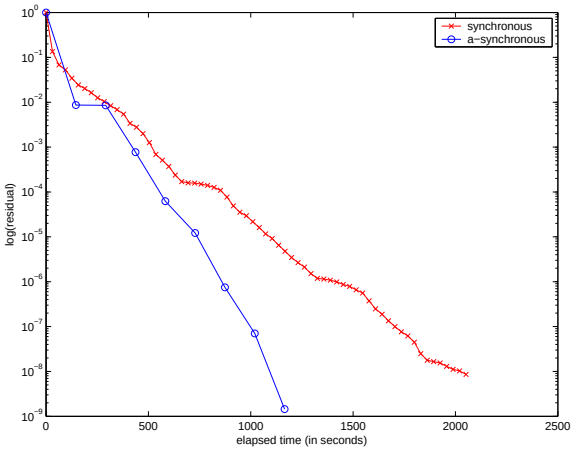

Fig. 4 In this experiment a comparison is made between synchronous and asynchronous preconditioning. The problem to be solved consists of one million equations using four servers within a heterogeneous computing environment. Each point represents a single outer iteration step. By devoting a significant (and fixed) amount of time to asynchronous preconditioning, the number of expensive outer iterations is reduced considerably, resulting in reduced total computing time.

5 Efficient numerical algorithms in Grid computing

The implementation of numerical methods on Grid computers is a complicated process that uniquely combines many concepts from mathematics, computer science, and physics. In the second part of this chapter the various facets of the whole process will be discussed in detail. Most of the concepts given here are taken from [16, 17, 18].

Four key ingredients may be distinguished when implementing numerical algorithms on Grid computers: (i) the numerical algorithm, (ii) the Grid middleware, (iii) the target hardware, and last but not least, (iv) the application. Choosing one particular component can have great consequences on the other components. For example, some middleware may not be suitable for a particular type of hardware. Another possibility is that some applications require that specific features are present in the algorithm.

The discussion will take place within the general framework of the aforementioned proposed algorithm, i.e., a flexible method in combination with an asynchronous iterative method as a preconditioner. As previously argued, it possesses many features that make it perfectly suitable for Grid computing. Furthermore, two important classes of Grid middleware will be discussed and correspondingly, two types of target hardware. Although the current approach is applicable to a wide range of scientific applications, the main focus will be on problems originating from large–scale computational fluid dynamics.

The exposition is concluded by briefly mentioning several more advanced techniques.

Table 3 Several characteristics of two types of Grid middleware.

CRAC	GridSolve
dedicated hardware	non–dedicated hardware
direct communication	bridge communication
asynchronous iterative algorithms	general algorithms
miscellaneous applications	embarrassingly parallel problems
data persistence	non–persistent data
no fault tolerance	fault tolerant

5.1 Grid middleware

One of the primary components in Grid computing is the *middleware*. It serves as the key software layer between the user and the computational resources. The middleware is designed to facilitate client access to remote resources and to cope with issues like heterogeneity and volatility. In which manner the middleware handles these important issues will be briefly discussed.

Although Grid middleware comes in many different shapes and sizes, the focus will be on two leading examples, i.e., GridSolve [26, 61] and CRAC [20]. Table 3 lists some *prototypical* classifications pertaining to both middleware. Most of these classifications are directly related in the sense that some middleware are better suited for particular applications than others. As an example, the bridge communication used in GridSolve would make it more appropriate for embarrassingly parallel problems.

5.1.1 Brief description of GridSolve

GridSolve 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 [42]. The GridRPC Application Programming Interface (API) is defined within the Global Grid Forum [35]. Other projects that implement the GridRPC API are DIET [14], NetSolve [43], Ninf–G [51], and OmniRPC [41].

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. In the context of the current version of GS[1] (see Fig. 5) these components will be discussed in detail. 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

[1] Latest version is v0.17.0 as of May 4th, 2008.

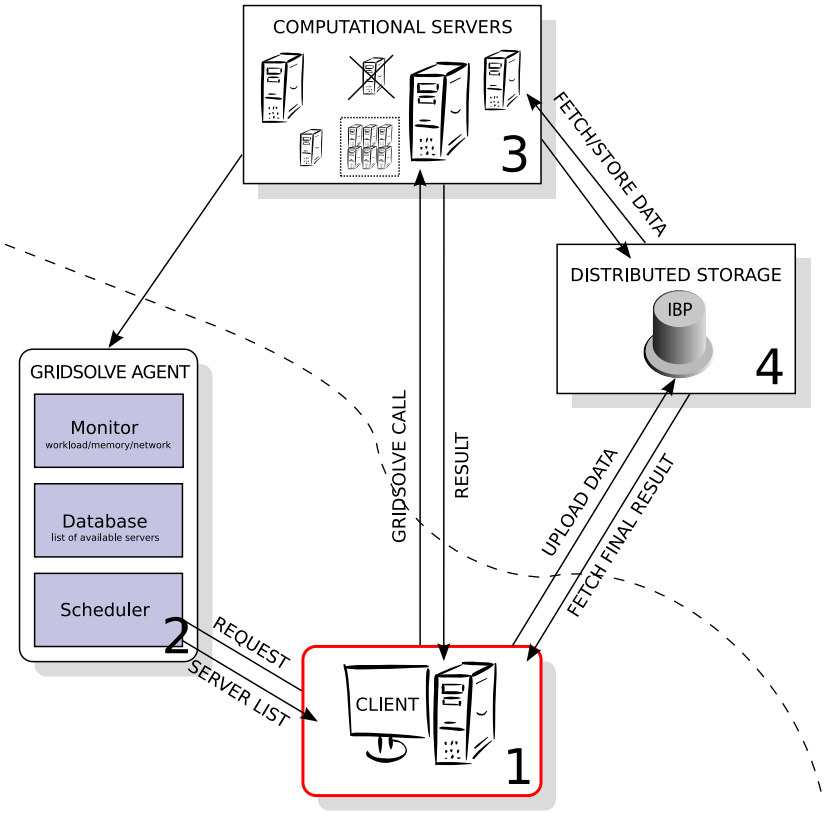

Fig. 5 Schematic overview of GridSolve. The dashed line symbolises (geographical) distance between the client and servers.

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) 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 cur-

rent 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 [13, 12, 34, 62, 22]. 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 [6]. 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. 5). Using the DSI can be considered as programming for a shared memory model.

5.1.2 Brief description of CRAC

The Grid middleware CRAC (*Communication Routines for Asynchronous Computations*) was developed by Stéphane Domas at Laboratoire d'Informatique de Franche–Comté (LIFC) and is specifically designed for efficient implementation of parallel asynchronous iterative algorithms. It allows for direct communication between the processors, both synchronous and asynchronous.

The CRAC library is primarily intended for dedicated parallel systems consisting of geographically separated computational resources. For this reason there are no built–in facilities for detecting properties like varying workload or other types of heterogeneity in computational hardware. However, the object orientated approach of the software ensures that such functionalities can be easily incorporated.

In the current version of CRAC^2, there are no countermeasures in place for handling resources that have completely failed. It is the responsibility of the algorithm designer to make sure that such an event does not destroy the convergence process. Furthermore, it is not yet possible to add or remove computational resources during an iteration process.

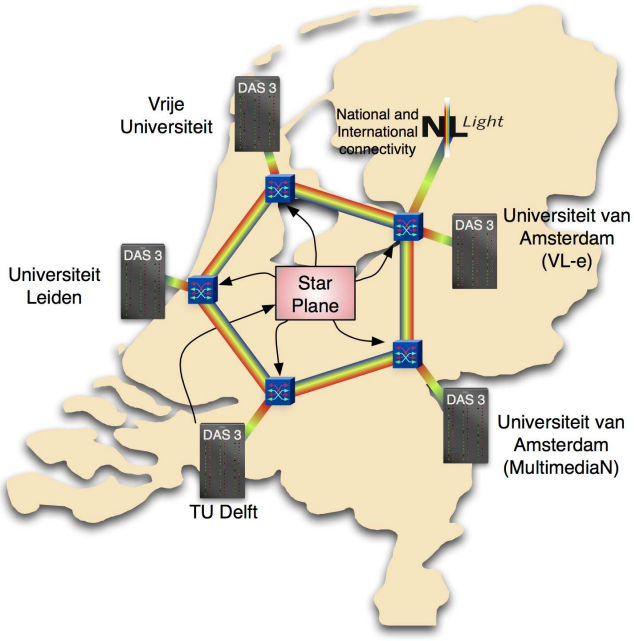

Fig. 6 The DAS–3 supercomputer and StarPlane.

5.2 Target hardware

There exists numerous computing platforms that may be qualified as Grid computing hardware. However, for the purpose of this chapter the focus will be on the following two architectures.

1. *Local networks of non–dedicated computers associated with organisations,* such as universities and companies. These networks typically consists of the computers used daily by employees. Such hardware may considerably differ in speed, memory size, and availability. An example of such a cluster is the network at the Numerical Analysis department at the Delft University of Technology.
2. *Cluster of dedicated clusters linked by a high–speed network.* For example, the Dutch DAS–3 national supercomputer is a cluster of five clusters, located at four academic institutions across the Netherlands, connected by specialised fiber optic technology (i.e., *StarPlane* [48]). It is designed for dedicated parallel computing and although each cluster separately is homogeneous, the system as a whole can be considered heterogeneous. For more specific details on the architecture see Tab. 4 and Fig. 6.

[2] Latest version is v1.0 as of May 4th, 2008.

Table 4 DAS–3: five clusters, one system.

Cluster	Nodes	Type	Speed	Memory	Storage	Node HDDs	Network
VU	85	dual–core	2.4 GHz	4 GB	10 TB	85 x 250 GB	Myri-10G and GbE
LU	32	single–core	2.6 GHz	4 GB	10 TB	32 x 400 GB	Myri-10G and GbE
UvA	41	dual–core	2.2 GHz	4 GB	5 TB	41 x 250 GB	Myri-10G and GbE
TUD	68	single–core	2.4 GHz	4 GB	5 TB	68 x 250 GB	GbE (no Myri-10G)
UvA-MN	46	single–core	2.4 GHz	4 GB	3 TB	46 x 1.5 TB	Myri-10G and GbE

Not surprisingly, the most likely candidates for these types of Grid hardware are GridSolve and CRAC, respectively.

5.3 Parallel iterative methods: building blocks revisited

The next vital step in implementing numerical algorithms on Grid computers is to revisit the four building blocks of subspace methods as mentioned in Sect. §4. Where appropriate, each item will be discussed in the context of the aforementioned types of target architectures.

Dividing the work is an essential aspect of parallel iterative methods. Traditional load balancing aims to divide the computational work as evenly as possible under the constraint of minimal communication. In most cases, this is achieved by a form of hypergraph partitioning algorithm, such as *Mondriaan* [54]. In addition, the current methodology dictates that the load balancer incorporates properties related to the heterogeneity of the computational hardware into the partitioning process [52]. Also, the computational effort involved with the partitioning process itself is far from negligible and may be performed in parallel as well [23].

It is not unlikely that the preconditioning iteration is performed on completely different hardware as the outer iteration. Taking the DAS–3 architecture as an example, the outer iteration may be performed on a single cluster, while the preconditioning iteration is performed utilising all five clusters. The point is that the data distribution used in the outer iteration may be different from the data distribution used in the preconditioning iteration.

Depending on the type of Grid middleware, it may be advantageous to perform the preconditioning iteration on the same hardware as the outer iteration in order to preserve data locality.

5.3.1 Matrix–vector multiplication

Partitioning the matrix–vector multiplication may be done in numerous ways. In the current type of application, the number of non–zeros on each row of the coefficient matrix is roughly the same. A simple but effective distribution is the one–

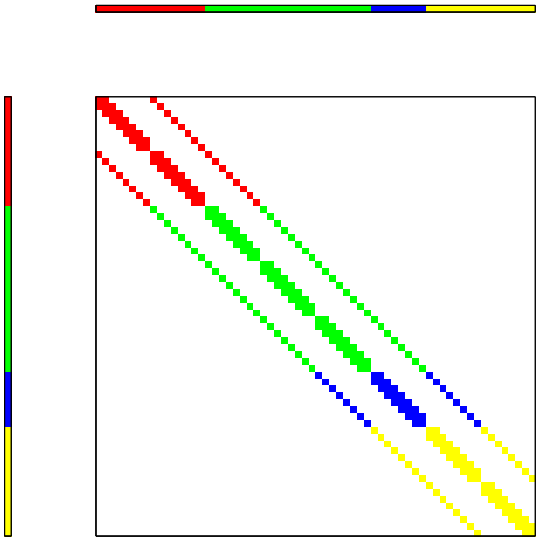

Fig. 7 Heterogeneous one–dimensional block–row partitioning for four servers of a two–dimensional *Poisson* problem. The input (shown at the top) and output (shown left) vectors are partitioned identically.

dimensional block–row partitioning, depicted in Fig. 7. When performing the parallel matrix–vector multiplication only nearest–neighbour communication is required. Nevertheless, nothing prevents the algorithm designer from using more advanced partitioning algorithms in the outer iteration, such as the aforementioned hypergraph partitioner.

The bulk of the computational work in the outer iteration is comprised of the matrix–vector multiplication. Taking into account the fact that the general idea is to minimise the total number of outer iterations, it is unlikely that this operation will be the computational bottleneck of the complete algorithm. As a result, efficient load balancing of the matrix–vector multiplication appears less crucial.

5.3.2 Vector operations

In every subspace method, a newly obtained vector from a preconditioning step is orthogonalised against one or more previous vectors. This is done by an orthogonalisation procedure, such as classical Gram–Schmidt. Although this procedure has good parallel properties, it may suffer from numerical instabilities. This may be remedied by using a selective reorthogonalisation procedure [11, 21].

5.3.3 Preconditioning step

An efficient and robust preconditioner is crucial for rapid convergence of iterative methods. Generally speaking, preconditioners fall into three different classes.

1. *Algebraic techniques.* These methods exploit algebraic properties of the coefficient matrix, such as sparsity patterns and size of matrix elements. For example, incomplete factorisations such as Incomplete Cholesky and block ILU [39].
2. *Domain decomposition techniques.* Most applications in scientific computing involving solving some partial differential equation on a computational domain. Often, the domain can be divided quite naturally into subdomains that may be handled more efficiently. Examples include block Jacobi and alternating Schwarz methods [46].
3. *Multilevel techniques.* Solutions often contain both slow varying and high varying components. By solving the same problem at different scales in a recursive manner, these components can be efficiently captured. Example of such methods are multigrid, deflation, and domain decomposition with coarse grid correction [30, 59].

Efficient parallelisation of a preconditioner is a difficult problem, especially in extremely heterogeneous computational environments. A possible solution is to use an asynchronous iterative method as a preconditioner. In addition, by using a flexible method as the outer iteration, the preconditioning operator is allowed to vary in each outer iteration step and the preconditioning iteration may be performed on unreliable computational hardware.

In the context of asynchronism, efficient load balancing of the preconditioning iteration appears less important. Nevertheless, significant desynchronisation of the Jacobi processes may result in suboptimal convergence rates and some form of load balancing may be appropriate.

The bulk of the computational work in the preconditioning iteration consists of solving the block diagonal system in each Jacobi iteration step. As opposed to the work performed by the outer iteration, this amount is difficult to predict. The reason is that the local linear systems are solved iteratively and in most cases inexactly. Furthermore, problem characteristics may cause highly erratic convergence rates. These issues make efficient load balancing highly problematic.

5.3.4 Convergence detection

The final but essential component of iterative methods is knowing when to stop. In the proposed algorithm a distinction has to be made between convergence detection in the preconditioning iteration and convergence detection in the outer iteration. In most cases, the outer iteration is performed on reliable hardware in a local manner and as a result, convergence detection in the outer iteration is relatively straightforward.

Matters are far more complicated for the preconditioning step. If the preconditioning iteration is performed on unreliable computational hardware as may be the case with GridSolve in combination with a local network of non–dedicated hardware, it is difficult to construct a robust and efficient convergence detection algorithm. In this case, some form of time–dependent stopping criteria may be more appropriate. An obvious disadvantage is that determining the ideal amount of said time may be extremely problem–dependent.

On the other hand, if the preconditioning iteration is performed on dedicated but geographically separated hardware such as the DAS–3 architecture, some sophisticated decentralised convergence detection algorithm may have to be employed [55]. In analogy to the aforementioned case, determining how accurate one should solve the preconditioning iteration is far from trivial.

5.4 Applications

Many important large–scale problems from computational fluid dynamics are solved on highly refined meshes in conjunction with large jumps in the coefficients. The arising linear systems are often severely ill–conditioned and finding efficient (parallel) preconditioners for these systems is vital to fast solution methods. Examples of said applications are swimming of fish, airflow around wind turbine rotor blades, and bubbly flow.

The presence of many large and many small eigenvalues severely hampers convergence rates, which can only be remedied by using sophisticated multi–level preconditioners. As previously mentioned, such preconditioners can be efficiently incorporated in the proposed algorithm.

For these type of multiphase flow applications, the so–called *Immersed Boundary Method* (IBM) is extremely appropriate. Although IBMs come in many different flavours, they all share one common characteristic. Instead of adapting the computational mesh to the (possibly complex and moving) boundary, an IBM immerses the boundary on simple Cartesian meshes and modifies the governing equations in the vicinity of the boundary. The use of fixed and structured meshes expedites the implementation of numerical algorithms immensely, particularly in a parallel context. For a more thorough discussion on IBMs the reader is kindly referred to the chapter elsewhere in this book.

5.5 Advanced techniques

Block Jacobi iterations and domain decomposition techniques are closely related. Combined with the large–scale size of the linear systems involved, some type of *coarse grid correction* within the asynchronous preconditioning iteration may become appropriate. However, the inherently global nature of these techniques may

not suit the current context of asynchronism. Nevertheless, this approach warrants further investigation.

There exists a large number of multi–level preconditioning methods, some more robust than others. Finding the most efficient technique for the current application is also a vital research question.

6 Concluding remarks and further reading

In the early days of iterative methods, Jacobi and Gauss–Seidel iterations for solving linear systems were quite popular. However, their slow convergence rates and strict convergence conditions severely limited the applicability of such methods to the constantly increasing pool of computational problems. This was followed by the discovery of subspace methods in conjunction with incomplete factorisations as *preconditioners*, which immensely boosted the popularity of iterative methods for solving large sparse linear systems from a wide variety of applications.

Then came the era of parallel and vector processing, which rekindled the interest in classical methods as highly parallel block preconditioners. The need for increasingly realistic simulations motivated using the aggregated power of computational resources, which introduced the problem of dealing with heterogeneity. The lack of any synchronisation and coarse–graininess in parallel *asynchronous* classical iterations motivated the idea of using these methods for solving linear systems on large heterogeneous networks of computers.

Nowadays, history is repeating itself and said asynchronous iterations are being used — again as parallel preconditioners — in *flexible* subspace methods, where the preconditioner is allowed to change in each iteration step. By combining the best of worlds, extremely large sparse linear systems may be solved on extremely large heterogeneous networks of computers.

Designing efficient numerical algorithms for Grid computing is a complex process that brings together many different scientific disciplines. By using an asynchronous iterative method as a preconditioner in a flexible iterative method, an algorithm is obtained that has the potential to reap the benefits and awards of both cluster and Grid computing. In this chapter a comprehensive study is made of the various advantages and disadvantages of said approach. Some of the advantages include coarse–graininess and fault tolerance, while potential robustness issues warrant further investigation.

In addition, the efficient implementation of these algorithms on Grid computers depends on many aspects related to the type of target hardware, Grid middleware, and the application. Some of these aspects were also discussed in detail. It is believed that the proposed algorithm has the potential to perform efficient large–scale numerical simulations on loosely coupled networks of computers in various fields of science.

Large sparse linear systems are emerging from a constantly growing number of scientific applications and finding efficient preconditioners for these problems is be-

coming increasingly important. This observation has partly motivated the decision of using an asynchronous iterative method as a preconditioner. However, there are many other potential applications of this kind of preconditioner. For example, an asynchronous iterative method could be used as a so–called *smoother* in *Multigrid*, which in itself is often used as a preconditioner. Another possibility is using an asynchronous method to approximate the *correction equation* in large–scale eigenvalue problems.

It is evident that there are many interesting applications and that much research is still needed. It is hoped that the reader has gained some understanding of the complexities related to the design of efficient numerical algorithms for Grid computers.

For the interested reader, the book by Dimitri Bertsekas and John Tsitsiklis contains a wealth of information on parallel asynchronous iterative algorithms for various applications [9]. Furthermore, more extensive discussions on various aspects of parallel scientific computing may be found in the excellent book by Rob Bisseling [10].

For a comprehensive discussion on iterative methods for solving linear systems, the classic book by Gene Golub and Charles van Loan is greatly recommended, as well as the more recent book by Henk van der Vorst [32, 58]. More on domain decomposition techniques can be found in [46, 53]. For more technical details on Grid hardware and Grid software technologies, the reader is referred to [8, 24, 25, 29]. The recent overview article on iterative methods by Valeria Simoncini and Daniel Szyld is also highly recommended [45]. Another excellent overview article by Michele Benzi discusses various types of preconditioning techniques [7].

Extensive experimental results and specific implementation details pertaining to implementing numerical algorithms on Grid computers may be found in [16, 17, 18].

Acknowledgements The work of the first author was financially supported by the Delft Centre for Computational Science and Engineering. This work is performed as part of the research project *"Development of an Immersed Boundary Method, Implemented on Cluster and Grid Computers, with Application to the Swimming of Fish."* and is joint work with Barry Koren and Yunus Hassen from CWI. The Netherlands Organisation for Scientific Research (NWO) is gratefully acknowledged for the use of the DAS–3. The authors would like to thank the GridSolve team for their prompt response pertaining to our questions and also Stéphane Domas for his prompt and extensive responses pertaining to our questions regarding the CRAC programming system. They also thank Hans Blom for information on the performance of the DAS–3 network system and Kees Verstoep for answering questions regarding DAS–3 inner workings. Figure 6 was kindly donated by Xu Lin, whilst Fig. 8 has been provided by Tobias Baanders. Paulo Anita kindly provided information on the communication patterns induced by the algorithm on the DAS–3 cluster.

References

1. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public–resource computing. Commun. ACM **45**(11), 56–61 (2002). DOI http://doi.acm.org/10.1145/581571.581573

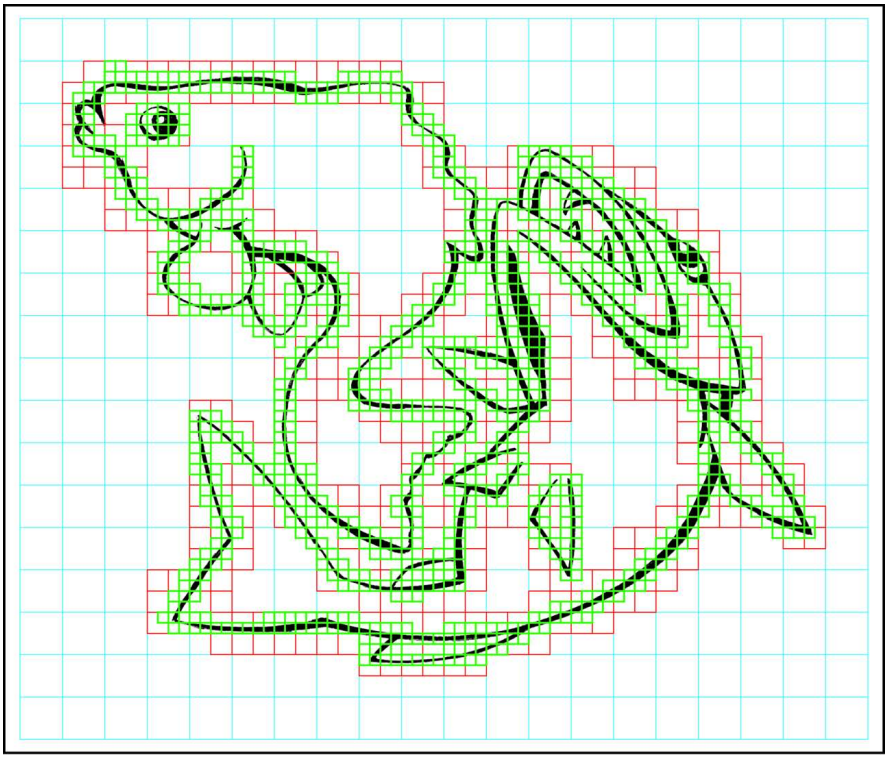

Fig. 8 Artist's Impression of Fishes in Immersed Boundaries.

2. Axelsson, O.: Iterative solution methods. Cambridge University Press, New York, NY, USA (1994)
3. 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), 439–461 (2005). DOI http://dx.doi.org/10.1016/j.parco.2005.02.009
4. Baz, D.E.: A method of terminating asynchronous iterative algorithms on message passing systems. Parallel Algorithms and Applications **9**, 153–158 (1996)
5. Baz, D.E., Spiteri, P., Miellou, J.C., Gazen, D.: Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. J. Parallel Distrib. Comput. **38**(1), 1–15 (1996). DOI http://dx.doi.org/10.1006/jpdc.1996.0124
6. 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), 1773–1787 (2002). DOI http://dx.doi.org/10.1016/S0167-8191(02)00185-0
7. Benzi, M.: Preconditioning techniques for large linear systems: a survey. J. Comput. Phys. **182**(2), 418–477 (2002). DOI http://dx.doi.org/10.1006/jcph.2002.7176
8. Berman, F., Fox, G., Hey, A.J.G.: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons, Inc., New York, NY, USA (2003)
9. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation: Numerical Methods. Prentic Hall, Englewood Cliffs NJ (1989)

10. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP and MPI. Oxford University Press (2004)
11. Björck, Å.: Solving linear least squares problems by Gram–Schmidt orthogonalization. BIT **7**, 1–21 (1967)
12. Brady, T., Konstantinov, E., Lastovetsky, A.: SmartNetSolve: High level programming system for high performance Grid computing. IEEE Computer Society, Rhodes Island, Greece (2006). CD-ROM/Abstracts Proceedings
13. Caron, E., Del-Fabbro, B., Desprez, F., Jeannot, E., Nicod, J.M.: Managing data persistence in network enabled servers. Sci. Program. **13**(4), 333–354 (2005)
14. 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), 335–352 (2006)
15. Chronopoulos, A.T., Gear, C.W.: *S*–step iterative methods for symmetric linear systems. J. Comput. Appl. Math. **25**(2), 153–168 (1989). DOI http://dx.doi.org/10.1016/0377-0427(89)90045-9
16. 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, pp. 527–540 (2007)
17. Collignon, T.P., van Gijzen, M.B.: Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. Tech. rep., Delft University of Technology, Delft, the Netherlands (2008). DUT Report 08–08
18. Collignon, T.P., van Gijzen, M.B.: Two implementations of the preconditioned Conjugate Gradient method on a heterogeneous computing grid with applications to 3D bubbly flow problems. Tech. rep., Delft University of Technology, Delft, the Netherlands (2008). DUT Report 08–??
19. Couturier, R., Denis, C., Jézéquel, F.: GREMLINS: a large sparse linear solver for grid environment. Parallel Computing (2008)
20. Couturier, R., Domas, S.: CRAC: a Grid Environment to solve Scientific Applications with Asynchronous Iterative Algorithms. In: 21th IEEE and ACM Int. Symposium on Parallel and Distributed Processing Symposium, IPDPS'2007, p. 289 (8 pages). IEEE computer society press, Long Beach, USA (2007)
21. 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**, 772–795 (1976)
22. 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), pp. 193–200. IEEE Computer Society, Washington, DC, USA (2004)
23. Devine, K., Boman, E., Heaphy, R., Bisseling, R., Catalyurek, U.: Parallel hypergraph partitioning for scientific computing. In: Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06). IEEE (2006)
24. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A. (eds.): Sourcebook of Parallel Computing. Morgan Kaufmann (2003)
25. Dongarra, J., Lastovetsky, A.: An overview of heterogeneous high performance and Grid computing. Engineering the Grid: Status and Perspective (2006)
26. Dongarra, J., Li, Y., Shi, Z., Fike, D., Seymour, K., YarKhan, A.: Homepage of NetSolve/GridSolve (2007)
27. Eisenstat, S.C., Elman, H.C., Schultz, M.H.: Variational iterative methods for nonsymmetric systems of linear equations. SIAM J. Numer. Anal. **20**, 345–357 (1983)
28. Folding: Folding@home distributed computing. `http://folding.stanford.edu/`
29. Foster, I., Kesselman, C.: The Grid: Blueprint for a new Computing Infrastructure, second edn. Morgan Kaufman Publishers (2004)
30. Frank, J., Vuik, C.: On the construction of deflation–based preconditioners. SIAM J. Sci. Comput. **23**(2), 442–462 (2001). DOI http://dx.doi.org/10.1137/S1064827500373231

31. Frommer, A., Szyld, D.B.: Asynchronous iterations with flexible communication for linear systems. Calculateurs Parallèles Réseaux et Systèmes Répartis **10**, 421–429 (1998)
32. Golub, G.H., Van Loan, C.F.: Matrix Computations (Johns Hopkins Studies in Mathematical Sciences). The Johns Hopkins University Press (1996)
33. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for solving linear systems. Journal of Research of National Bureau Standards **49**, 409–436 (1952)
34. Lastovetsky, A., Zuo, X., Zhao, P.: A non–intrusive and incremental approach to enabling direct communications in RPC–based grid programming systems. Tech. rep. (2006)
35. Lee, C., Nakada, H., Tanimura, Y.: GridRPC Working Group (2007). `http://forge.ogf.org/sf/projects/gridrpc-wg/`
36. Miellou, J.C., Baz, D.E., Spiteri, P.: A new class of asynchronous iterative algorithms with order intervals. Math. Comput. **67**(221), 237–255 (1998). DOI http://dx.doi.org/10.1090/S0025-5718-98-00885-0
37. Notay, Y.: Flexible conjugate gradients. SIAM Journal on Scientic Computing **22**, 1444–1460 (2000)
38. Saad, Y.: A flexible inner–outer preconditioned GMRES algorithm. SIAM J. Sci. Comput. **14**(2), 461–469 (1993)
39. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
40. Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems. SIAM J. Sci. Stat. Comput. **7**(3), 856–869 (1986)
41. 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, pp. 206–213. IEEE Computer Society, Washington, DC, USA (2003)
42. 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, pp. 274–278. Springer–Verlag, London, UK (2002)
43. Seymour, K., YarKhan, A., Agrawal, S., Dongarra, J.: NetSolve: Grid enabling scientific computing environments. In: L. Grandinetti (ed.) Grid Computing and New Frontiers of High Performance Processing. Elsevier (2005)
44. Simoncini, V., Szyld, D.B.: Flexible inner–outer Krylov subspace methods. SIAM J. Numer. Anal. **40**(6), 2219–2239 (2002). DOI http://dx.doi.org/10.1137/S0036142902401074
45. Simoncini, V., Szyld, D.B.: Recent computational developments in Krylov subspace methods for linear systems. Numerical Linear Algebra with Applications **14**, 1–59 (2007)
46. Smith, B.F., Bjørstad, P.E., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge (1996)
47. Sonneveld, P., van Gijzen, M.B.: IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems. Tech. rep., Delft University of Technology, Delft, the Netherlands (2007). DUT Report 07–07
48. StarPlane: Application-specific management of photonic networks (2007). `http://www.starplane.org/`
49. Sterling, T., Lusk, E., Gropp, W. (eds.): Beowulf Cluster Computing with Linux. MIT Press, Cambridge, MA, USA (2003)
50. de Sturler, E.: Truncation strategies for optimal Krylov subspace methods. SIAM J. Numer. Anal. **36**(3), 864–889 (1999). DOI http://dx.doi.org/10.1137/S0036142997315950
51. 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), 41–51 (2003)
52. Teresco, J.D., Devine, K.D., Flaherty, J.E.: Numerical Solution of Partial Differential Equations on Parallel Computers, chap. Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations. Springer–Verlag (2005)
53. Toselli, A., Widlund, O.B.: Domain Decomposition: Algorithms and Theory, vol. 34. Springer Series in Computational Mathematics, Springer, Berlin, Heidelberg (2005)

54. Vastenhouw, B., Bisseling, R.H.: A two–dimensional data distribution method for parallel sparse matrix-vector multiplication. SIAM Rev. **47**(1), 67–95 (2005). DOI http://dx.doi.org/10.1137/S0036144502409019
55. Vernier, F., Bahi, J.M., Contassot-Vivier, S., Couturier, R.: A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. IEEE Trans. Parallel Distrib. Syst. **16**(1), 4–13 (2005). DOI http://dx.doi.org/10.1109/TPDS.2005.2
56. van der Vorst, H., Vuik, C.: GMRESR: a family of nested GMRES methods. Num. Lin. Alg. Appl. **1**(4), 369–386 (1994)
57. van der Vorst, H.A.: Bi–CGSTAB: A fast and smoothly converging variant of Bi–CG for the solution of nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing **13**(2), 631–644 (1992)
58. van der Vorst, H.A.: Iterative Krylov Methods for Large Linear systems. Cambridge University Press, Cambridge (2003)
59. Wesseling, P.: An Introduction to Multigrid Methods. John Wiley & Sons, Chichester (1992)
60. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: CF '06: Proceedings of the 3rd conference on Computing frontiers, pp. 9–20. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1128022.1128027
61. YarKhan, A., Seymour, K., Sagi, K., Shi, Z., Dongarra, J.: Recent Developments in GridSolve. International Journal of High Performance Computing Applications (IJHPCA) **20**(1), 131–141 (2006)
62. 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). IEEE Computer Society, Long Beach, California, USA (2007)