DELFT UNIVERSITY OF TECHNOLOGY

REPORT 10-11

Exploiting the flexibility of IDR(s) for grid computing

Martin B. van Gijzen and Tijmen P. Collignon

ISSN 1389-6520

Reports of the Department of Applied Mathematical Analysis

Delft 2010

Exploiting the flexibility of IDR(s) for grid computing

Martin B. van Gijzen and Tijmen P. Collignon [*]

March 29, 2010

Abstract

The IDR(s) method that is proposed in [26] is an efficient limited memory method for solving large nonsymmetric systems of linear equations. In [11] an IDR(s) variant is described that has a single synchronisation point per iteration step, which makes this variant well-suited for parallel and grid computing. In this paper, we combine this IDR(s) variant with an a-synchronous preconditioning iteration to further improve the performance of IDR(s) on a grid computer. A-synchronous preconditioners do not require expensive synchronisation and adapt to volatile computational resources, and are therefore well-suited for such a computational environment. However, an a-synchronous preconditioning operation is also non-constant by nature: the preconditioner changes in every iteration. The success of the combination of IDR(s) with an a-synchronous preconditioner therefore depends on the flexibility of IDR(s). We will explain why IDR(s) can be used as a flexible method, and we will successfully use the combination of IDR(s) with an a-synchronous preconditioner for solving large convection-diffusion problems. The numerical experiments are performed on the DAS-3 grid computer, which is composed of five geographically separated parallel clusters.

Keywords. Iterative methods, IDR(s), Krylov-subspace methods, grid computing, flexible methods, a-synchronous preconditioning

AMS subject classification. 65F10, 65F50

1 Introduction

The IDR(s) method and its derivatives are short recurrence Krylov subspace methods for iteratively solving large nonsymmetric linear systems

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

[*]Delft University of Technology, Delft Institute of Applied Mathematics, Mekelweg 4, 2628 CD Delft, The Netherlands. E-mail: `M.B.vanGijzen@tudelft.nl, T.P.Collignon@tudelft.nl`

The method has attracted considerable attention, e.g., see [24, 19, 25, 21, 22]. For $s = 1$, IDR(s) is mathematically equivalent to the ubiquitous Bi-CGSTAB algorithm [27]. For important types of problems and for relatively small values of $s > 1$, IDR(s) outperforms the Bi-CGSTAB method.

The parallelisation of IDR(s) – in particular on Grid computers – is addressed in [11]. The IDR(s) variant that is derived in that paper is called IDR(s) *minsync* and has only one global synchronisation point per iteration. Here we define an iteration as all the operations to compute a new approximate solution and corresponding residual. In IDR(s) one matrix-vector product is performed per iteration. The IDR(s) *minsync* algorithm is a reformulation of the efficient and stable IDR(s) *bi-ortho* algorithm that is described in [29]. The IDR(s) *bi-ortho* and *minsync* variants require the same amount of arithmetic operations. The main difference is that in IDR(s) *minsync* only at one point in an iteration step inner products with global vectors are explicitly computed, which means that all (global) communication for these inner products can be combined. All other inner products are computed via scalar updates, which do not require communication.

In [11] we only evaluated the unpreconditioned IDR(s) *minsync* algorithm. For most problems in practice, however, a preconditioner needs to be applied to speed-up the iterative process. The development of efficient and well-parallelisable preconditioners is still an area of active research. Constructing preconditioners that are suitable for grid computing, where computational resources are volatile and synchronisation is prohibitively expensive is still in its infancy.

A-synchronous iterative methods have attracted some attention in the past for parallel computing, e.g., see [6, 7, 8, 9, 15, 17, 16, 18]. The main advantage of a-synchronous iterative methods is that no global synchronisation is necessary. However, convergence of a-synchronous methods is often slow and not always guaranteed. Moreover, efficient convergence detection is a difficult issue. Therefore, with the advent of fast communication networks the interest for these methods diminished. Over the past years, however, a-synchronous iteration methods have received renewed attention in the context of grid computing. The cost of global synchronisation on grid computers can be prohibitive, which makes the lack of synchronisation points a feature of pivotal importance. Another interesting feature of a-synchronous methods is that they adapt to variations in the computational resources, for example in computational load or in network load. References for a-synchronous methods for grid computing are for example [3, 4, 5, 2, 12, 13].

In [10], an a-synchronous iterative method is successfully used as a preconditioner in the flexible Conjugate Gradient (FCG) method [1, 20] to solve large sparse symmetric linear systems originating from a 3D bubbly flow problem. The experimental results showed that using the partially asynchronous algorithm is more efficient than using a fully synchronous method or a fully a-synchronous method. The results also showed that the a-synchronous preconditioner adapts to a computational environment in which the network load varies strongly. Following these results, we will combine in this paper a-synchronous preconditioning with IDR(s) to solve large convection-diffusion problems. A-synchronous preconditioners are non-constant by nature and are therefore in principle only applicable to speed-up *flexible* iterative methods, where the preconditioner is allowed to change in each iteration step. Examples of such methods are flexible CG and GMRESR [28]. We will explain in this paper why an a-synchronous preconditioner can be used with IDR(s) algorithms.

This paper is organised as follows. In Section 2 the IDR(s) *minsync* method from [11] is reproduced, and the parallelisation of the algorithm is explained. Also, a description of an a-synchronous preconditioner is given and it is explained why a non-constant preconditioner can be combined with IDR(s), without compromising the final accuracy of the solution. Section 3 contains extensive numerical experiments on the DAS-3 grid computer. We make concluding remarks in Section 4.

2 Combining IDR(s) with an a-synchronous preconditioner

2.1 An IDR(s) variant with a minimum number of synchronisation points

We first review the IDR(s) *minsync* algorithm and its parallelisation. The (right) preconditioned version of the algorithm is shown in Alg. 1. The derivation of the algorithm is described in detail in [11]. It is mathematically equivalent with the IDR(s) *bi-ortho* variant that is described in [29].

The IDR(s) *minsync* algorithm is composed of only a few building blocks: matrix-vector multiplications (lines 10 and 33), preconditioning operations (lines 12 and 32), vector updates (lines 11, 13, 18, 23, 24, 35 and 36), inner products (lines 15 and 34), and scalar operations (lines 10, 16, 20, 22, 26–29, 37). All the operations can be straightforwardly parallelised on distributed memory computers by making a block partitioning of the matrix and of the vectors:

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

Each block is assigned to a processor; all data corresponding to a block is processed by its corresponding processor and are stored in the processors local memory. Using this approach, the different operations can be parallelised as follows:

- The vector updates are performed locally: every processor performs the update for its part of the vector without communication.

- The inner products are computed by computing the local inner products and broadcasting the results to all other processors. This is an inherently global operation.

Algorithm 1 IDR(s) *minsync* with bi-orthogonalisation of intermediate residuals and with minimal number of synchronisation points

INPUT: $A \in \mathbb{C}^{N\times N}; x, b \in \mathbb{C}^N; Q \in \mathbb{C}^{N\times s}$; preconditioner $B \in \mathbb{C}^{N\times N}$; accuracy ε
OUTPUT: Approximate solution x such that $||b - Ax|| \leq \varepsilon$.

```
 1: // Initialisation
 2: $G = U = 0 \in \mathbb{C}^{N\times s}; M = [\mu] = I \in \mathbb{C}^{s\times s}; \omega = 1$
 3: Compute $r = b - Ax$
 4: $\phi = Q^H r, \phi = (\phi_1, \ldots, \phi_s)^T$
 5: // Loop over nested $\mathcal{G}_j$ spaces, $j = 0, 1, \ldots$
 6: while $||r|| > \varepsilon ||b||$ do
 7:    // Compute $s$ linearly independent vectors $g_k$ in $\mathcal{G}_j$
 8:    for $k = 1$ to $s$ do
 9:       // Compute $v \in \mathcal{G}_j \cap Q^\perp$
10:       Solve $M_l \gamma_{(k:s)} = \phi_{(k:s)}$
11:       $v = r - \sum_{i=k}^{s} \gamma_i g_i$
12:       $\tilde{v} = B^{-1} v$ // Preconditioning step
13:       $\widehat{u}_k = \sum_{i=k}^{s} \gamma_i u_i + \omega \tilde{v}$ // Intermediate vector $\widehat{u}_k$
14:       $\widehat{g}_k = A\widehat{u}_k$ // Intermediate vector $\widehat{g}_k$
15:       $\boxed{\psi = Q^H \widehat{g}_k}$ // $s$ inner products (combined)
16:       Solve $M_t \alpha_{(1:k-1)} = \psi_{(1:k-1)}$
17:       // Make $\widehat{g}_k$ orthogonal to $q_1, \ldots, q_{k-1}$ and update $\widehat{u}_k$ accordingly
18:       $g_k = \widehat{g}_k - \sum_{i=1}^{k-1} \alpha_i g_i, \quad u_k = \widehat{u}_k - \sum_{i=1}^{k-1} \alpha_i u_i$
19:       // Update column $k$ of $M$
20:       $\mu_{i,k} = \psi_i - \sum_{j=1}^{k-1} \alpha_j \mu_{i,j}^c$ for $i = k, \ldots, s$
21:       // Make $r$ orthogonal to $q_1, \ldots, q_k$ and update $x$ accordingly
22:       $\beta = \phi_k / \mu_{k,k}$
23:       $r \leftarrow r - \beta g_k$
24:       $x \leftarrow x + \beta u_k$
25:       // Update $\phi = Q^H r$
26:       if $k + 1 \leq s$ then
27:          $\phi_i = 0$ for $i = 1, \ldots, k$
28:          $\phi_i \leftarrow \phi_i - \beta \mu_{i,k}$ for $i = k + 1, \ldots, s$
29:       end if
30:    end for
31:    // Entering $\mathcal{G}_{j+1}$. Note: $r \perp Q$
32:    $\tilde{v} = B^{-1} r$ // Preconditioning step
33:    $t = A\tilde{v}$
34:    $\boxed{\omega = (t^H r)/(t^H t); \phi = -Q^H t}$ // $s + 2$ inner products (combined)
35:    $r \leftarrow r - \omega t$
36:    $x \leftarrow x + \omega \tilde{v}$
37:    $\phi \leftarrow \omega \phi$
38: end while
```

– The matrix-vector products are computed by performing local products with the submatrices. Multiplication with a diagonal block A_{ii} does not require communication, but multiplication with an off-diagonal block requires communication with another processor. Note that almost all off-diagonal blocks are zero in the type of application we are interested in.

– The scalar operations are inexpensive and are not parallelised, but are performed on all processors.

– The preconditioning operation will be discussed separately below.

For our type of application, the communication for the matrix-vector multiplication is nearest-neighbour only and does not require global synchronisation. The inner products do require global communication, involving all the processors, and are therefore synchronisation points in the algorithm. In the IDR(s) *minsync* algorithm shown in Alg. 1 the operations are organised so that all communication for the inner products can be combined. As a result, there is only one global synchronisation point per iteration step.

2.2 A-synchronous preconditioning

A-synchronous preconditioners are attractive in the context of grid computing since they do not require global synchronisation and can adapt to changes in computational load and network load [10]. In the IDR(s) *minsync* algorithm, an a-synchronous preconditioning step is performed by applying an a-synchronous iterative method to the system $A\tilde{v} = v$ (line 12) or $A\tilde{v} = r$ (line 32) for a fixed amount of time $T_{\max}$.

Algorithm 2 (A-)synchronous block Jacobi iteration using p processors applied to $A\tilde{v} = r$.

```
1: Initialize $\tilde{v}^{(0)}$;
2: for $n = 1, 2, \ldots,$ until convergence do
3:     for $i = 1, 2, \ldots, p$ do
4:         (i.) Solve $A_{ii}\tilde{v}_i^{(n)} = r_i - \sum_{j=1, j\neq i}^{p} A_{ij}\tilde{v}_j^{(n-1)}$; // synchronous iterations
5:         (ii.) Solve $A_{ii}\tilde{v}_i^{\text{new}} = r_i - \sum_{j=1, j\neq i}^{p} A_{ij}\tilde{v}_j^{\text{old}}$; // a-synchronous iterations
6:     end for
7: end for
```

A-synchronous methods generalise simple iterative methods such as the classical block Jacobi iteration. In the standard synchronous Jacobi iteration process (see line 4 of Alg. 2), the processors operate in parallel on their part of the vector $\tilde{v}^{(n)}$, followed by a synchronisation point at each Jacobi iteration step n. In our a-synchronous algorithm (see line 5 of Alg. 2), a processor computes $\tilde{v}^{\text{new}}$ using information $\tilde{v}^{\text{old}}$ that is available on the process at that particular time. As a result, each separate block Jacobi iteration process may use out-of-date information, but the lack of synchronisation points and the reduction of communication can potentially result in improved parallel performance. Note that in practical implementations, the inner systems of Alg. 2 are often solved (approximately) by some other iterative method.

De-synchronising the preconditioning phase in this manner has the advantage that: (i) the preconditioner can be easily and efficiently parallelised on Grid computers, (ii) no additional synchronisation points are introduced, and (iii) by devoting the bulk of the computational effort to the preconditioner, the computation to communication ratio can be improved significantly, while reducing the number of expensive (outer) synchronisations considerably.

2.3 Flexible preconditioning in IDR(s) methods

The a-synchronous preconditioning step consists of a random (typically nonlinear) process,

$$\tilde{v} = \mathcal{B}(r), \qquad \mathcal{B} : \mathbb{R}^N \to \mathbb{R}^N, \tag{3}$$

which differs from one iteration to the next. The IDR(s) algorithm, however, is designed for constant preconditioners, and its theoretical properties rely on this. So the question arises whether we can use IDR(s) with a non-constant preconditioner, i.e., can we use IDR(s) as a flexible method?

```
for $k = 1$ to $s$ do
    $v = r - G\gamma$
    $u = \omega v + U\gamma$
    $g = AB^{-1}u$
    $r = r - g$
    $y = y + u$
end for
$x = B^{-1}y$
```

```
for $k = 1$ to $s$ do
    $v = r - G\gamma$
    $B^{-1}u = \omega B^{-1}v + B^{-1}U\gamma$
    $g = AB^{-1}u$
    $r = r - g$
    $B^{-1}y = B^{-1}y + B^{-1}u$
end for
$x = B^{-1}y$
```

```
for $k = 1$ to $s$ do
    $v = r - G\gamma$
    $u' = \omega B^{-1}v + U'\gamma$
    $g = Au'$
    $r = r - g$
    $x = x + u'$
end for
$\times$
```

Figure 1: Flexible preconditioning in IDR(s), shown for a single cycle.

In the following it is shown that the (simplified) key recursions of IDR(s) remain valid within the context of flexible preconditioning. Applying a right preconditioner B^{-1} to the system $Ax = b$ gives

$$AB^{-1}y = b, \quad x = B^{-1}y. \tag{4}$$

Applying the key IDR(s) recursions to the preconditioned system (4) gives the leftmost part of Fig. 1, where the final solution is obtained by $x = B^{-1}y$. The main problem with this approach is that if the preconditioner B changes in each iteration step, the computed iterates do not correspond to the computed residuals.

This inconsistency can be remedied by premultiplying y with B^{-1}, scaling back the iterates (middle part of Fig. 1). Again the solution is obtained by $x = B^{-1}y$. Note that as a result, the update u is also scaled back. Defining $u' = B^{-1}u$ for the new updates and setting $x = B^{-1}y$ gives the correct right-preconditioned recursions for IDR(s) (rightmost part of Fig. 1). The iterate x and residual r are now computed in a consistent manner (and basically independent of how the update u' is constructed).

Note that according to the theoretical finite termination property of IDR(s), finite termination at the exact solution should occur within $N + N/s$ iterations [26]. However, this no longer holds if IDR(s) is used as a flexible method. However, the method is still finite for $s = N$, since in that case a full set of basis vectors for $\mathbb{R}^N$ is generated and the method terminates at the exact solution after N iteration.

3 Numerical experiments

3.1 Test problem

As a test problem we take the following three-dimensional convection-diffusion equation:

$$\nabla^2 u + w u_x = f(x, y, z), \tag{5}$$

defined on the unit cube $[0,1]\times[0,1]\times[0,1]$. Homogeneous Dirichlet conditions are imposed on the boundaries. The vector f is such that the solution is

$$u = \exp(xyz)\sin(\pi x)\sin(\pi y)\sin(\pi z). \tag{6}$$

Discretisation by the finite difference scheme with a seven point stencil on a uniform $n_x \times n_y \times n_z$ grid results in a sparse linear system of equations $Ax = b$ where A is of order $N = n_x n_y n_z$. Centered differences are used for the first derivatives. The grid points are numbered using the standard (lexicographic) ordering.

The matrix Q consists of s orthogonalised random vectors. The initial guess is set to $x_0 \equiv 0$ and the iteration is terminated when $||r||/||b|| \leq \varepsilon \equiv 10^{-6}$. At the end of the iteration process convergence is verified by comparing the true residual with the iterated final residual.

3.2 Hardware and grid middleware

The target hardware consists of the distributed ASCI Supercomputer 3 (DAS-3), which is a cluster of five geographically separated clusters spread over four academic institutions in the Netherlands [23]. The DAS-3 multi-cluster is designed for dedicated parallel computing and although each separate cluster is relatively homogeneous, the system as a whole can be considered to be heterogeneous.

We have used the CRAC library [13] to implement the complete algorithm. CRAC has been specifically designed to build parallel iterative asynchronous applications.

3.3 Results for unpreconditioned IDR(s)

In order to illustrate the parallel performance of the basic, unpreconditioned algorithm we consider a test problem of $N = 256^3 \approx 17,000,000$ equations. For the convection parameter we take $w = 100$. The domain is partitioned into rectangular cuboids. A total of four of the five clusters is used, with 32 nodes on each cluster. We have excluded the fifth cluster in this experiment since it uses a slower communication network. Figures 2(a) and 2(b) show the speed-up of the algorithm, which includes the speed-up results for the IDR(s) *bi-ortho* algorithm for comparison. Clearly, the minimum number of synchronisation points in IDR(s) *minsync* results in a better parallel performance, in particular for higher values of s.

For completeness, the total number of iterations of both variants for this test problem is shown in Tab. 1. Also, Fig. 2(c) and 2(d) shows the total wall clock times of both variants. This shows that the reduction in iteration steps for increasing s is similar for both variants. However, for this test problem and particular computational hardware, the differences in execution times of both variants is relatively small.

s	IDR(s) *minsync*	IDR(s) *bi-ortho*
1	1362	1422
3	948	916
5	870	882
10	737	737

Table 1: Total number of iterations.

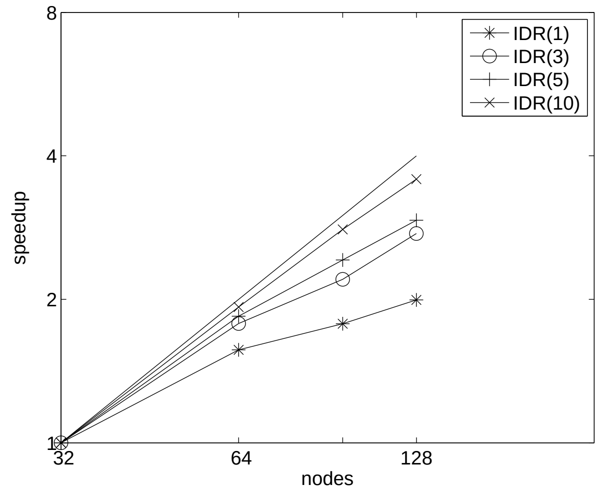
3.4 Results for IDR(s) with a-synchronous preconditioning

For the asynchronous preconditioning experiments, the domain is partitioned in horizontal slices along the z-direction. For all experiments we use 60 computing nodes, distributed evenly over all five sites. The problem size is $N = 180^3 \approx 6,000,000$ equations and the local systems in the preconditioning iteration are solved using (truncated) GCR(100) [14] with relative accuracy 10^{-1}.

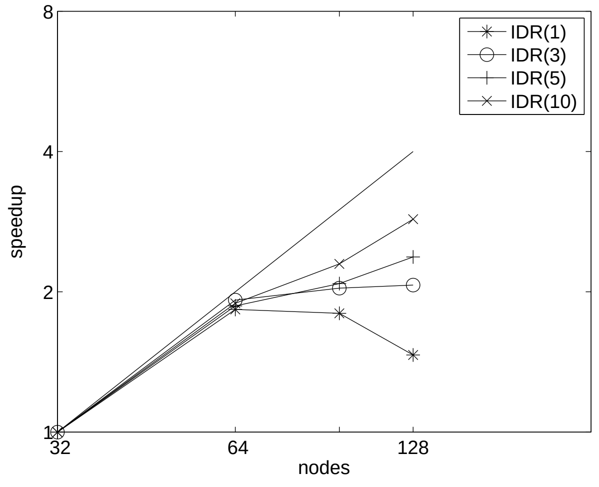
We will consider the following values for the convection parameter: $w = 180$, yielding a mesh-Péclet number of $Pe = 0.5$, $w = 360$, which gives $Pe = 1$, and $w = 1440$, which gives $Pe = 4$. We apply a-synchronous preconditioning for $T_{\max} = 0$s (i.e., no preconditioning), $T_{\max} = 5$s, $T_{\max} = 10$s, $T_{\max} = 15$s, and $T_{\max} = 20$s.

In realistic Grid computing environments, network load may vary extensively and can result in highly expensive global synchronisation. In order to simulate such an environment, network load will be varied artificially. Similar to [10], experiments will be performed on both a *lightly loaded* global network and on a *heavily loaded* global network. The results are displayed in Fig. 3.

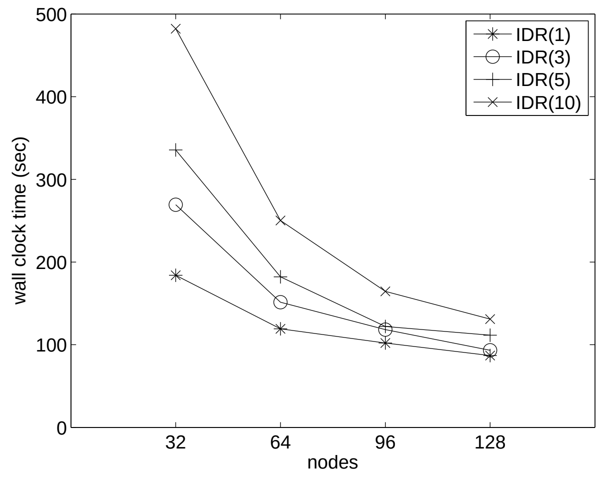
We can make the following observations:



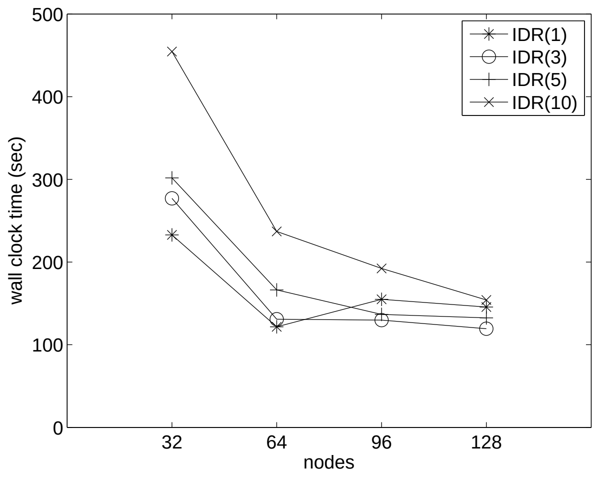
(a) Speed-up IDR(s) *minsync.*



(b) Speed-up IDR(s) *bi-ortho.*



(c) Total time IDR(s) *minsync.*



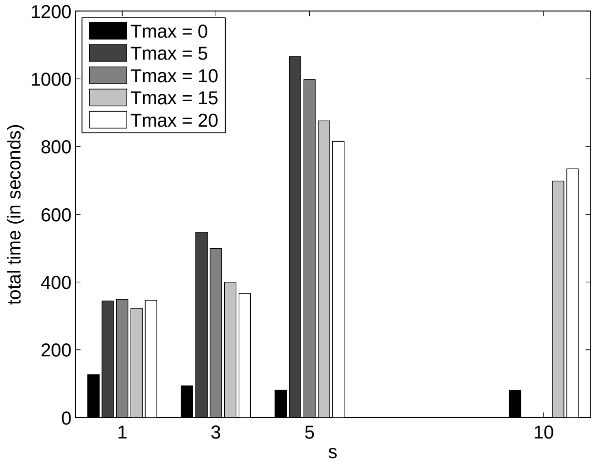
(d) Total time IDR(s) *bi-ortho.*

Figure 2: Speed-up on DAS-3, fixed problem size of $N = 256^3$ with $w = 100$.

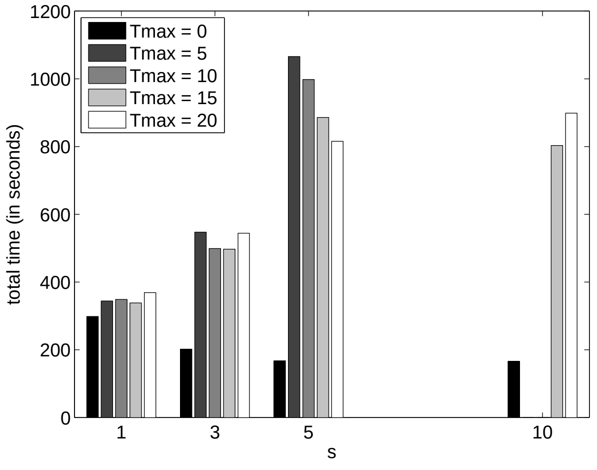
- In all experiments with preconditioning, after the convergence criterion was satisfied, the required accuracy was indeed achieved. This shows that the a-synchronous preconditioner can be used with IDR(s) without compromising the final accuracy.

- For the lower Péclet numbers $Pe = 0.5$ and $Pe = 1$, the unpreconditioned algorithm is fastest, but the unpreconditioned method does not converge for $Pe = 4$ for any of the values of s that we tested. With a-synchronous preconditioning IDR(s) converges for all experiments except for $Pe = 0.5, s = 10$ with $T_{\max} = 5$ or $T_{\max} = 10$. Moreover, the performance becomes better for increasing mesh-Péclet number.

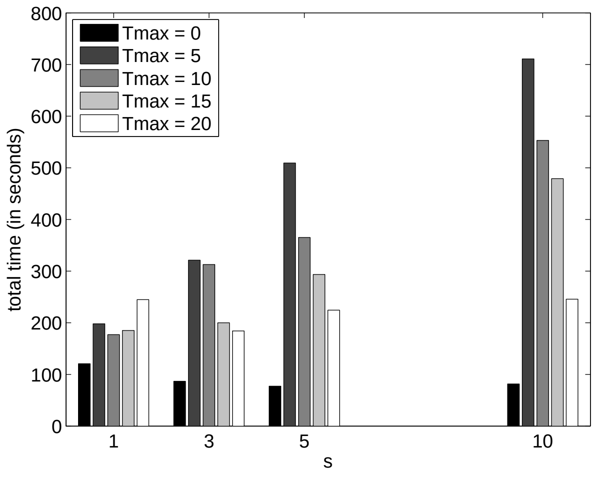
- The a-synchronous preconditioner is robust against changes in network load: little change in performance can be observed between the results with preconditioning for



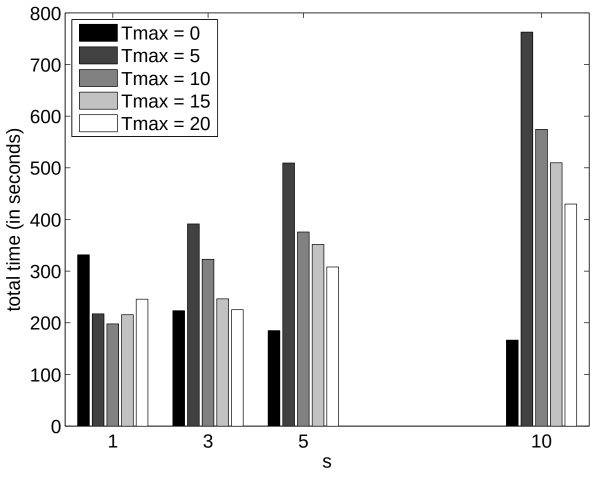
(a) $w = 180, Pe = 0.5$, lightly loaded network.



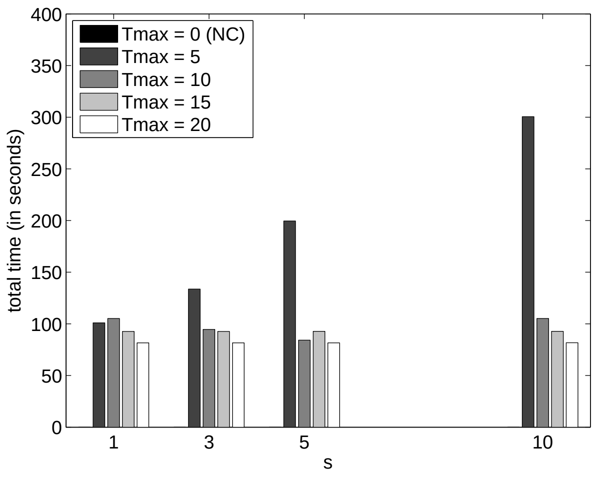
(b) $w = 180, Pe = 0.5$, heavily loaded network.



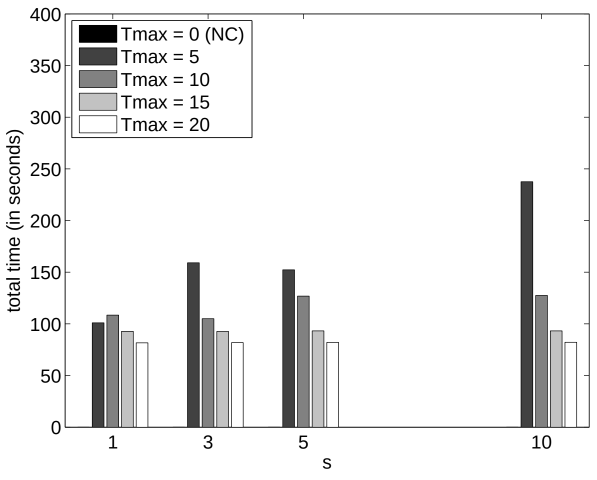
(c) $w = 360, Pe = 1.0$, lightly loaded network.



(d) $w = 360, Pe = 1.0$, heavily loaded network.



(e) $w = 1440, Pe = 4.0$, lightly loaded network.



(f) $w = 1440, Pe = 4.0$, heavily loaded network.

Figure 3: A-synchronous preconditioning, total computing time (NC denotes 'no convergence').

low and high network load. Synchronisation is more expensive if the network load is high. As a result, the computing times of unpreconditioned IDR(s) are importantly higher if the network load is high. The preconditioned method therefore performs relatively better in this case.

- IDR(s) without preconditioning performs better for higher s. With a-synchronous preconditioning, choosing a higher s negatively affects the convergence if $T_{\max}$ is chosen too small. In this case the preconditioner varies too much. This also explains the non-convergence in the cases metioned above ($Pe = 0.5, s = 10$ with $T_{\max} = 5$ or $T_{\max} = 10$). For higher $T_{\max}$, the variations in the preconditioner are smaller, and the theoretical properties of IDR(s) are less compromised.

4 Conclusions

We have discussed the combination of IDR(s) with an a-synchronous preconditioner in the context of grid computing. Experiments on convection-diffusion problems using a grid computer that consists of five geographically separated clusters show that this combination is particularly effective for high Péclet numbers. Moreover, the a-synchronous preconditioner makes the performance of the solution algorithm robust against variations in network load.

By using IDR(s) as a flexible method, some of its theoretical properties are lost. Because of this, choosing s high does not result in a faster convergence, as is normally the case with unpreconditioned IDR(s). By making the preconditioning step more accurate, the theoretical properties can in part be recovered. How accurate the preconditioning step should be, or more precisely, for how long an a-synchronous preconditioning step should be performed is at this moment still an open question.

Acknowledgments

The work of the second author was financially supported by the Delft Centre for Computational Science and Engineering (DCSE) within the framework of the DCSE project entitled *"Development of an Immersed Boundary Method, Implemented on Cluster and Grid Computers"*. The Netherlands Organisation for Scientific Research (NWO/NCF) is gratefully acknowledged for the use of the DAS-3 system.

References

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

[2] J. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid: three case studies. In *procs. of Vecpar 2004*, volume 3402 of *LNCS*, pages 302–314, Valencia, Spain, June 2004. Springer–Verlag.

[3] 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.

[4] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Coupling dynamic load balancing with asynchronism in iterative algorithms on the computational grid. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 40.1, Washington, DC, USA, 2003. IEEE Computer Society.

[5] 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.

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

[7] 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.

[8] Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *ICS '89: Proceedings of the 3rd international conference on Supercomputing*, pages 461–470, New York, NY, USA, 1989. ACM Press.

[9] Kostas Blathras, Daniel B. Szyld, and Yuan Shi. Timing models and local stopping criteria for asynchronous iterative algorithms. *Journal of Parallel and Distributed Computing*, 58(3):446–465, 1999.

[10] Tijmen P. Collignon and Martin B. van Gijzen. Fast iterative solution of large sparse linear systems on geographically separated clusters. Technical report, Delft University of Technology, Delft, the Netherlands, 2009. DUT report 09–12.

[11] Tijmen P. Collignon and Martin B. van Gijzen. Fast solution of nonsymmetric linear systems on Grid computers using parallel variants of IDR(s). Technical report, Delft University of Technology, Delft, the Netherlands, 2010. DUT report 10–05.

[12] Raphaël Couturier, Christophe Denis, and Fabienne Jézéquel. GREMLINS: a large sparse linear solver for grid environment. *Parallel Comput.*, 34:380–391, July 2008.

[13] 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.

[14] 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.

[15] Andreas Frommer, Hartmut Schwandt, and Daniel B. Szyld. Asynchronous weighted additive Schwarz methods. *Electronic Transactions on Numerical Analysis*, 5:48–61, 1997.

[16] Andreas Frommer and Daniel B. Szyld. Asynchronous two-stage iterative methods. *Numer. Math.*, 69(2):141–153, 1994.

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

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

[19] Martin H. Gutknecht. IDR Explained. *Electr. Trans. Numer. Anal.*, October 2009. (to appear).

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

[21] Yusuke Onoue, Seiji Fujino, and Norimasa Nakashima. Improved IDR(s) method for gaining very accurate solutions. *World Academy of Science, Engineering and Technology*, 55:520–525, 2009.

[22] Yusuke Onoue, Seiji Fujino, and Norimasa Nakashima. An overview of a family of new iterative methods based on IDR theorem and its estimation. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2009 Vol II IMECS 2009, March 18 - 20, 2009, Hong Kong*, pages 2129–2134, 2009.

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

[24] Valeria Simoncini and Daniel B. Szyld. Interpreting IDR as a Petrov–Galerkin method. Technical Report 09-10-22, Department of Mathematics, Temple University, October 2009.

[25] Gerard L. G. Sleijpen, Peter Sonneveld, and Martin B. van Gijzen. Bi–CGSTAB as an induced dimension reduction method. *Applied Numerical Mathematics*, In Press, Corrected Proof, 2009.

[26] Peter Sonneveld and Martin B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 31(2):1035–1062, 2008.

[27] H. A. van der Vorst. 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.

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

[29] Martin B. van Gijzen and Peter Sonneveld. An elegant IDR(s) variant that efficiently exploits bi–orthogonality properties. Technical report, Delft University of Technology, Delft, the Netherlands, 2008. DUT report 08–21.