DELFT UNIVERSITY OF TECHNOLOGY

REPORT 09-12

Fast Iterative Solution of Large Sparse Linear Systems on Geographically Separated Clusters

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

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2009

Copyright © 2009 by Delft Institute of Applied Mathematics Delft, The Netherlands.

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

Fast Iterative Solution of Large Sparse Linear Systems on Geographically Separated Clusters

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

Abstract

Parallel asynchronous iterative algorithms exhibit features that are extremely well–suited for Grid computing, such as lack of synchronisation points. Unfortunately, they also suffer from slow convergence rates. In this paper we propose using asynchronous methods as a coarse–grain preconditioner in a flexible iterative method, where the preconditioner is allowed to change in each iteration step. A full implementation of the algorithm is presented using Grid middleware that allows for both synchronous and asynchronous communication. Advantages and disadvantages of the approach are discussed. Numerical experiments on heterogeneous computing hardware demonstrate the effectiveness of the proposed algorithm on Grid computers, with application to large 2D and 3D bubbly flow problems.

Key words. Grid computing, linear systems of equations, asynchronous iterative methods, flexible iterative methods, geographically separated clusters, bubbly flows.

1 Introduction

This paper describes an efficient iterative method for solving large linear systems on geographically separated computational resources. The algorithm uses an asynchronous iterative method as a preconditioner in a synchronous *flexible* method, where the preconditioner is allowed to vary in each iteration step.

The parallel solution of linear systems using asynchronous iterative methods has been studied in several papers, for example in [9, 2, 3, 6]. For a comprehensive overview paper and more references on asynchronous iterative methods, see [14].

However, asynchronous methods have never gained widespread popularity. The main reason is that the slow convergence rates limit the applicability of these methods. Nevertheless, the lack of global synchronisation points in these methods is a highly favourable

*Delft Institute of Applied Mathematics, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, P.O. Box 5031, 2600 GA Delft,The Netherlands. e-mail: T.P.Collignon@tudelft.nl, M.B.vanGijzen@tudelft.nl

property in parallel computing. This is even more the case in Grid computing, where synchronisation between geographically separated clusters is the bottleneck operation.

Although Krylov subspace methods such as the Conjugate Gradient method [16] offer significantly improved convergence rates, the global synchronisation points induced by the inner product operations in each iteration step limits the applicability. By using an asynchronous iterative method as a *preconditioner* in a Krylov subspace method, the best of both worlds can be combined. It will be shown in this paper that the combination of a slow but asynchronous inner iteration with a fast but synchronous outer iteration results in high convergence rates on heterogeneous networks of computers.

In the proposed inner–outer algorithm, the asynchronous preconditioning iteration is performed on heterogeneous computational resources and for a fixed amount of time. As a result, the preconditioner varies in each outer iteration step, which requires the use of a flexible subspace method as the outer iteration.

The target hardware consists of the DAS–3 Grid computer [20], which is a cluster of five geographically separated clusters spread over four academic institutions in the Netherlands. The DAS–3 is designed for dedicated parallel computing and although each separate cluster is relatively homogeneous, the system as a whole can be considered heterogeneous.

The algorithm is applied to a bubbly flow problem, which is an important and difficult application from computational fluid dynamics in two–phase fluid flow [22]. This application involves the solution of large sparse symmetric and positive definite systems, which leaves the flexible Conjugate Gradient method [1, 18] as the method of choice for the outer iteration. Nevertheless, the proposed approach can be used for non–symmetric systems as well by using a flexible method such as GCR [12, 24] as the outer iteration [8].

In this paper, both the outer iteration and the preconditioning iteration are performed on the same set of *dedicated* computing nodes. A different strategy is used in [8], where these two iteration processes are physically decoupled. That is, the GCR method is used as the outer iteration on the user machine, while the preconditioning iteration is performed on a cluster of *non–dedicated* computers. By physically decoupling the two iterations, an algorithm is obtained that is partially fault–tolerant. Section 3.2 contains further details on this issue.

The algorithm is implemented using the CRAC library, which was developed within the GREMLINS project [10, 9]. The aim of this project is to design efficient iterative algorithms for solving large sparse linear systems on geographically separated computational resources. The CRAC library can be used to easily implement (partially) asynchronous iterative algorithms on such systems.

The experimental results on the DAS–3 multi–cluster demonstrate that the proposed algorithm is highly effective in the context of loosely coupled networks of computers. Furthermore, the results show that the algorithm can adapt to a computational environment in which the network is heavily loaded.

The remainder of the paper is organised as follows. Section 2 describes the complete algorithm, including a discussion of the various advantages and disadvantages of the proposed method. In Sect. 3 various details pertaining to the parallel implementation of the algorithm are discussed, such as the employed Grid middleware CRAC and the data

distribution. In Sect. 4 extensive numerical experiments are performed using the DAS–3 multi–cluster and Sect. 5 contains concluding remarks.

2 Iterative solvers in Grid computing

This section starts by exposing the key bottleneck in iteratively solving large linear systems on Grid hardware: expensive synchronisation. Section 2.2 describes asynchronous parallel iterative methods, which exhibit several characteristics that are extremely suitable for Grid computing. Unfortunately, they also suffer from slow convergence rates. Section 2.3 explains how asynchronism can be introduced into fast but fine–grain subspace methods. The key idea is that by using an asynchronous method as a preconditioner in a flexible subspace method, the best of both worlds can be combined.

2.1 The problem

The goal is to iteratively and efficiently solve large sparse linear systems,

$$Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad x, b \in \mathbb{R}^n, \tag{1}$$

on large, heterogeneous, and geographically separated computational resources. The key characteristic of iterative 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.

In distributed memory computing, each processor operates on its local memory. For many parallel iterative methods this implies that at some point in time a form of synchronisation has to be performed. For extremely large problem sizes, the potentially high number of iteration steps and the high cost of a synchronisation operation poses significant efficiency issues in the context of iterative solvers and heterogeneous computing environments.

2.2 Asynchronous iterations

There exists a class of parallel iterative methods which lack synchronisation points (in theory), making them excellent candidates for heterogeneous computing environments as found in Grid computing. These methods generalise simple iterative methods such as the classical block Jacobi iteration [19]. To compute the solution of the linear system $Ax = b$ using p processors, the coefficient matrix A, the solution vector x, and the right–hand side vector b are partitioned into non–overlapping blocks as follows,

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}, \quad \text{and } b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{bmatrix}. \tag{2}$$

Algorithm 1 (A–)synchronous block Jacobi iteration on p processors.

1: Choose $x^{(0)}$;
2: **for** $k = 1, 2, \ldots,$ until convergence **do**
3: **for** $i = 1, 2, \ldots, p$ **do**
4: (i.) Solve $A_{ii}x_i^{(k)} = b_i - \sum_{j=1,j\neq i}^{p} A_{ij}x_j^{(k-1)}$; // *synchronous iterations*
5: (ii.) Solve $A_{ii}x_i^{\text{new}} = b_i - \sum_{j=1,j\neq i}^{p} A_{ij}x_j^{\text{old}}$; // *a–synchronous iterations*
6: **end for**
7: **end for**

Algorithm 1 lists the (a)synchronous block Jacobi method for solving this system. In standard block Jacobi, at iteration step k each processor independently solves a linear subsystem — either iteratively or directly — followed by a synchronisation point where information is exchanged between the processors (see line 4). Instead of synchronising at each iteration step k, an asynchronous variant of Alg. 1 performs their local iterations based on information that is available at that particular time (see line 5).

In asynchronous iterations, 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, 9, 13, 17]. For example, asynchronous iterative methods exist where newly received information is immediately incorporated by the iteration processes.

In other words, 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 processor workload and problem characteristics.

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

- *Reduction of the global synchronisation penalty.* No global synchronisations are performed, an operation that 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–grain.* Techniques from domain decomposition can be used to effectively divide the computational work and the lack of synchronisation results in a highly

Algorithm 2 Flexible Conjugate Gradients (pure truncation strategy)

INPUT: Parameters $m_{\max}, \epsilon_{\text{in}}, T_{\max}$; Set $m_k = \min(k, m_{\max})$; Initial guess x_0; Set $r_0 = b - Ax_0$.

1: **for** $k = 0, 1, \ldots,$ until convergence **do**
2: Evaluate $u = \mathcal{M}(r_k, \epsilon_{\text{in}}, T_{\max})$; *// Preconditioning step: see Alg. 3*
3: Compute $u_k = \text{orthonorm}(u, c_i, u_i, k, m_k)$; *// Orthogonalisation step: see Alg. 4*
4: Compute $c_k = Au_k$; *// Matrix–vector multiplication*
5: Compute $\alpha_k = \dfrac{u_k^\top r_k}{u_k^\top c_k}$;
6: Update $x_{k+1} = x_k + \alpha_k u_k$;
7: Update $r_{k+1} = r_k - \alpha_k c_k$;
8: **end for**

favourable computation/communication ratio.

In extremely heterogeneous computing environments, these properties can potentially result in improved parallel performance. However, no method is without its disadvantages and asynchronous algorithms are no exception. The following list gives some idea on the various difficulties and potential 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 propagate slowly throughout the domain. Furthermore, the iteration may even diverge in some cases.

- *Non–trivial convergence detection.* Although there are no inherent 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 killed, 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 the failed 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.

2.3 Best of both worlds

The key disadvantage of block Jacobi–type methods — both synchronous and asynchronous — is that they suffer from slow convergence rates and that they only converge under certain strict conditions [6]. Krylov subspace methods are a class of iterative methods that exhibit significantly improved convergence rates. The main characteristic of these methods is that (non–standard) projections are used to extract a new approximation to the solution from a Krylov subspace. This implies that inner products need to be computed, which introduces global synchronisation points in each iteration step.

The potentially large number of synchronisation points in Krylov methods make them less suitable for Grid computing. Vice versa, the improved parallel performance of asynchronous algorithms make them perfect candidates. To reap the benefits and awards of both techniques, we propose to use an asynchronous iterative method as a preconditioner in a flexible iterative method, where the preconditioner is allowed to change in each iteration step. The goal is to achieve high convergence rates on Grid computers by combining a slow but coarse–grain asynchronous preconditioning iteration with a fast but fine–grain outer iteration.

As mentioned before, the application considered in this paper involves solving a large symmetric positive (semi–)definite system. This suggests that the flexible Conjugate Gradient (FCG) is the method of choice for the outer iteration [1, 18].

Listed in Alg. 2 is the flexible CG method. Three main phases can be distinguished, which are the preconditioning step (line 2), the orthogonalisation step (line 3), and the remaining operations such as the matrix–vector multiplication (line 4) and the vector updates. These phases will be discussed separately.

Asynchronous preconditioning In standard preconditioned CG, the preconditioner is a fixed symmetric and positive definite matrix M such that solving the residual equation $Mu = r$ is 'cheap' in some sense. In the proposed algorithm, the preconditioning operation in line 2 of Alg. 2 consists of an asynchronous iterative method applied to the system $Au = r_k$ and is performed for a fixed amount of time $T_{\max}$. The local systems within the asynchronous method are solved iteratively and with accuracy ϵ_{in}. In other words, the preconditioning step consists of a random (typically nonlinear) process,

$$u = \mathcal{M}(r_k, \epsilon_{\text{in}}, T_{\max}), \qquad \mathcal{M} : \mathbb{R}^n \to \mathbb{R}^n, \tag{3}$$

which differs from one iteration step k to the next. In Alg. 3 the specific steps are shown that are performed by the asynchronous preconditioning iteration processes.

Note that if a fixed amount of time is devoted to each preconditioning step, there is no need for a — possibly complicated and expensive — convergence detection algorithm for the asynchronous preconditioning iteration. Convergence detection can be done efficiently in the outer iteration.

The nonlinearity of the preconditioning step implies that the operator $\mathcal{M}$ does not correspond to some symmetric positive definite matrix M. To minimise the number of expensive (outer) synchronisations, the bulk of the computational work is to be performed by the preconditioner.

Algorithm 3 Asynchronous block Jacobi iteration for task i.

FUNCTION: $u_i = \mathcal{M}(r_i, \epsilon_{\text{in}}, T_{\max})$

1: Wait until r_i is updated; Set $u_i = 0$;
2: **while** $t_{\text{elapsed}} < T_{\max}$ **do**
3: Compute $v_i = r_i - \sum_j A_{ij} u_j$;
4: Solve $A_{ii} p_i = v_i$ with accuracy ϵ_{in};
5: Update $u_i \leftarrow u_i + p_i$;
6: Exchange u_i asynchronously with neighbours;
7: **end while**

Algorithm 4 Modified Gram–Schmidt

FUNCTION: $u_k = \text{orthonorm}(u, c_i, u_i, k, m_k)$;

1: Set $u_k^{(k-m_k)} = u$;
2: **for** $i = k - m_k, \ldots, k-1$ **do**
3: Compute $\beta_i = \dfrac{c_i^\top u_k^{(i)}}{c_i^\top u_i}$;
4: Set $u_k^{(i+1)} = u_k^{(i)} - \beta_i u_i$;
5: **end for**
6: Set $u_k = u_k^{(k)}$;

Orthogonalisation The main difference with standard preconditioned CG are the additional orthogonalisations in line 3 of Alg. 2. The newly obtained search direction vector u has to be orthogonalised with respect to the A–inner product ($\langle x, y\rangle_A := x^\top A y$) against a number of previous search directions.

For practical implementations of flexible methods a truncation or restart strategy has to be applied. In this paper a pure truncation strategy is employed, which basically means that the new search direction vector is orthogonalised against $m_{\max}$ previous vectors, subsequently replacing the oldest search direction vector. This variant will be denoted by FCG($m_{\max}$). Other truncation or restart strategies are possible [18].

In the context of heterogeneous computing environments, choosing an appropriate orthogonalisation procedure becomes critically important. Naturally, the (numerically stable) modified Gram–Schmidt (MGS) procedure introduces expensive global synchronisation points. It is hoped that a low truncation parameter $m_{\max}$ is sufficient, thus keeping the number of expensive synchronisations to a minimum.

Vice versa, the classical Gram–Schmidt algorithm has excellent parallel properties. Although it may suffer from numerical instabilities, this may be remedied by using a (relatively complicated) selective reorthogonalisation procedure [11, 7].

Since it is the intention to devote the bulk of the computational effort to preconditioning, the number of expensive synchronisations induced by the MGS procedure will not pose a significant bottleneck. Therefore, the MGS algorithm is chosen as the orthogonalisation

procedure, which is listed in Alg. 4.

The vector updates do not require any communication, while the matrix–vector multiplication only requires synchronous nearest–neighbour communication.

3 Parallel implementation details

3.1 Brief description of CRAC

The algorithm is implemented using the CRAC library (Communication Routines for Asynchronous Computations), which was developed at Laboratoire d'Informatique de Franche–Comté (LIFC) and is specifically designed for efficient implementation of parallel asynchronous iterative algorithms [10, 9]. It allows for direct communication between the processes, both synchronous and asynchronous. The middleware employs a small set of simple communication primitives, which greatly facilitates the implementation of (partially) asynchronous iterative algorithms.

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

In the context of asynchronous iterative algorithms and heterogeneous environments, messages do not necessarily arrive in the same order as they were sent. Furthermore, iteration processes can desynchronise considerably and it may happen that updated information is received multiple times during a local iteration step. To properly handle these events, CRAC employs so-called message crunching, which is a technique to ensure that a process always operates on the most recent local data.

In the current version of CRAC (v1.0, May 2008), resources that fail completely will cause the complete application to abort. On the other hand, a resource that is temporarily unavailable might not necessarily destroy the iteration process. It is the responsibility of the programmer to make sure that such an event does not result in stagnation. Furthermore, it is not yet possible to add or remove computational resources during an iteration process.

Although MPI–2 has functionalities for handling asynchronous and non–blocking communication, it lacks specific features such as message crunching [15].

3.2 Coupled/decoupled inner–outer iterations

The fact that there are essentially two separate iteration processes opens up a whole range of possibilities with respect to the algorithm, implementation, target hardware, and application.

The DAS–3 multi–cluster is designed for dedicated parallel computing and in order to preserve data locality, the outer iteration and preconditioning iteration are performed on the same set of nodes. With respect to the CRAC library, the possibility of having both

synchronous and asynchronous communication allows for straightforward implementation of both iteration processes.

A disadvantage of this approach is that every single task should be performed on reliable and stable hardware, which may be an unacceptable restriction in the context of Grid computing. In the worst case, should any of the tasks fail, it is not unlikely that important intermediate information is lost, halting the entire iteration process. If a particular node merely becomes temporarily unavailable, the iteration process would be able to continue when this node becomes available again.

In [8] the Grid middleware GridSolve [26] is used, which allows for a natural decoupling of the inner and outer iteration processes. Here, the GCR method is used as the outer iteration and the asynchronous preconditioning is performed on a local cluster of non–dedicated computers used daily by employees. By physically decoupling the outer iteration and the preconditioning iteration, it becomes feasible to perform the inner iteration on unreliable (heterogeneous and distant) computational resources, while the outer iteration is performed on more stable (homogeneous and local) hardware, resulting in a partially fault–tolerant algorithm. Despite the inherent limitations of the employed middleware GridSolve and the extremely volatile nature of the computational resources, encouraging experimental results are obtained.

This decoupled iteration approach is somewhat unnatural in the context of CRAC and the DAS–3 multi–cluster. The two main reasons are that the current version of CRAC cannot properly handle resources that fail completely and that the synchronisation primitives in CRAC are global operations. Synchronisation of a subset of processes is possible, but relatively complicated. The CRAC middleware is more suited for dedicated computational hardware where network connections between nodes may become temporarily unavailable. Thus, for the purpose of this paper the coupled iteration approach is used.

3.3 Data distribution

In theory, the matrix distribution used in the outer iteration may differ from the matrix distribution used in the preconditioning iteration. A disadvantage of this approach is that exchanging the new search direction and updated residual between the outer iteration and the preconditioning iteration becomes non–trivial.

In the preconditioning iteration, a (block and/or heterogeneous) row distribution may be sufficient, due to the specific structure of the matrix. In the outer iteration, a square matrix distribution may be employed (i.e., produced by some (hyper)graph partioning algorithm). However, this is specifically designed to optimise the matrix–vector multiplication, which is not the bottleneck operation for this application. For Laplacian matrices a simple row distribution is sufficient. This also greatly simplifies the exchange of information between the inner and outer iteration when using a decoupled iteration approach.

When using the coupled iteration approach, it is natural to use the same the data distribution for both the outer iteration and the preconditioning iteration in order to maintain data locality.

4 Numerical experiments

4.1 Motivation

Our main goal is to simulate general moving boundary problems on Grid computers. Examples of such problems are the swimming of fish, airflow around wind turbine blades, and bubbly flows. These simulations involve solving the governing fluid equations on structured grids, where the most expensive part consists of solving a large sparse linear system at each time step. When using a pressure–correction method [25] to solve the governing equations for bubbly flows on a highly refined mesh, such a large sparse linear system arises from a finite difference discretisation of the following Poisson equation with discontinuous coefficients and Neumann boundary conditions,

$$\begin{cases} -\nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}) \right) = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ \frac{\partial}{\partial \mathbf{x}} p(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega, \end{cases} \tag{4}$$

for given functions f and g. Here, Ω and $\partial\Omega$ denote the computational domain and boundary respectively, while p and ρ represent the pressure and density. In this paper the test problem from [23, 21] is considered. It is a two–phase bubbly flow problem with two separate fluids Γ_0 and Γ_1, representing water (high–density phase) and vapour (low–density phase) respectively. The corresponding density function has a jump defined by

$$\rho(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in \Gamma_0; \\ \tau, & \mathbf{x} \in \Gamma_1, \end{cases} \tag{5}$$

where typically $\tau = 10^{-3}$. Such a discontinuity in the coefficient results in a highly ill–conditioned linear system, making it a difficult problem for iterative methods. For the purpose of this paper a unit domain is used containing a single bubble with radius $\frac{1}{4}$ located at the center. For more details on applying the pressure–correction method to bubbly flows the reader is referred to [23].

Both 2D and 3D experiments will be performed. Applying standard finite differences to (4) on a structured $n_x \times n_y$ or $n_x \times n_y \times n_z$ mesh results in the linear system $Ax = b$ where $A \in \mathbb{R}^{n \times n}$ is a penta or hepta–diagonal symmetric positive semi–definite (SPSD) sparse matrix with $n = n_x n_y$ or $n = n_x n_y n_z$.

Note that this implies that the solution x is determined up to a constant. It can be shown that this does not pose any problems for the iterative solver [22].

4.2 Target hardware and experimental setup

The Distributed ASCI Supercomputer 3 (DAS–3) is a multi–cluster consisting of five clusters, located at four academic institutions across the Netherlands [20]. The five sites are connected through SURFnet, which is the academic and research network in the Netherlands. Four of the five local clusters are equipped with both Gigabit Ethernet interconnect

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

Table 1: Specific details on the five DAS–3 sites.

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

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

and high speed Myri–10G interconnect. The TUD site only employs Gigabit Ethernet interconnect.

More specific details on the five sites are given in Tab. 1, while Tab. 2 lists average roundtrip measurements between several DAS–3 sites on a lightly loaded network. These facts show that a large amount of heterogeneity exists between the sites with respect to the computational resources and network capabilities, making the DAS–3 a perfect testbed for Grid computing. Note that in this case the preconditioning iteration and the outer iteration are performed using the same computational hardware.

The matrix is partitioned using a homogeneous one–dimensional block–row distribution, both in the preconditioning iteration and in the outer iteration. The vectors are distributed accordingly. The preconditioning step in each outer iteration is performed for a fixed number of $T_{\max}$ seconds and the local systems are solved (inexactly) with relative tolerance $\epsilon_{\text{in}} = 10^{-1}$ using standard CG preconditioned with Incomplete Cholesky.

Experiments reveal that solving the local subdomains more accurately does not result in improved convergence rates. A possible explanation is that the asynchronous block Jacobi iteration is an inherently slow process, which makes the accurate solution of the inner systems ineffectual. The complete linear system is solved with relative tolerance $\epsilon_{\text{outer}} = 10^{-8}$.

In the context of Grid computing, it is natural to fix the problem size per node and investigate the scalability of the algorithm by adding nodes in order to solve bigger problems. The nodes are evenly divided between the five clusters with increments of five nodes, starting with a single node on each cluster.

In each 3D experiment, n_x, n_y, and n_z are chosen such that the number of equations of

FCG($m_{\max}$)	wall clock time (s)	iterations	memory requirements (vectors)
0	> 1000	> 100	4
1	> 1000	> 100	6
3	839	87	10
5	636	68	14
10	572	62	24
15	515	61	34

Table 3: Influence of parameter m ($T_{\max} = 5$s, five nodes, 2D problem).

unknowns on each node is approximately 500,000. The largest experiments are performed using 100 nodes, which implies that the largest 3D problem solved consists of approximately fifty million degrees of freedom. In the 2D experiments the number of unknowns on each node is approximately 250,000.

Since the DAS–3 is solely intended for experimental research, the maximum allowed time for a single job is sixty minutes. All the timing results shown are wall clock times. For comparison studies, synchronous preconditioning is also performed, which involves performing a single block Jacobi iteration step per preconditioning phase. The effectiveness of the asynchronous preconditioner depends on multiple (and random) factors, so these experiments are performed three times (when possible) and the average execution times are given.

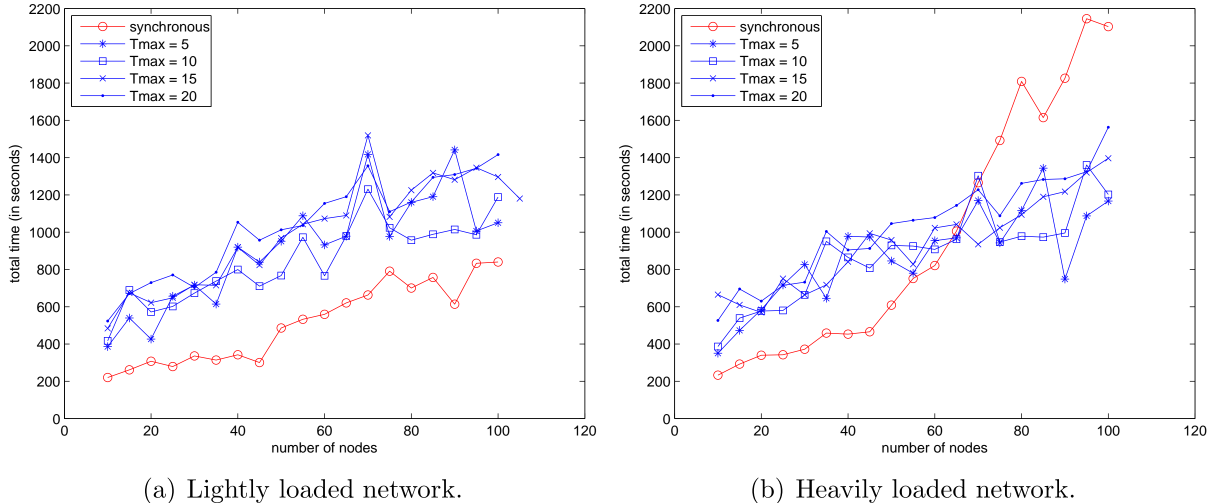
To justify the use of a flexible method, results for a representative experiment using different values of $m_{\max}$ are given in Tab. 3. The number of vectors that needs to be stored for FCG($m_{\max}$) is also given, which is $2m_{\max} + 4$. Note that FCG(0) corresponds to the asynchronous Jacobi iteration, which shows that a fully asynchronous method is impractical for this application. These results indicate that the use of a flexible method is fully justified and that choosing $m_{\max} = 5$ results in a good trade–off between efficiency and memory requirements.

4.3 Experimental results

In order to properly investigate the effectiveness of the proposed algorithm on Grid hardware, the experiments consist of two distinct parts:

1. Experiments using a 3D test problem and where the network load is varied for showing that asynchronous preconditioning adapts to a heterogeneous network environment.

2. Experiments on a lightly loaded network and using a 2D test problem for showing that asynchronous preconditioning can outperform synchronous preconditioning.



(a) Lightly loaded network. (b) Heavily loaded network.

Figure 1: Total execution time (3D problem).

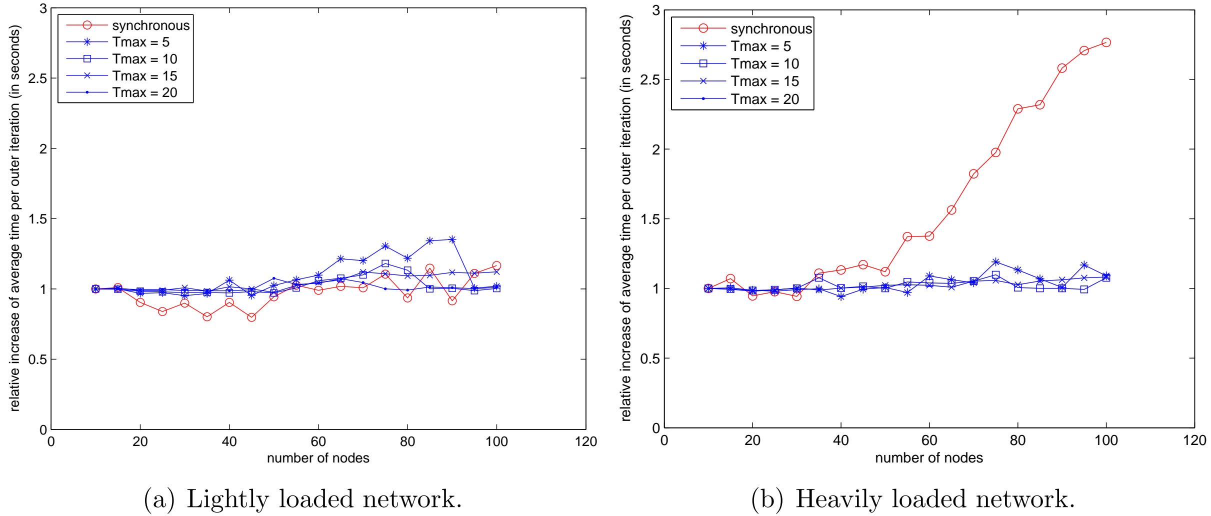
3D experiments

Figure 1(a) shows the total execution time until convergence for different values of $T_{\max} \in \{5, 10, 15, 20\}$ on an *lightly loaded* network. For comparison, results using both asynchronous and synchronous preconditioning are shown. In every experiment, synchronous preconditioning outperforms asynchronous preconditioning. A key observation is that the amount of asynchronous preconditioning does not seem to have a significant impact on the total computing time.

Figure 1(b) shows the total execution time until convergence using up to 100 nodes for different values of $T_{\max} \in \{5, 10, 15, 20\}$ on a *heavily loaded* network. To simulate a loaded network, a special parallel application is used that continuously sends massive amounts of data from all to all processes. Again for comparison, results using synchronous and asynchronous preconditioning are given. In this case, the total execution time for synchronous preconditioning increases significantly when using more than approximately 60 nodes. However, asynchronous preconditioning remains highly effective. These results can be explained by the following two observations.

(i) Time per outer iteration Keeping the problem size per node fixed implies that — in the ideal case where communication overhead is negligible — the execution time per outer iteration is constant. Fig. 2 shows the *relative increase* of the *average* times per outer iteration for both the single and the multi–cluster case.

The results given in Fig. 2(a) for a lightly loaded network shows almost constant average times per outer iteration for both synchronous and asynchronous preconditioning. This indicates that in this case communication overhead is relatively small, which is not surprising.



(a) Lightly loaded network.

(b) Heavily loaded network.

Figure 2: Relative increase of time per outer iteration step (3D problem).

As for the loaded network results, Fig. 2(b) shows that the relative increase in time per outer iteration for synchronous preconditioning is far greater than with asynchronous preconditioning.

(ii) Number of outer iterations Table 4 lists the total number of outer iterations for synchronous and asynchronous preconditioning with $T_{\max} = 15$s. For asynchronous preconditioning, results for a lightly loaded and a heavily loaded network are given. The table shows that when using synchronous preconditioning, the number of outer iterations is relatively large. Combined with the relatively large increase in time per outer iteration when using a loaded network, this explains the major increase in total execution time as seen in Fig. 1(b).

Vice versa, the relatively small number of outer iterations using asynchronous preconditioning — for both a lightly loaded and a heavily loaded network — combined with the relatively small increase in time per outer iteration results in significantly improved parallel performance in a heterogeneous network environment. Again, the total execution time is not significantly affected by the amount of asynchronous preconditioning.

2D experiments

In Fig. 3(a) results are given for $T_{\max} \in \{5, 10\}$ on a lightly loaded network using a 2D test problem. Note that in this case there is less overlap between the subdomains. The numerical results show that synchronous preconditioning is always outperformed by asynchronous preconditioning. For 100 nodes, the total execution time for synchronous preconditioning is almost twice as long as for asynchronous preconditioning.

Figure. 3(b) gives the relative increase in time per outer iteration step. This shows that

number of nodes	synchronous	async. (lightly loaded)	async. (heavily loaded)
10	219	30	31
20	338	39	36
30	371	44	41
40	376	56	52
50	511	61	58
60	561	64	62
70	653	84	55
80	743	70	66
90	665	71	71
100	715	80	80

Table 4: Outer iterations for synchronous and asynchronous preconditioning (3D problem).

number of nodes	synchronous	asynchronous ($T_{\max} = 10\text{s}$)
10	286	77
20	601	91
30	805	139
40	1060	124
50	1388	130
60	1473	141
70	1881	151
80	1905	174
90	2082	175
100	2332	195

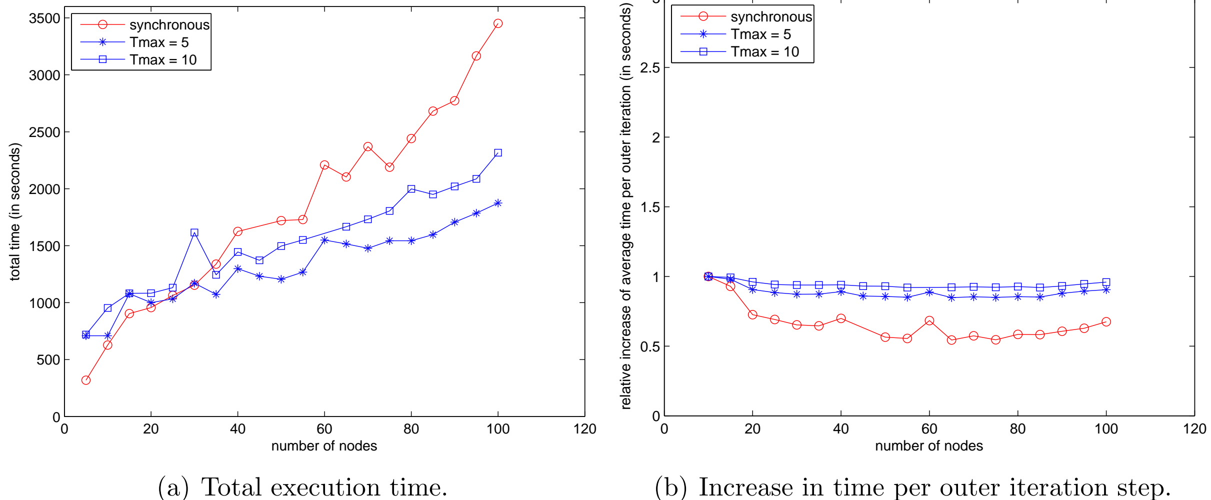
Table 5: Outer iterations for synchronous and asynchronous preconditioning (2D problem).

despite the fact that synchronous preconditioning does not show a relatively large increase in time per outer iteration, it is still outperformed by asynchronous preconditioning. This can be explained by examining the number of outer iterations, which are shown in Table 5. For synchronous preconditioning, using twice the number of nodes almost doubles the number the outer iterations. In contrast, using asynchronous preconditioning increases the number of outer iterations merely by a factor of approximately 1.4. As a result, asynchronous preconditioning is more effective.

4.4 Discussion

Increasing the problem size by adding nodes has the following adverse consequences.

1. The coefficient matrix becomes increasingly ill–conditioned; and
2. the number of subdomains in asynchronous block Jacobi increases.



(a) Total execution time.

(b) Increase in time per outer iteration step.

Figure 3: Lightly loaded network (2D problem).

Both these effects have a negative impact on the number of outer iterations. The first consequence is inherent to the problem and the second effect applies to all block Jacobi–type preconditioners. A possible third consequence is that the average number of Jacobi iteration steps per node decreases due to increased communication. However, this was not observed in the experiments. Also, factors that may have a large impact on the effectiveness of the preconditioner are the heterogeneity of the hardware and the variations in network activity.

Despite these unfavourable conditions the experimental results show a fairly limited increase in total computing time for increasing number of nodes, which suggests that the asynchronous iterative method is an effective preconditioner in the context of Grid computing.

5 Concluding remarks

5.1 Main conclusions and contributions

The efficient iterative solution of large sparse linear systems on Grid computers is a difficult problem. The induced heterogeneity and volatile nature of the aggregated computational resources present numerous algorithmic challenges. Synchronisation is the critical bottleneck of parallel subspace methods in the context of loosely coupled networks of computers. By using an asynchronous iterative method as a preconditioner in a synchronous subspace method, the number of expensive synchronisations can be reduced significantly.

Extensive numerical experiments using approximately 100 nodes divided between five geographically separated clusters show that:

1. Using the partially asynchronous algorithm is more efficient than using (i.) a fully synchronous method or using (ii.) a fully asynchronous method;
2. The asynchronous preconditioner adapts to a computational environment in which the network is heavily loaded;

Therefore, the proposed partially asynchronous algorithm is highly effective in iteratively solving large–scale linear systems within the context of heterogeneous networks of computers.

5.2 Future work

The CRAC middleware has specific functionalities for efficient implementation of asynchronous iterative algorithms, such as message crunching. However, the MPI–2 library also contains several routines related to asynchronous communication. It may be instructive to compare the communication libraries used by CRAC and MPI–2 by measuring network performance such as latency and throughput.

References

[1] Owe Axelsson. *Iterative Solution Methods.* Cambridge University Press, New York, NY, USA, 1994.

[2] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Asynchronism for iterative algorithms in a global computing environment. In *HPCS '02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 90–97, Washington, DC, USA, 2002. IEEE Computer Society Press.

[3] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Comput.*, 31(5):439–461, 2005.

[4] D. El Baz. A method of terminating asynchronous iterative algorithms on message passing systems. *Parallel Algorithms and Applications*, 9:153–158, 1996.

[5] Didier El Baz, Pierre Spiteri, Jean Claude Miellou, and Didier Gazen. Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. *J. Parallel Distrib. Comput.*, 38(1):1–15, 1996.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice–Hall, Englewood Cliffs, N.J., 1989. republished by Athena Scientific, 1997.

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

[8] Tijmen P. Collignon and Martin B. van Gijzen. Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. Technical report, Delft University of Technology, Delft, the Netherlands, 2008. DUT report 08–08 (submitted).

[9] Raphaël Couturier, Christophe Denis, and Fabienne Jézéquel. GREMLINS: a large sparse linear solver for grid environment. *Parallel Computing*, December 2008.

[10] Raphaël Couturier and Stéphane Domas. 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*, page 289 (8 pages), Long Beach, USA, March 2007. IEEE computer society press.

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

[12] Stanley C. Eisenstat, Howard C. Elman, and Martin H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20:345–357, 1983.

[13] Andreas Frommer and Daniel B. Szyld. Asynchronous iterations with flexible communication for linear systems. *Calculateurs Parallèles Réseaux et Systèmes Répartis*, 10:421–429, 1998.

[14] Andreas Frommer and Daniel B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.

[15] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI–2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[16] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for solving linear systems. *Journal of Research of National Bureau Standards*, 49:409–436, 1952.

[17] J. C. Miellou, D. El Baz, and P. Spiteri. A new class of asynchronous iterative algorithms with order intervals. *Math. Comput.*, 67(221):237–255, 1998.

[18] Y. Notay. Flexible Conjugate Gradients. *SIAM Journal on Scientific Computing*, 22:1444–1460, 2000.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[20] Frank J. Seinstra and Kees Verstoep. DAS–3: The distributed ASCI supercomputer 3, 2007. http://www.cs.vu.nl/das3/.

[21] J. M. Tang and C. Vuik. On deflation and singular symmetric positive semi–definite matrices. *J. Comput. Appl. Math.*, 206(2):603–614, 2007.

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

[23] S.P. van der Pijl, A. Segal, C. Vuik, and P. Wesseling. A mass–conserving Level–Set method for modelling of multi–phase flows. *International Journal for Numerical Methods in Fluids*, 47:339–361, 2005.

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

[25] J. van Kan. A second–order accurate pressure correction scheme for viscous incompressible flow. *SIAM J. Sci. Stat. Comput.*, 7(3):870–891, 1986.

[26] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent developments in GridSolve. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1):131–141, 2006.