Efficient Iterative Solution of Large Linear Systems on Heterogeneous Computing Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 1 april 2011 om 15:00 uur

door

Tijmen Pieter COLLIGNON
Master of Science Scientific Computing, Universiteit Utrecht

geboren te Haarlem

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. C. Vuik

Copromotor: Dr.ir. M.B. van Gijzen

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. C. Vuik,	Technische Universiteit Delft, promotor
Dr.ir. M.B. van Gijzen,	Technische Universiteit Delft, copromotor
Prof.dr. D. Boley,	University of Minnesota, USA
Prof.dr. D.B. Szyld,	Temple University, USA
Prof.dr.ir. B. Koren,	Universiteit Leiden
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft
Dr. G.L.G. Sleijpen,	Universiteit Utrecht
Prof.dr.ir. C.W. Oosterlee,	Technische Universiteit Delft, reservelid

Keywords: iterative methods, linear systems, preconditioning, asynchronous methods, parallel computing, Grid computing, high performance computing, IDR(s), flexible methods, domain decomposition, deflation

Efficient Iterative Solution of Large Linear Systems on Heterogeneous Computing Systems.
Dissertation at Delft University of Technology.
Copyright © 2011 by T. P. Collignon
Typeset in LaTeX

The work described in this thesis was financially supported by the Delft Centre for Computational Science and Engineering (DCSE).

ISBN 978-94-91211-15-7

Acknowledgments

This dissertation concludes four years of intense scientific research in the numerical analysis group at Delft University of Technology. It was an unforgettable experience and I learned many valuable lessons, both personally and professionally. I would not have been able to finish such a large body of work without the help of many people, who I want to thank here.

First of all, I want to mention that this PhD research was financially supported by the Delft Centre for Computational Science and Engineering (DCSE). Also, 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*", which 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.

My most sincere gratitude goes to my daily supervisor and copromotor Martin van Gijzen. His endless enthusiasm and constructive criticism made working with him a real pleasure. Martin always managed to provide the perfect balance between freedom and supervision. If I am a better scientist than before, it is largely due to Martin's influence. I could truly not have asked for a better supervisor.

The other person that I am greatly indebted to is my promotor prof.dr. Kees Vuik. He provided invaluable feedback during the many meetings that we had. It was a true privilege being able to work in his numerical analysis group.

I also thoroughly enjoyed working with dr. Gerard Sleijpen. He truly knows everything about everything and I loved picking his brain for inspiration and knowledge. I also immensely enjoyed our time together in Gifu and Tokyo with Yoshi and Emiko.

I was fortunate enough to have numerous inspiring conversations with Scott MacLachlan, Jok Tang, and Peter Sonneveld. Their immense body of knowledge has helped me improve my understanding of difficult mathematical material in many ways.

I want to thank my colleagues Barry Koren and Yunus Hassen from CWI for many constructive meetings during the beginning of my PhD research. I only regret that our research paths deviated somewhat towards the end. That is typical of being on the frontier of science, where it is always hard to predict how things will go.

I thank from the bottom of my heart prof. Kuniyoshi Abe from Gifu Shotoku University and prof. Emiko Ishiwata from Tokyo University of Science for being such excellent hosts during my short but intense stay in Japan. I want to mention in particular some of Emiko's students: Akiko, Atsushi, Masaki, Kensuke, Tsubasa, Dai, Muran, and Hama–chan. The hospitality of all these people extended far beyond the confines of the university walls and I had the privilege to see numerous interesting things, not only in Gifu and in Tokyo but also in other parts of Japan.

I had the good opportunity to attend several international conferences in Poland, Switzerland, France, Sweden, and Belgium. I thank my supervisors for giving me this chance to expand my academic horizon. I especially want to mention the annual Woudschoten conferences in Zeist, which were always a real pleasure attending.

Without the invaluable assistance of the secretaries Diana Droog, Mechteld de Boer-van Doorninck, Daniëlle Docter, and Deborah Dongor, my academic life much would have been much more difficult. And whenever our own secretary was not available, Evelyn Sharabi, Dorothée Engering, and Cindy Bosman were always there to fill in.

Kees Lemmens and Xiwei Wu were invaluable in providing and maintaining the excellent computing facilities. I thank them for their continuous support.

It was a pleasure organising the excursion to Deltares in 2009 together with Peter Lucas and Wim van Balen.

I want to thank Dick Epema and his (former) PhD students — in particular Alexandru Iosup — from the Parallel and Distributed Systems group for the interesting discussions I had in the many “Grid meetings”.

I would like to thank the GridSolve team for their prompt response pertaining to my questions and also Stéphane Domas for his prompt and extensive responses pertaining to the CRAC programming system. I 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 1.6 is based on a figure kindly donated by Xu Lin and Paulo Anita kindly provided information on the communication patterns induced by the algorithm on the DAS–3 cluster. Paulo als helped me whenever a DAS–3 node gave me troubles.

I would also like to thank Rob Bisseling for careful proof–reading of the manuscript that Chapter 2 was based on. Also, I thank the editors and anonymous referees for their constructive criticism of the manuscripts, which considerably improved the presentation of each final paper.

On a more personal note, I thank all of the (former and current) people of the numerical analysis group for providing such a pleasant working environment. First of all, the PhD researchers Sander van Veldhuizen, John Brussche, Reijer Idema, Jok Tang, Fang Fang, Bowen Zhang, Liangyue Ji, and Paulien van Slingerland. Also, the members of the (more) permanent staff Peter Sonneveld, Fons Daalderop, Jos van Kan, Fred Vermolen, Duncan van der Heul, Domenico Lahaye, Guus Segal, Kees Oosterlee, Jennifer Ryan, and Sergey Zemskov. I would especially like to thank Liangyue Ji for introducing me to Chinese cooking and to Lao Gan Ma.

I want to mention in particular the people that I had the good fortune to share an office with. In the beginning these were Sander van Veldhuizen and John Brussche, and later on Reijer Idema and Pavel Prokharau. I thank them all for providing such a pleasant and effective distraction from work. I can only hope that my future group of colleagues will be as pleasant as all these people.

The daily lunch breaks with my colleagues were always a real pleasure, and I want to mention in particular Miriam ter Brake and Jelle Hijmissen from the Mathematical Physics department.

I want to name a few of my fellow participants and friends (in no particular order) of the PhDays weekends, which I attended and enjoyed immensely four years in a row: Arthur van Dam, Albert–Jan Yzelman, Jeroen Wackers, Joris Vanbiervliet, Ricardo Jorge Reis Silva, Peter In ’t Panhuis, Liesbeth Vanherpe, Katrijn Frederix, Bart Vandereycken, Samuel Corveleyn, Virginie De Witte, Charlotte Sonck, Tinne Haentjens, Kim Volders, Eveline Rosseel, Yvette Vanberghen, Ward Van Aerschot, Jeroen de Vlieger, and Yves Frederix.

My greatest debts are to my family. My father continues to encourage me, as does the memory of my late mother. The support and affection of Congli have made this project an odyssey and not a marathon. My brothers and sister are my greatest friends and I dedicate this dissertation to them.

Tijmen Collignon
Delft, November 2010

Summary

Efficient Iterative Solution of Large Linear Systems on Heterogeneous Computing Systems

Tijmen P. Collignon

This dissertation deals mainly with the design, implementation, and analysis of efficient iterative solution methods for large sparse linear systems on distributed and heterogeneous computing systems as found in Grid computing.

First, a case study is performed on iteratively solving large symmetric linear systems on both a multi–cluster and a local heterogeneous cluster using standard block Jacobi preconditioning within the software constraints of standard Grid middleware and within the algorithmic constraints of preconditioned Conjugate Gradient–type methods. This shows that global synchronisation is a key bottleneck operation and in order to alleviate this bottleneck, three main strategies are proposed: exploiting the hierarchical structure of multi–clusters, using asynchronous iterative methods as preconditioners, and minimising the number of inner products in Krylov subspace methods.

Asynchronous iterative methods have never really been successfully applied to the solution of extremely large sparse linear systems. 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 property in heterogeneous computing systems. Krylov subspace methods offer significantly improved convergence rates, but 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 flexible Krylov subspace method, the best of both worlds is combined. It is shown that this hybrid combination of a slow but asynchronous inner iteration and a fast but synchronous outer iteration results in high convergence rates on heterogeneous networks of computers. Since the preconditionering iteration is performed on heterogeneous computing hardware, it varies in each iteration step. Therefore, a flexible iterative method which can handle a varying preconditioner has to be employed. This partially asynchronous algorithm is implemented on two different types of Grid hardware applied to two different applications using two different types of Grid middleware.

The IDR(s) method and its variants are new and powerful algorithms for iteratively solving large nonsymmetric linear systems. Four techniques are used to construct an efficient IDR(s) variant for parallel computing and in particular for Grid computing. Firstly, an efficient and robust IDR(s) variant is presented that has a single global synchronisation point per matrix–vector multiplication step. Secondly, the so–called IDR test matrix in IDR(s) can be chosen freely and this matrix is constructed such that the work, communication, and storage involving this matrix are minimised in the context of multi–clusters. Thirdly, a methodology is presented for *a priori* estimation of the optimal value of s in IDR(s). Finally, the proposed IDR(s) variant

is combined with an asynchronous preconditioning iteration.

By using an asynchronous preconditioner in IDR(s), the IDR(s) method is treated as a *flexible* method, where the preconditioner changes in each iteration step. In order to begin analysing mathematically the effect of a varying preconditioning operator on the convergence properties of IDR(s), the IDR(s) method is interpreted as a special type of *deflation* method. This leads to a better understanding of the core structure of IDR(s) methods. In particular, it provides an intuitive explanation for the excellent convergence properties of IDR(s).

Two applications from computational fluid dynamics are considered: large bubbly flow problems and large (more general) convection–diffussion problems, both in 2D and 3D. However, the techniques presented can be applied to a wide range of scientific applications.

Large numerical experiments are performed on two heterogeneous computing platforms: (i) local networks of non–dedicated computers and (ii) a dedicated cluster of clusters linked by a high–speed network. The numerical experiments not only demonstrate the effectiveness of the techniques, but they also illustrate the theoretical results.

Samenvatting

Efficiënte iteratieve oplosmethoden voor grote lineaire systemen op gedistribueerde heterogene computersystemen

Tijmen P. Collignon

Deze dissertatie gaat over het ontwerpen, implementeren en analyseren van efficiënte iteratieve oplosmethoden voor grote lineaire en ijle systemen op gedistribueerde heterogene computersystemen zoals bijvoorbeeld in Grid computing.

Als eerste is er een *casestudy* uitgevoerd waarbij iteratief grote symmetrische lineaire systemen worden opgelost op zowel een globaal multicluster als een lokaal heterogeen cluster, gebruikmakend van standaard blok Jacobi preconditionering. Hierbij is geprobeerd de limieten van gestandaardiseerd Grid middleware en de limieten van de geconjugeerde gradiënten methode zoveel mogelijk op te rekken. Deze *casestudy* laat ondere andere zien dat globale synchronisatie één van de grootste kritieke punten is. Om dit te verhelpen, worden er drie oplossingen voorgesteld: het gebruikmaken van de hiërarchische structuur van multiclusters, het gebruikmaken van asynchrone methoden en het minimalizeren van het aantal inprodukten in Krylov methoden.

Asynchrone methoden zijn nooit echt populair geweest voor het iteratief oplossen van extreem grote lineaire systemen. De hoofdreden is dat deze methoden over het algemeen traag convergeren. Desalniettemin is het ontbreken van globale synchronisatiepunten in zulke methoden een zeer gewaardeerde eigenschap in de context van heterogene computersystemen. Aan de andere kant bieden Krylov methoden een aanzienlijke snelheidwinst, maar de inherente globale synchronisatiepunten van zulke methoden maakt het lastig deze effectief toe te passen. Door een asynchrone methode te gebruiken als preconditioneerder in een zogenaamde flexibele Krylov methode wordt het beste van twee werelden gecombineerd. Het blijkt dat de hybride combinatie van een trage maar asynchrone binneniteratie en een snelle maar synchrone buiteniteratie leidt tot een hoge convergentiesnelheden op heterogene netwerken van computers. Aangezien de preconditionering onder andere wordt uitgevoerd op heterogene computerhardware, varieert deze in iedere iteratiestap. Daarom is het noodzakelijk om een flexibele iterative methode te gebruiken die een variabele preconditionering aankan. Deze gedeeltelijk asynchrone methode is geïmplementeerd op twee verschillende soorts Grid computers, gebruikmakend van twee verschillende Grid middleware en met twee verschillende soorten toepassingen.

De IDR(s) methode en zijn varianten zijn krachtige algoritmes voor het iteratief oplossen van grote niet-symmetrische lineaire systemen. In deze dissertatie zijn vier technieken gebruikt om een efficiente IDR(s) variant te construeren voor parallelle computers en dan met name voor Grid computers. Als eerst wordt er een efficiënte en robuuste IDR(s) variant voorgesteld met één globaal synchronisatiepunt. Als tweede wordt de zogenaamde testmatrix in IDR(s) zo gekozen dat het rekenwerk, communicatie en opslag behorende bij operaties met deze matrix zoveel mogelijk worden geminimaliseerd in de context van multiclusters. Als derde wordt een

techniek gepresenteerd voor het *a priori* schatten van de optimale waarde voor s in IDR(s). Ten slotte wordt de voorgestelde IDR(s) variant gecombineerd met een asynchrone preconditionering.

Het feit dat een asynchrone methode als preconditioneerder wordt gebruikt in IDR(s), betekent dat IDR(s) als een flexibele methode wordt behandeld. Hierbij wordt de preconditionering in iedere iteratiestap aangepast. Om een begin te maken met het analyseren van het effect van een variabele preconditionering op de convergentie van IDR(s), wordt de IDR(s) methode geïnterpreteerd als een speciaal soort deflatiemethode. Dit leidt tot een beter begrip van de structuur van IDR(s) methoden. Daarnaast geeft dit een idee waarom IDR(s) methoden vaak zo goed convergeren.

Twee belangrijke toepassingen uit de numerieke stromingsleer zijn bekeken: stromingsproblemen met bellen en (meer algemene) convectie–diffusie problemen. Beiden worden in 2D en 3D behandeld. Echter, de technieken die worden gepresenteerd kunnen op uiteenlopende problemen worden toegepast.

Grootschalige numerieke experimenten zijn uitgevoerd op twee heterogene computerplatformen: lokale netwerken bestaande uit standaard PC's en een groot cluster van meerdere clusters verbonden door een supersnel netwerk. Deze experimenten hebben niet alleen als doel de effectiviteit van de voorgestelde technieken te demonstreren, maar ook om de theoretische resultaten te illustreren.

Contents

Acknowledgments iii

Summary v

Samenvatting vii

Contents ix

List of Figures xiii

List of Tables xv

1 Solving Large Linear Systems on Grid Computers 1
Part I: Efficient iterative methods in Grid computing . . . 2
1.1 Introduction . . . 2
1.2 The problem . . . 4
1.3 Iterative methods . . . 6
1.3.1 Simple iterations . . . 6
1.3.2 Impatient processors: asynchronism . . . 7
1.3.3 Acceleration: subspace methods . . . 9
1.4 Hybrid methods: best of both worlds . . . 11
1.5 Some experimental results . . . 14
Part II: Efficient implementation in Grid computing . . . 14
1.6 Introduction . . . 14
1.7 Grid middleware . . . 15
1.7.1 Description of GridSolve . . . 16
1.7.2 Description of CRAC . . . 18
1.7.3 MPI–based libraries . . . 19
1.8 Target hardware . . . 19
1.8.1 Local heterogeneous clusters . . . 19
1.8.2 DAS–3: “five clusters, one system” . . . 21
1.9 Coupled vs. decoupled inner–outer iterations . . . 21
1.10 Parallel iterative methods: building blocks revisited . . . 22
1.10.1 Matrix–vector multiplication (or data distribution) . . . 22
1.10.2 Preconditioning . . . 23
1.10.3 Convergence detection . . . 23
1.10.4 Vector operations . . . 24
1.11 Applications . . . 24

1.12 Further reading . . . 25
Part III: General thesis remarks . . . 25
1.13 Related work and main contributions . . . 25
1.14 Scope and outline . . . 26
1.15 Notational conventions and nomenclature . . . 29

2 Conjugate Gradient Methods for Grid Computing **31**
2.1 Introduction . . . 32
2.2 Heterogeneous sparse linear solvers in GridSolve . . . 32
2.2.1 Motivation . . . 32
2.2.2 Resource–aware load balancing . . . 33
2.2.3 Partitioning algorithm and CG schemes . . . 34
2.2.4 Implementation details . . . 36
2.3 Numerical experiments . . . 37
2.3.1 Overview . . . 37
2.3.2 Target hardware . . . 38
2.3.3 Preliminary testing . . . 38
2.3.4 Heterogeneous environment . . . 40
2.3.5 Parallel performance . . . 41
2.4 Concluding remarks . . . 42
2.4.1 Conclusions . . . 42
2.4.2 Suggestions for improvements . . . 43
2.4.3 General remarks . . . 43

3 Asynchronous Iterative Methods as Preconditioners **45**
Part I: Coupled iterations . . . 46
3.1 Introduction . . . 46
3.2 Parallel implementation details . . . 47
3.2.1 Asynchronous preconditioning . . . 47
3.2.2 Orthogonalisation . . . 48
3.3 Numerical experiments . . . 49
3.3.1 Motivation . . . 49
3.3.2 Target hardware and experimental setup . . . 50
3.3.3 Experimental results . . . 51
3.3.4 Discussion . . . 54
Part II: Decoupled iterations . . . 55
3.4 Introduction . . . 55
3.5 Parallel implementation details . . . 56
3.5.1 Brief description of GridSolve . . . 56
3.5.2 Decoupled iterations . . . 56
3.6 Numerical experiments . . . 58
3.6.1 Target hardware and experimental setup . . . 58
3.6.2 Experimental results and discussion . . . 59
Part III: Deflation and smoothing . . . 61
3.7 Motivation . . . 61
3.8 Deflation methods . . . 62
3.9 Numerical experiments . . . 63
3.10 Using asynchronous iterative methods as smoothers . . . 65
3.11 Conclusions . . . 67

4 IDR(s) for Grid Computing **69**
4.1 Introduction . . . 70
4.2 IDR(s) variant with one synchronisation point per MV . . . 71
4.3 Parallelising IDR(s) methods . . . 79
4.4 Choosing s and $\widetilde{\mathbf{R}}$ . . . 79
4.4.1 Parallel performance models for IDR(s) . . . 80
4.4.2 Using piecewise sparse column vectors for $\widetilde{\mathbf{R}}$ . . . 82
4.5 Combining IDR(s) with asynchronous preconditioning . . . 83
4.6 Numerical experiments . . . 84
4.6.1 Target hardware . . . 85
4.6.2 Test problem . . . 85
4.6.3 Estimating parameters of performance model . . . 87
4.6.4 *A priori* estimation of optimal parameter s . . . 88
4.6.5 Validation of the parallel performance model . . . 89
4.6.6 Comparing the time per IDR(s) cycle to the performance model . . . 90
4.6.7 Parallel speedup results . . . 91
4.6.8 Results for IDR(s) with asynchronous preconditioning . . . 93
4.7 Conclusions . . . 94

5 IDR(s) as a Deflation Method **97**
5.1 Introduction . . . 98
5.2 Relation between IDR and deflation . . . 99
5.2.1 IDR methods . . . 100
5.2.2 Deflation methods . . . 101
5.2.3 Interpreting IDR(s) as a deflation method . . . 103
5.2.4 IDR algorithms . . . 107
5.3 The IDR projection theorem . . . 110
5.3.1 Single IDR(s) cycle . . . 110
5.3.2 Main result: IDR projection theorem . . . 111
5.3.3 Numerical examples . . . 112
5.3.4 Discussion . . . 113
5.4 Explicitly deflated IDR(s) . . . 114
5.4.1 Deflation vs. augmentation . . . 115
5.4.2 Choosing $\widetilde{\mathbf{R}}$ and ω_k . . . 120
5.4.3 Numerical examples . . . 121
5.5 Conclusions . . . 122

6 General Conclusions and Outlook **123**
6.1 Aim of research . . . 123
6.2 Research challenges and principal findings . . . 124
6.3 Broader implications . . . 125

Curriculum Vitæ **127**

Scientific Résumé **129**

Bibliography **133**

Index **145**

List of Figures

1.1 Depiction of the oceans of the world, divided into two computational subdomains. 6
1.2 Time line of a certain type of asynchronous algorithm, showing three Jacobi iteration processes. 8
1.3 Three–stage iterative method: GMRESR—asynchronous block Jacobi—IDR(s). 12
1.4 Experimental results asynchronous preconditioning. 14
1.5 Schematic overview of GridSolve. The dashed line represents (geographical) distance between the client and servers. 17
1.6 The DAS–3 multi–cluster. 20

2.1 Heterogeneous block–row partitioning for $s = 4$ and $k = 8$ for a 2D Poisson problem. 35
2.2 Wall clock times of CG implementations in GridSolve on the local cluster. 39
2.3 Breakdown of wall clock time of tasks in communication (bottom part) and computation, for $n = 4 \times 10^6$ and using seven servers. 39
2.4 Heterogeneous experiments with the 3D bubbly flow problem (local cluster). 41
2.5 Comparison of two preconditioning techniques (local cluster). 42
2.6 Comparison of two preconditioning techniques (DAS–3). 42

3.1 Total execution time (3D problem). 51
3.2 Relative increase of time per outer iteration step (3D problem). 52
3.3 Total execution time (2D problem). 54
3.4 Relative increase of time per outer iteration step (2D problem). 55
3.5 Experiments on a large heterogeneous cluster with 250,000 equations per server. 59
3.6 Jacobi sweeps performed by each server during outer iteration steps. 60
3.7 Point smoothers. 65
3.8 Smoothing for block Jacobi and for asynchronous iterations (A-BJ). 66

4.1 Varying preconditioner in IDR(s), shown for a single IDR cycle. 83
4.2 Estimating model parameters: $\widehat{n}$ and processor speed. 87
4.3 Performance model results for variants (i) and (ii) using 32 nodes per cluster. 90
4.4 Investigating s–dependence for $s \in \{1, \ldots, 16\}$ using 64 nodes, four sites, and fast network. 91
4.5 Strong scalability results, scaled to number of iterations, $n = 128^3$. 91
4.6 Strong scalability results, scaled to number of iterations, $n = 256^3$. 92
4.7 Weak scaling experiments on the DAS–3. 93
4.8 A-synchronous preconditioning, total computing time (NC denotes 'no convergence'). 95

5.1 Left: solve $\mathbf{AB}^{-1}\mathbf{y} = \mathbf{x}$. Middle: premultiply $\mathbf{y}$ and $\mathbf{u}$ by $\mathbf{B}^{-1}$. Right: substitute $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$. . . . 106
5.2 $\mathbf{B} = \mathbf{I}, n = 20, s = 5$, four cycles in total, $\mu_k \in \{0\} \cup \{2.9, 2.5, 2.2\}$ for $k = 0, 1, 2, 3$. 113
5.3 $\mathbf{B} = \mathbf{I}, n = 20, s = 5$, four cycles in total, $\mu_k \in \{0\} \cup \{1\}$ for all k. . . . 114
5.4 Residual norms for IDR(s), $n = 25, s = 5, \mathbf{B} = \mathbf{I}$, showing primary "$*$" and secondary "$\circ$" residuals. . . . 121

List of Tables

1.1 Parallel and distributed computing on cluster and Grid hardware. 4
1.2 Main difficulties and possible solutions associated with designing efficient numerical algorithms in Grid computing. 5
1.3 Several characteristics pertaining to three different types of (Grid) middleware. . 15
1.4 DAS–3: “five clusters, one system”. 20
1.5 Average roundtrip measurements (in ms) between several DAS–3 sites, with exception of the TUD site. 21
1.6 Overview of the iterative methods and middleware used in this thesis. 27
1.7 Notational conventions and nomenclature. Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ and $\mathbf{v} \in \mathbb{C}^{n}$. 29

3.1 Differences in computational setting. 46
3.2 Influence of parameter m ($T_{\max} = 5$s, five nodes, 2D problem). 50
3.3 Outer iterations for synchronous and asynchronous preconditioning (3D problem). 53
3.4 Outer iterations for synchronous and asynchronous preconditioning (2D problem). 53
3.5 Number of outer iterations (3D problem). 64

4.1 Specifications DAS–3: all values (except WAN latencies) are obtained from [16]. 85
4.2 Processor grids and problem size for the strong scalability experiments. 86
4.3 Processor grids and problem sizes for the weak scalability experiments. 86
4.4 *A priori* estimation of s for the LU cluster. 88
4.5 *A priori* estimation of s for the DAS–3 multi–cluster. 89
4.6 Total number of iterations. 93

5.1 Extra computational cost per MV compared to “standard” IDR(s). 119

List of Algorithms

1.1 (A–)synchronous block Jacobi iteration without overlap for solving $\mathbf{Ax} = \mathbf{b}$ on p processors. 8
1.2 The preconditioned Conjugate Gradient method [81]. 10
2.1 Resource–aware Preconditioned Conjugate Gradient Algorithm; s servers 34
2.2 Preconditioned CG; Task i with three subtasks. 36
2.3 Preconditioned CG; Chronopoulos and Gear variant; Task i. 37
3.1 Flexible Conjugate Gradients (pure truncation strategy) 47
3.2 Asynchronous block Jacobi iteration for CRAC task i. 48
3.3 Modified Gram–Schmidt. 49
3.4 GMRESR (truncated version) 56
3.5 Classical Gram–Schmidt 57
3.6 Asynchronous block Jacobi iteration task for each GridSolve server i. 57
3.7 Computation of $\mathbf{u} = \mathbf{P}^{\text{adef2}}\mathbf{r}_k$ 63
3.8 Computation of $\widehat{\Pi}\mathbf{y}$ 63
4.1 IDR(s)-biortho with bi–orthogonalisation of intermediate residuals. 73
4.2 IDR(s)-minsync with bi–orthogonalisation of intermediate residuals and with a single synchronisation point per MV. 75
5.1 IDR(s) as a deflation method: generating a sequence of subspaces of $\mathcal{G}_k$. 104
5.2 IDR(s) as a deflation method: a “traditional” listing. 108
5.3 IDR(s) as a deflation method: the IDR(s)-biortho variant from Algorithm 4.1. . 109
5.4 Computation of $\Pi^{\text{idr}'}\mathbf{y}$ 119
5.5 Computation of $\Pi^{\text{idr}'}\mathbf{y}$ with special choice for $\widetilde{\mathbf{R}}$. 120

Chapter 1

Solving Large Linear Systems on Grid Computers

This chapter has appeared as:

T. P. Collignon and M. B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. In B. Koren and C. Vuik, editors, *Advanced Computational Methods in Science and Engineering. Springer Series Lecture Notes in Computational Science and Engineering*, volume 71, pages 79–106. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

Overview

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. The design of efficient numerical algorithms is therefore a complex process that brings together many different scientific disciplines. This introductory chapter of the thesis is divided into three parts and whenever appropriate specific references to other chapters are given.

The purpose of the first part (Section 1.1–1.5 starting on page 2) is to give a bird's eye view of the issues pertaining to designing efficient numerical algorithms for *Grid computing* and is aimed at a general audience. The first part starts in Section 1.1 by giving a general introduction to parallel computing and in particular to Grid computing. The discussion continues in Section 1.2 by clearly stating the problem and exposing the various bottlenecks, subsequently followed by the presentation of potential solutions. Thus, the stage is set and Section 1.3 proceeds by detailing *classical* iterative solution methods for linear systems, along with the concept of *asynchronous* iterative methods. Although these type of methods exhibit features that are extremely well–suited for Grid computing, they can also suffer from slow convergence speeds. In Section 1.4 it is explained how asynchronous methods can be combined with faster but more expensive *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 first part is wrapped up in Section 1.5 by presenting some illustrative experimental results using this partially asynchronous approach.

The second part (Section 1.6–1.12 starting on page 14) deals with more advanced topics and contains discussions on the various intricacies related to *efficiently implementing* the proposed algorithms on Grid computers. After giving some general remarks in Section 1.6, a description is given in Section 1.7 of the three classes of so–called Grid middleware used in this thesis. Section 1.8 discusses the two types of target hardware that are considered. Then, each building block of a subspace method as discussed in the first part is revisited in Section 1.10 and put in the context of parallelisation on Grid computers. The exposition of the second part is concluded in Section 1.12 by giving suggestions for further reading.

In the last part (Section 1.13–1.15 starting on page 25) general statements concerning this thesis are given. Section 1.13 discusses related work and gives the main contributions of this thesis. In Section 1.14 the scope and outline of the thesis are given and in Section 1.15 the notational conventions used in this thesis are listed.

Part I: Efficient iterative methods in Grid computing

1.1 Introduction

Solving extremely large sparse linear systems of equations is the computational bottleneck in a wide range of scientific and engineering applications. Examples include simulation of air flow around wind turbine blades, weather prediction, option pricing, and Internet 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 super-

computers, 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 non–dedicated 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 [121]. 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 [132].

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 [5].

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 [65].

The fact that in Grid computing resources are geographically separated implies that *communication* is less efficient compared to more tightly–coupled 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 unavoidable 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 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 computing* [84] and *GPU computing* [105]. Modern gaming consoles and graphics cards employ dedicated high–performance processors for compute–intensive tasks, such as rendering high–resolution 3D images. In combination with their inherent parallel design and cheap manufacturing process, this makes them extremely appropriate for parallel scientific computing, in particular for problems with high data parallelism [147]. 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 [64]. For a recent paper that use results from this project, see [99].

Nowadays, multi–core desktop computers with eight or more computing cores are becoming increasingly mainstream. Many existing software products such as audio 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 [1].

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

Table 1.1: Parallel and distributed computing on cluster and Grid hardware.

1.2 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.

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 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 be paid is that of synchronisation.

Table 1.1 lists some of the classifications that may be associated with cluster and Grid computing, respectively. In reality, things are never as clear–cut as this table might suggest. For example, a cluster of homogeneous and dedicated clusters connected by a network could be considered a Grid computer. Vice versa, a local cluster may consist of computers that have varying workloads, 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 Table 1.2 the main problems are listed, along with possible solutions. Clearly there are many aspects that need to be addressed, requiring substantial expertise from a broad spectrum of scientific disciplines.

When designing numerical algorithms for general applications, a proper balance should be found between *robustness* (predictable performance using few parameters) and *efficiency* (optimal scalability, both algorithmic and parallel). At the risk of trivialising these two important

Difficulties and challenges	Possible solution(s)
– **Global synchronisation.** In most operations that require global synchronisation, the data that is being exchanged is relatively small compared to the communication overhead, which makes this extremely expensive 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.** 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. – **Minimal synchronisation.** Many iterative algorithms can be modified in such a manner that the number of synchronisation points is reduced. These modifications include rearrangement of operations [37], truncation strategies [50], and the type of reorthogonalisation procedure [48].
– **Heterogeneity.** Resources from many different sources may be combined, potentially resulting in a highly heterogeneous environment. This can apply to things like 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 [132].
– **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.	– **Adaptive.** 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 [132]. – **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.

Table 1.2: Main difficulties and possible solutions associated with designing efficient numerical algorithms in Grid computing.

issues, the ultimate numerical algorithm wish list for Grid computing contains the following additional items: *coarse–grained*, *asynchronous*, *minimal synchronisation*, *resource–aware*, *adaptive*, and *fault–tolerant*. The ultimate challenge is to devise an algorithm that exhibits all of these eight features. It is shown at the end of this chapter in Section 1.14 that the algorithms presented in this thesis exhibit all of these features in some way or another.

Parallel scalability can be investigated in both the strong and weak sense. In strong scalability experiments a fixed *total* problem size is used, while in weak scalability experiments a fixed problem size *per node* is used. In this thesis, the parallel algorithms will be investigated in both senses.

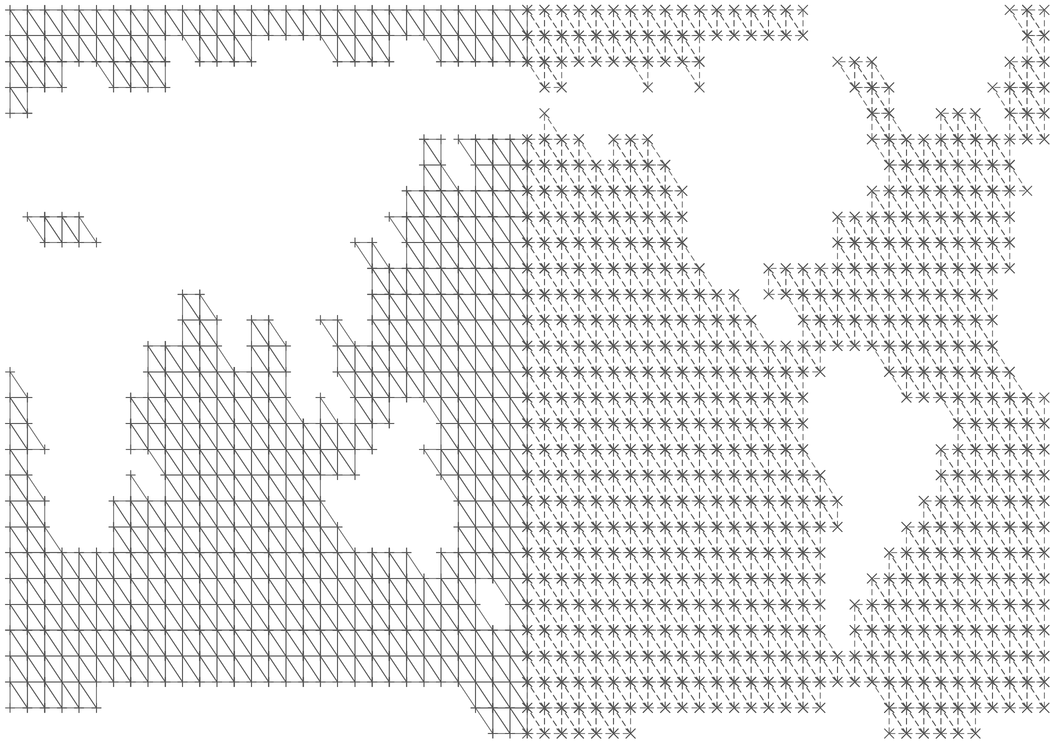


Figure 1.1: Depiction of the oceans of the world, divided into two computational subdomains.

1.3 Iterative methods

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

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.1}$$

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

1.3.1 Simple iterations

Given an initial solution $\mathbf{x}^{(0)}$, the classical iteration for solving the system (1.1) is

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{B}^{-1}\left(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}\right), \qquad k = 0, 1, \ldots, \tag{1.2}$$

where $\mathbf{B}^{-1}$ serves as an approximation for $\mathbf{A}^{-1}$. For practical reasons, solving systems involving the matrix $\mathbf{B}$ should be cheap and this is reflected in the different choices for $\mathbf{B}$. The simplest option would be to choose the identity matrix for $\mathbf{B}$, which results in the *Richardson iteration*, e.g., [102]. Another variant is the *Jacobi iteration*, which is obtained by taking for $\mathbf{B}$ the diagonal matrix having entries from the diagonal of $\mathbf{A}$. Choices that in some sense better approximate the matrix $\mathbf{A}$ tend to result in methods that converge to the solution in less iterations. However,

inverting the matrix $\mathbf{B}$ will be more expensive and it is clear that some form of trade–off is necessary.

The iteration (1.2) can be generalised to a block version, which results in an algorithm closely related to *domain decomposition* techniques [118]. 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 Figure 1.1. 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 coefficient matrix $\mathbf{A}$, the solution vector $\mathbf{x}$, and the right–hand side $\mathbf{b}$ are partitioned into non–overlapping blocks as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}. \tag{1.3}$$

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

The block Jacobi iteration generalises the Jacobi iteration by taking for $\mathbf{B}$ the block diagonal elements, giving

$$\mathbf{B} = \begin{bmatrix} \mathbf{A}_{11} & \emptyset \\ \emptyset & \mathbf{A}_{22} \end{bmatrix}. \tag{1.4}$$

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

$$\begin{cases} \mathbf{x}_1^{(k+1)} = \mathbf{x}_1^{(k)} + \mathbf{A}_{11}^{-1} \left(\mathbf{b}_1 - \mathbf{A}_{11}\mathbf{x}_1^{(k)} - \mathbf{A}_{12}\mathbf{x}_2^{(k)} \right); \\ \mathbf{x}_2^{(k+1)} = \mathbf{x}_2^{(k)} + \mathbf{A}_{22}^{-1} \left(\mathbf{b}_2 - \mathbf{A}_{21}\mathbf{x}_1^{(k)} - \mathbf{A}_{22}\mathbf{x}_2^{(k)} \right), \end{cases} \quad k = 0, 1, \ldots. \tag{1.5}$$

On a parallel computer, each complete subdomain can be mapped to a single processor and these iterations may be performed independently for each iteration step k. This is followed by a synchronisation point where information is exchanged between the processors. This so–called *synchronous* variant of a block Jacobi iteration is shown in line 4 of Algorithm 1.1 for the general case of p processors and/or subdomains. Here, $\mathbf{A}, \mathbf{x}$, and $\mathbf{b}$ are partitioned into non–overlapping blocks as follows:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1p} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{p1} & \mathbf{A}_{p2} & \cdots & \mathbf{A}_{pp} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \end{bmatrix}. \tag{1.6}$$

An extra complication is that the block matrices $\mathbf{A}_{ii}$ located on the diagonal need to be inverted. In practical implementations, these inner systems are often solved approximately by some other iterative method. In this case, how accurately these systems should be solved becomes an important issue.

1.3.2 Impatient processors: asynchronism

Asynchronous algorithms *generalise* simple iterative methods such as the classical block Jacobi iteration and line 5 of Algorithm 1.1 shows an asynchronous block Jacobi iteration. Instead of

Algorithm 1.1 (A–)synchronous block Jacobi iteration without overlap for solving $\mathbf{Ax} = \mathbf{b}$ on p processors.

OUTPUT: Approximation $\mathbf{x}$ to $\mathbf{Ax}' = \mathbf{b}$;

1: Choose $\mathbf{x}^{(0)}$;
2: **for** $k = 0, 1, \ldots,$ until convergence **do**
3: **for** $i = 1, 2, \ldots, p$ **do**
4: Solve $\mathbf{A}_{ii}\mathbf{x}_i^{(k+1)} = \mathbf{b}_i - \sum_{j=1, j\neq i}^{p} \mathbf{A}_{ij}\mathbf{x}_j^{(k)}$ // *synchronous iterations*
5: Solve $\mathbf{A}_{ii}\mathbf{x}_i^{\text{new}} = \mathbf{b}_i - \sum_{j=1, j\neq i}^{p} \mathbf{A}_{ij}\mathbf{x}_j^{\text{old}}$ // *a–synchronous iterations*
6: **end for**
7: **end for**

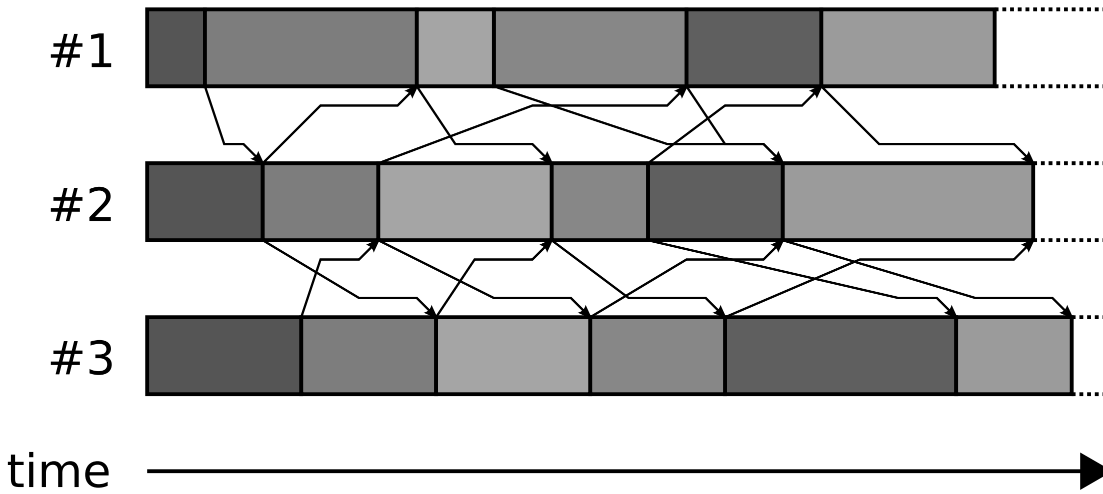


Figure 1.2: Time line of a certain type of asynchronous algorithm, showing three Jacobi iteration processes.

using the most recent information $\mathbf{x}^{(k)}$ to compute $\mathbf{x}^{(k+1)}$ at each iteration step k, a processor now computes $\mathbf{x}^{\text{new}}$ using information $\mathbf{x}^{\text{old}}$ 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 can potentially result in improved parallel performance.

In Figure 1.2 a graphical representation taken from [14] 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 pattern 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. In other words, newly computed information is sent at the end of each iteration step and newly received information is not used until the start of the next iteration step. Other kinds of asynchronous communication are possible [19, 20, 46, 69, 92]. For example, asynchronous iterative methods

exist where newly received information is immediately incorporated by the iteration processes.

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, problem characteristics, and fluctuations in processor workload.

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

(i) *Reduction of the global synchronisation penalty.* No global synchronisations are performed, an operation that may be extremely expensive in a heterogeneous environment.

(ii) *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.

(iii) *Coarse–grain.* Techniques from domain decomposition can be used to effectively divide the computational work and the lack of synchronisation results in a very attractive computation to 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 an overview of the various difficulties and bottlenecks.

(i) *Suboptimal convergence rates.* It is well–known that block Jacobi–type methods exhibit slow convergence rates. Furthermore, 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. Furthermore, the iteration may even diverge in some cases.

(ii) *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.

(iii) *Fault–tolerance issues.* If a Jacobi process is killed, the complete iteration process will break down. On the other hand, a process may become temporarily unavailable for some reason (e.g., due to network problems). Although this could delay convergence, the complete iteration process would resume upon reinstatement of said process.

(iv) *Load balancing difficulties.* Since the Jacobi processes can perform their computations practically independently of each other, dividing the computational work evenly may appear less important. However, significant desynchronisation of the iteration processes could negatively impact convergence rates. Therefore, some form of (possibly resource–aware) load balancing may still be appropriate.

In the next section it is shown how these disadvantages of asynchronous iterative methods are addressed in this thesis.

1.3.3 Acceleration: subspace methods

The main disadvantages of both synchronous and asynchronous block Jacobi–type iterations are that they suffer from slow convergence rates and that they only converge under certain strict

Algorithm 1.2 The preconditioned Conjugate Gradient method [81].

INPUT: Choose $\mathbf{x}_0$ and preconditioner $\mathbf{B}$; Compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
OUTPUT: Approximate solution $\mathbf{x}$ to $\mathbf{A}\mathbf{x}' = \mathbf{b}$

```
1: for k = 1, 2, ... do
2:     Solve B z_{k-1} = r_{k-1} // Preconditioning
3:     Compute ρ_{k-1} = r*_{k-1} z_{k-1}
4:     if k = 1 then
5:         Set p_1 = z_0
6:     else
7:         Compute β_{k-1} = ρ_{k-1}/ρ_{k-2}
8:         Set p_k = z_{k-1} + β_{k-1} p_{k-1}
9:     end if
10:    Compute q_k = A p_k // Matrix–vector multiplication
11:    Compute α_k = ρ_{k-1}/p*_k q_k
12:    Set x_k = x_{k-1} + α_k p_k
13:    Set r_k = r_{k-1} − α_k q_k
14:    if converged then stop // Convergence detection
15: end for
```

conditions. These methods can be improved significantly as follows. Using a starting vector $\mathbf{x}_0$ and the initial residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, iteration (1.2) may be rewritten as

$$\mathbf{u}_{k-1} = \mathbf{B}^{-1}\mathbf{r}_{k-1}, \quad \mathbf{c}_k = \mathbf{A}\mathbf{u}_{k-1}, \quad \mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{u}_{k-1}, \quad \mathbf{r}_k = \mathbf{r}_{k-1} - \mathbf{c}_k, \quad k = 1, 2, \ldots. \tag{1.7}$$

Instead of finding a new approximation $\mathbf{x}_k$ using information solely from the previous iteration step $k-1$, *subspace methods* operate by iteratively constructing some special subspace and extracting an approximate solution from this subspace. The key difference with classical methods is that information is used from several previous iteration steps, resulting in more efficient methods. This is accomplished by performing projections, which suggests that *inner products* need be to computed. This operation requires global synchronisation. This is a very expensive operation in the context of Grid computing and should be avoided as much as possible. This makes subspace methods less suitable to Grid computing.

Some popular subspace methods are: the Conjugate Gradient (CG) method [81], GCR [60], GMRES [104], Bi–CGSTAB [136], and the recently proposed IDR(s) method [120]. Purely for illustrative purposes, the so–called *preconditioned* Conjugate Gradient method is listed in Algorithm 1.2, which is designed for symmetric positive definite systems.

Generally speaking, subspace methods consist of four key building blocks, which can be identified as follows. The line numbers given below refer to line numbers of the preconditioned CG algorithm in Algorithm 1.2.

(i) *Matrix–vector multiplication (line 10).* Generally speaking, this is the most computationally expensive operation in each iteration step. Therefore, the total number of iterations until convergence is a measure for the total cost of a particular method. However, in the context of Grid computing, the inner product is the bottleneck operation. In subspace methods, at least one inner product is performed in each iteration step, so for this case the number of iteration steps is also a measure for the total cost.

(ii) *Preconditioning (line 2).* The matrix $\mathbf{B}$ in iteration (1.7) is called a *preconditioner*. As mentioned in Section 1.3.1, the art of preconditioning is to find the optimal trade–off between

the cost of solving systems involving the preconditioning matrix $\mathbf{B}$ and the “effectiveness” of the newly obtained update for the iterate.

(iii) *Convergence detection (line 14).* Normally speaking, an iterative method has converged when the norm of the residual has reached zero. However, checking for convergence is not entirely trivial. This has two main reasons: (i) the *recursively computed* residual $\mathbf{r}_k$ does not necessarily have to resemble the *true* residual $\mathbf{b} - \mathbf{A}\mathbf{x}_k$, and (ii) computing the norm of the residual $||\mathbf{r}_k||$ requires the evaluation of an inner product, which is a potentially expensive operation.

(iv) *Vector operations (remaining lines).* These include the inner products and the vector updates. Note that classical iterative methods lack inner products.

In many applications, finding an efficient preconditioner is more important than choosing some particular subspace method. Therefore, it is advantageous to put much effort in finding an efficient preconditioner. A popular choice is to use so–called *incomplete factorisations* of the coefficient matrix as preconditioners, e.g., ILU and Incomplete Cholesky [102]. Another well–known strategy is to find an approximate solution ε to $\mathbf{A}\varepsilon = \mathbf{r}_k$ by performing one or more iteration steps of some iterative method, such as 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, some subspace methods may break down if a different preconditioning operator is used in each iteration step. An example is the aforementioned preconditioned Conjugate Gradient method. Subspace methods that can handle a *varying preconditioner* are called *flexible*, e.g., FQMR [123], GMRESR [137], FGMRES [103], and flexible Conjugate Gradients [6, 96, 110]. A major disadvantage of most flexible methods is that they can incur additional overhead in the form of inner products.

1.4 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, we propose 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 shown in Chapter 3 of this thesis that this results in an effective method for solving large sparse linear systems on Grid computers.

For example, in Chapter 3 the flexible method GMRESR is used as the outer iteration and asynchronous block Jacobi is used as the preconditioner. The local subdomains in the preconditioning iteration are solved inexactly using the preconditioned IDR(s) method. Figure 1.3 shows a graphical representation of the resulting three–stage inner–outer method.

The proposed partially asynchronous algorithm exhibits many of the features that are on the wish list from Table 1.2 in Section 1.2. These include the following items.

- *Coarse–grain.* The asynchronous preconditioning iteration can be efficiently performed on Grid hardware with the help of domain decomposition techniques.
- *Minimal synchronisation.* By devoting the bulk of the computing time to preconditioning, the number of expensive global synchronisations can be reduced significantly.

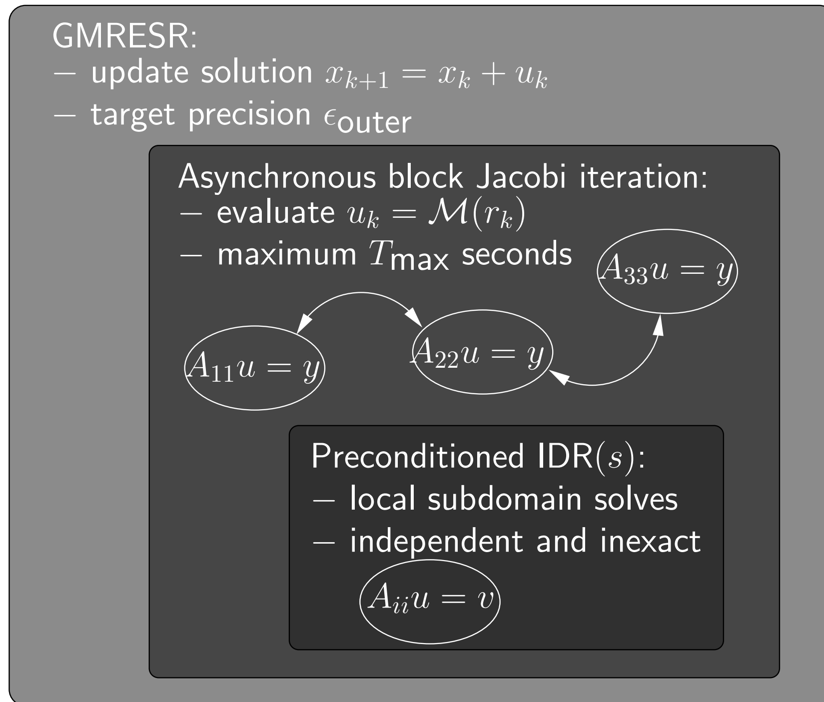


Figure 1.3: Three–stage iterative method: GMRESR—asynchronous block Jacobi—IDR(s).

- *Asynchronous.* Within the preconditioning iteration asynchronous communication is used, allowing for efficient overlap of communication with computation. In theory, the outer iteration process does not need to halt while waiting for a new update to arrive. It may continue to iterate until a new complete update can be incorporated.

- *Resource–aware and adaptive.* Experimental results indicate that the asynchronous preconditioning iteration is robust with respect to variations in network load.

In addition, the proposed method has several favourable properties.

- *Flexible and extendible iteration scheme.* The algorithm allows for many different implementation choices. For example, nested 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, resulting in a *decoupled* iteration approach (Section 1.9).

- *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 unfavourable 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.

Using this hybrid approach, the drawbacks of asynchronous iterative methods mentioned in Section 1.3.2 on page 9 are addressed as follows in this thesis.

(i) Problem: *Suboptimal convergence rates.* Solution: By using a (flexible) subspace method as the outer iteration, the asynchronous iteration is accelerated significantly.

(ii) Problem: *Non–trivial convergence detection.* Solution: 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.

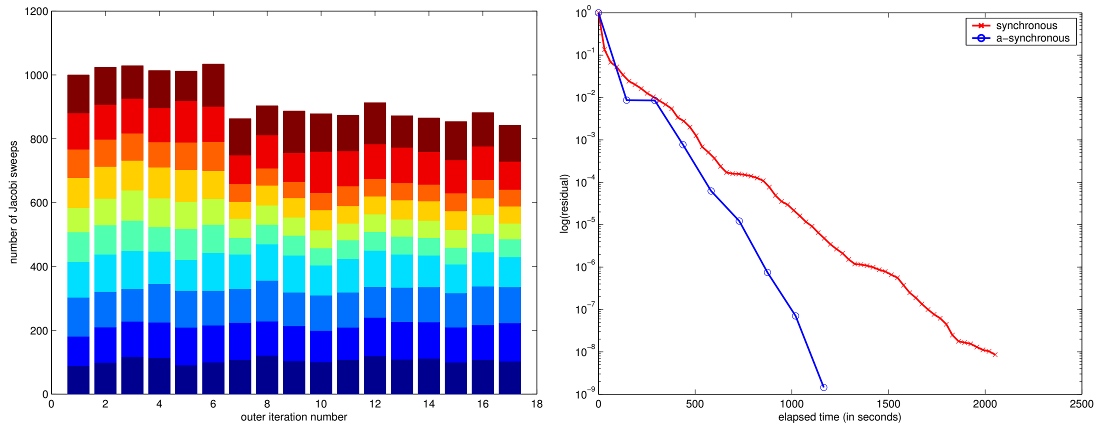
(iii) Problem: *Fault–tolerance issues.* Solution: In the preconditioning phase, each server iterates on a unique part of the vector $\mathbf{u}$. In heterogeneous computing environments, servers may become temporarily unavailable or completely disappear at any time, potentially resulting in loss of computed data. If the asynchronous process is used to solve the main linear system, these events would either severely hamper the convergence process or terminate the convergence process 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.

(iv) Problem: *Load balancing difficulties.* Solution: The fact that the asynchronous preconditioner can adapt to changes in network load suggests that the (possible) desynchronisation of the Jacobi processes does not significantly affect the global convergence rate.

The two remaining but essential issues are robustness and efficiency.

- *Robustness issues.* It is well–known that block Jacobi–type methods are slowly convergent for a large number of subdomains (i.e., processors). This problem is addressed in Chapter 3, where a coarse–grid correction is combined with the asynchronous iteration, resulting in a two–level preconditioning method [118]. Nevertheless, there are still 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 amount of time to spend on preconditioning can be 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, which implies that a flexible method has to be used. Not only can this introduce additional overhead in the outer iteration, the outer iteration has to be able to cope with a varying preconditioning operator in each iteration step. Therefore, *variable preconditioning* is an important theme in this thesis. For older methods such as CG and GMRES, this issue has been investigated extensively. However, to which extent the new IDR(s) method can cope with a varying preconditioner is an open problem and is investigated experimentally in Chapter 5. Also, in order to avoid potential computational bottlenecks, the outer iteration has to be performed in parallel as well.



(a) Effect of heterogeneity in computational environment on number of Jacobi iterations.

(b) Comparing synchronous and asynchronous preconditioning.

Figure 1.4: Experimental results asynchronous preconditioning.

1.5 Some experimental results

In order to give a general idea on the effect a heterogeneous computing environment may have on the performance of the proposed algorithm, two illustrative experiments will be presented.

Figure 1.4(a) shows the effect heterogeneity can have on the number of asynchronous Jacobi iterations performed on each server. This experiment is performed using ten servers on a large heterogeneous and non–dedicated local cluster during a typical workday. The figure shows the number of Jacobi iterations — broken down for each server — performed during each outer iteration step. Here, a fixed amount of time is devoted to each preconditioning step. After the sixth outer iteration several nodes began to experience an increased workload. The effect of the variability in computational environment on the number of Jacobi iterations is clearly visible.

In the second experiment a comparison in execution time is made between using synchronous preconditioning and asynchronous preconditioning to illustrates the potential gain of desynchronising part of a subspace method. The problem to be solved consists of one million equations using four servers within a heterogeneous computing environment. Figure 1.4(b) shows the execution time versus the 10–log norm of the residual. Each point represents an (expensive) outer iteration step (i.e., a GCR step). By devoting a relatively large (and fixed) amount of time to asynchronous preconditioning, the number of outer iterations is reduced considerably, cutting the total execution time nearly in half.

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

Part II: Efficient implementation in Grid computing

1.6 Introduction

The implementation of numerical methods on Grid computers is an involved process that 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.

Open MPI	CRAC	GridSolve
low–level language	mid–level language	high–level language
expert users	advanced users	novice users
general hardware	dedicated hardware	non–dedicated hardware
general communication	direct communication	bridge communicaton
general algorithms	asynchronous algorithms	task farming
general applications	coarse–grain applications	embarrassingly parallel
general data distributions	data persistence	non–persistent data
potentially fault–tolerant	no fault–tolerance	fault–tolerant
extendible	semi–extendible	mostly non–extendible
message passing	message passing	remote procedure call
coupled iterations	coupled iterations	decoupled iterations

Table 1.3: Several characteristics pertaining to three different types of (Grid) middleware.

At least four key ingredients can be distinguished when implementing numerical algorithms on Grid computers: (i) the *Grid middleware* (Section 1.7), (ii) the *target hardware* (Section 1.8), (iii) the *numerical algorithm* (Section 1.10), and last but not least, (iv) the *application* (Section 1.11). A particular component can greatly affect the remaining 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 in this part will take place within the general framework of the proposed approach mentioned in the first part, i.e., a flexible subspace method in combination with an asynchronous iterative method as a preconditioner. As argued, this algorithm possesses many features that make it perfectly suitable for Grid computing.

1.7 Grid middleware

One of the primary components in Grid computing is the *Grid middleware* (or *Grid programming environment*), which serves as the software layer between the user and the computational resources. The Grid middleware should facilitate client access to (possibly) remote resources and should deal with issues like heterogeneity and volatility. In which manner and to what extent the middleware handles these important issues depends on the type of middleware.

Although Grid middleware comes in many different shapes and sizes, the focus in this thesis will be on three examples, which are GridSolve [58, 154], CRAC [47] and Open MPI [71]. GridSolve is a distributed programming system that uses a client–server model for solving problems remotely on global networks. The CRAC library is specifically designed for efficient implementation of parallel asynchronous iterative algorithms on a cluster of clusters. Lastly, the Open MPI library is a widely–used implementation of the message–passing interface (MPI) standard.

These three middleware were chosen because they represent different types of programming models. The diversity is exemplified in Table 1.3, which lists some *prototypical* classifications pertaining to these three middleware. Some of these classifications are related to each other in the sense that some types of middleware are better suited for particular applications than others. For example, the bridge communication used in GridSolve would make it more appropriate for embarrassingly parallel problems. Note that both CRAC and Open MPI employ a message–passing programming model, while GridRPC uses the remote procedure call (RPC) programming model. Other important examples of Grid programming environments are GAT [3]

and ASSIST [2].

In this thesis, GridSolve, CRAC, and the Open MPI library are all used to implement parallel algorithms on Grid computers. In this context, GridSolve can be seen as a high–level programming language, while the Open MPI library can be viewed as a low–level programming language. The CRAC library can be considered as an inbetween. The apparent disadvantage of using a low–level library such as Open MPI is that functionalities important in Grid computing are often not readily available. For example, features like secure communication, detecting the availability of nodes, workload detection, and scheduling of processes.

The main reasons for choosing GridRPC were that is a standard for programming on Grid environments and that it is intended for implementing numerical algorithms on Grid computing. In addition, CRAC was chosen since it is designed for implementing (partially) asynchronous methods on a cluster of clusters, which is one of the two types of target hardware considered in this thesis. Finally, Open MPI was chosen for its flexibility and for its wide–spread availability.

In the following, descriptions of GridSolve, CRAC, and MPI–based libraries such as Open MPI will be given.

1.7.1 Description of GridSolve

GridSolve is a distributed programming system which uses a client–server model for solving complex problems remotely on Grid resources. It is an instantiation of the GridRPC model, a standard for a Remote Procedure Call (RPC) mechanism on Grid computers [109]. The GridRPC Application Programming Interface (API) is defined within the Global Grid Forum [89]. Other projects that implement the GridRPC API are DIET [34], NetSolve [108], Ninf–G [125], and OmniRPC [106].

Software environments such as GridSolve are often called Network Enabled Servers (NES). These systems typically consist of six components: clients, agents, servers, databases, monitors, and schedulers. We will elaborate on the specific details of these components in the context of version 0.17.0 of GS (see Figure 1.5). The GS servers (i.e., component 3 in Figure 1.5) are software components that are started on each computational node which may consist of a single CPU or a cluster. The server monitors the workload of the node and keeps an updated list of the services (or *tasks*) that are installed on the server. For example, a task can be a matrix–matrix multiplication or a parallel MPI job. Services can be added or modified without restarting the server.

A single GridSolve agent (component 2) actively monitors the server properties such as CPU speed, memory size, computational services, and availability. These properties are stored in a database on the agent node and are periodically updated. When a GridSolve client program (component 1) written in either C, Fortran, or Matlab uses the GridRPC API to initiate a GS call to a remote problem, the GS middleware first contacts the agent.

Based on the problem complexity, size of the input parameters, and the available computational resources, the agent then returns a list of servers sorted by minimum completion time. The client resorts the list after performing a quick network performance test. Input parameters are sent to the first server on the list and the task, which can be either blocking or non–blocking, is executed on the server. The result (if any) is then sent back to the client. If a task should fail it is transparently resubmitted to the next server on the list.

In order to determine the completion time of a particular task on server s, the total flop count of the problem is divided by the *effective speed* of the server. The latter is calculated using

$$\frac{s_{\text{flops}} \times s_{\text{ncpu}}}{\frac{s_{\text{workload}}}{100} + 1}, \tag{1.8}$$

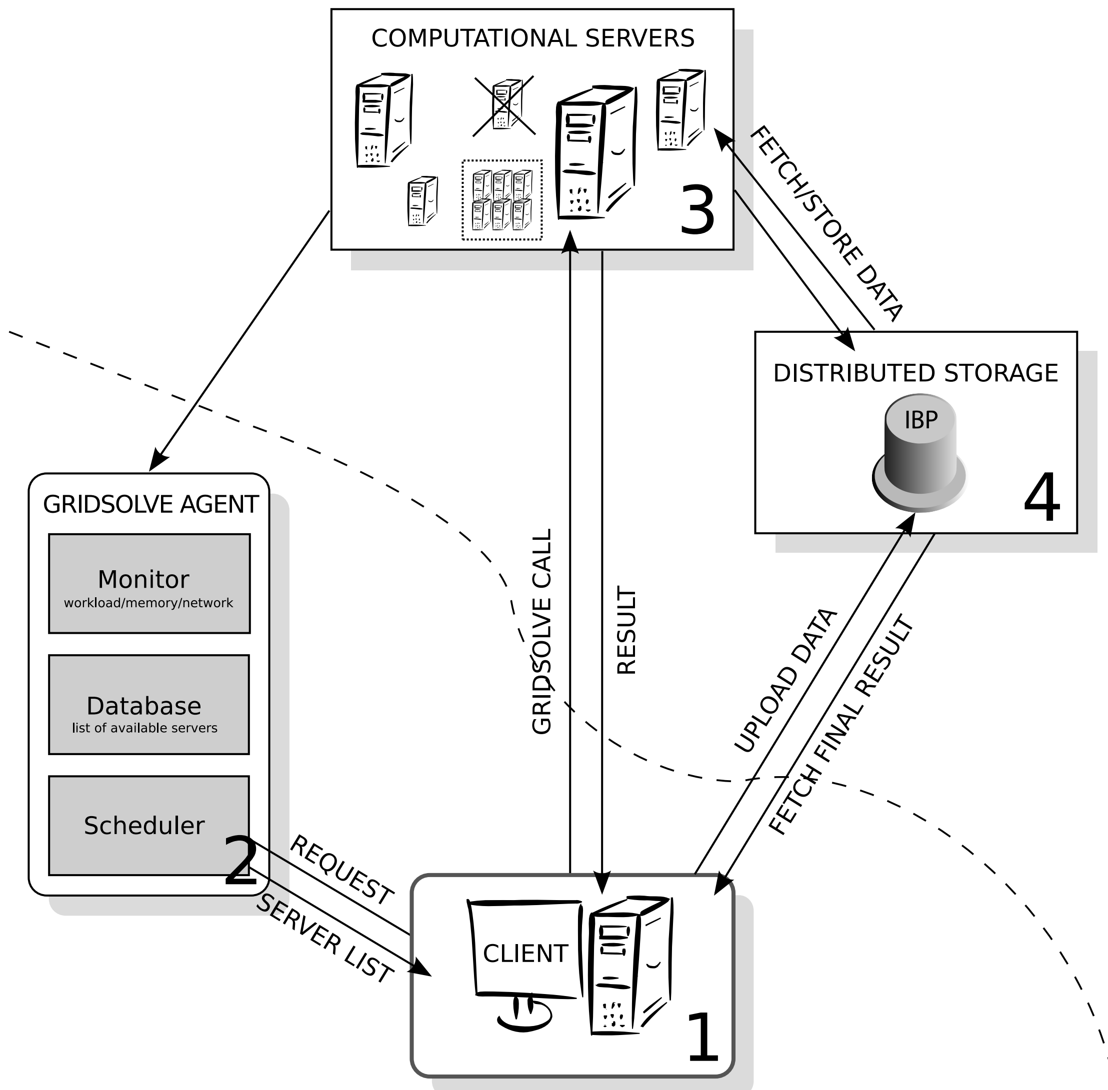


Figure 1.5: Schematic overview of GridSolve. The dashed line represents (geographical) distance between the client and servers.

where s_{flops} is the speed of server s in flops determined by multiplication of two dense matrices of fixed size, s_{ncpu} is the number of CPUs in the node, and $s_{\text{workload}} \in [0, 100]$ denotes the periodically updated workload. This means that if a server is fully occupied, the server can be used effectively half of the time, which is a realistic assumption.

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. It allows for convenient access to advanced remote computational resources. Furthermore, fault–tolerance is supported through a simple but effective mechanism. Nevertheless, the current implementation has several obvious limitations. For example, the remote servers cannot communicate directly, which imposes a severe constraint

on the type of applications that can be efficiently solved using the current implementation of GridSolve. It is therefore naturally suited for coarse–grained applications such as parametric studies and 'embarrassingly parallel' problems. In contrast, traditional parallel iterative solvers are inherently fine–grained and much research needs to be performed before iterative solvers can be efficiently applied in Grid computing.

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 is read or generated locally during the execution of a task is lost after it finishes. Several strategies such as data persistence and data redistribution have been proposed to tackle these deficiencies for different implementations of the GridRPC API [33, 30, 88, 156, 51, 29]. Furthermore, a proposal for a Data Management API within the GridRPC is currently being developed.

In GridSolve there is currently a partial solution to the data management problem called the Distributed Storage Infrastructure (DSI). At the Logistical Computing and Internetworking (LoCI) Laboratory of the University of Tennessee the IBP (Internet Backplane Protocol) middleware has been developed based on this approach [21]. To avoid multiple transmissions of the same data between the client and the server, the client can upload data to an IBP data depot which is in close proximity to the computational servers. Subsequently a data handle is sent to the server and the task can fetch and update the data on the IBP depot (component 4). Using the DSI can be considered as programming for a shared memory model.

An approach similar to [30] in which the RPC model of NetSolve is extended to include communication between remote servers has been developed for GridSolve [29].

1.7.2 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 easy implementation of (partially) asynchronous iterative algorithms [47, 46]. It allows for direct communication between the processes, both synchronous and asynchronous. The middleware provides 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 algorithm designer 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.

1.7.3 MPI–based libraries

The Open MPI library is a freely available implementation of the MPI–2 standard [71]. Although Open MPI and most other MPI implementations have functionalities for handling asynchronous and non–blocking communication, it lacks specific features such as message crunching as provided by CRAC. For a more extensive description of the MPI communication protocol, see for example [74, 73].

Special MPI–based libraries exist for running parallel applications in heterogeneous computing environments. For example, MPICH–G2 is a reference MPI implementation that is designed for running MPI applications in Grid environments [85]. It builds on the Grid software infrastructure provided by the Globus Toolkit [66]. Another MPI–based implementation that is tailored to heterogeneous networks of clusters is MPICH–Madeleine [90]. For real–world applications on Grid hardware that use MPICH–G2, see [150, 149, 28, 54, 93].

1.8 Target hardware

Numerous computing platforms exist that can be qualified as Grid computing hardware. However, in this thesis the focus will be on the following two types of architectures:

(i) *Local networks of non–dedicated computers* associated with organisations, such as universities and companies. These networks typically consist of the computers used daily by employees. Such hardware may considerably differ in speed, memory size, and availability. Examples of such clusters are the local networks at the Numerical Analysis group at Delft University of Technology.

(ii) *Dedicated cluster of clusters linked by a high–speed network.* For example, the Dutch DAS–3 national supercomputer is a multi–cluster of five geographically separated clusters. It is designed for dedicated parallel computing and although each cluster separately is homogeneous, the system as a whole can be considered heterogeneous.

The Grid middleware best suited to type (i) is GridSolve, while CRAC is more suitable to type (ii). Although the low–level nature of the Open MPI library allows it to be used on any type of architecture, it requires considerable programming effort. In the following, specific details will be given on the two types of architectures used in this thesis.

1.8.1 Local heterogeneous clusters

The first local cluster of the Numerical Analysis group at Delft University of Technology is a multi–user system and is moderately heterogeneous in design, consisting of twelve nodes: six Intel 2.20 GHz machines, two Intel 2.66 GHz machines, and four AMD Athlon 2.20 GHz machines. The nodes are equipped with memory in the range 2–4 GB and the cluster is interconnected through 100 MB/s Ethernet links.

The second local cluster consists of ten single core (AMD Athlon 64 Processor 3700 at 2.4GHz) and two dual core CPU nodes (Intel Core 2 CPU 6700 at 2.66GHz) with 3 GB and 8 GB of memory respectively and running Linux 2.6.

Since the clusters are used by employees for their daily work activities, the workload of a node can vary considerably.

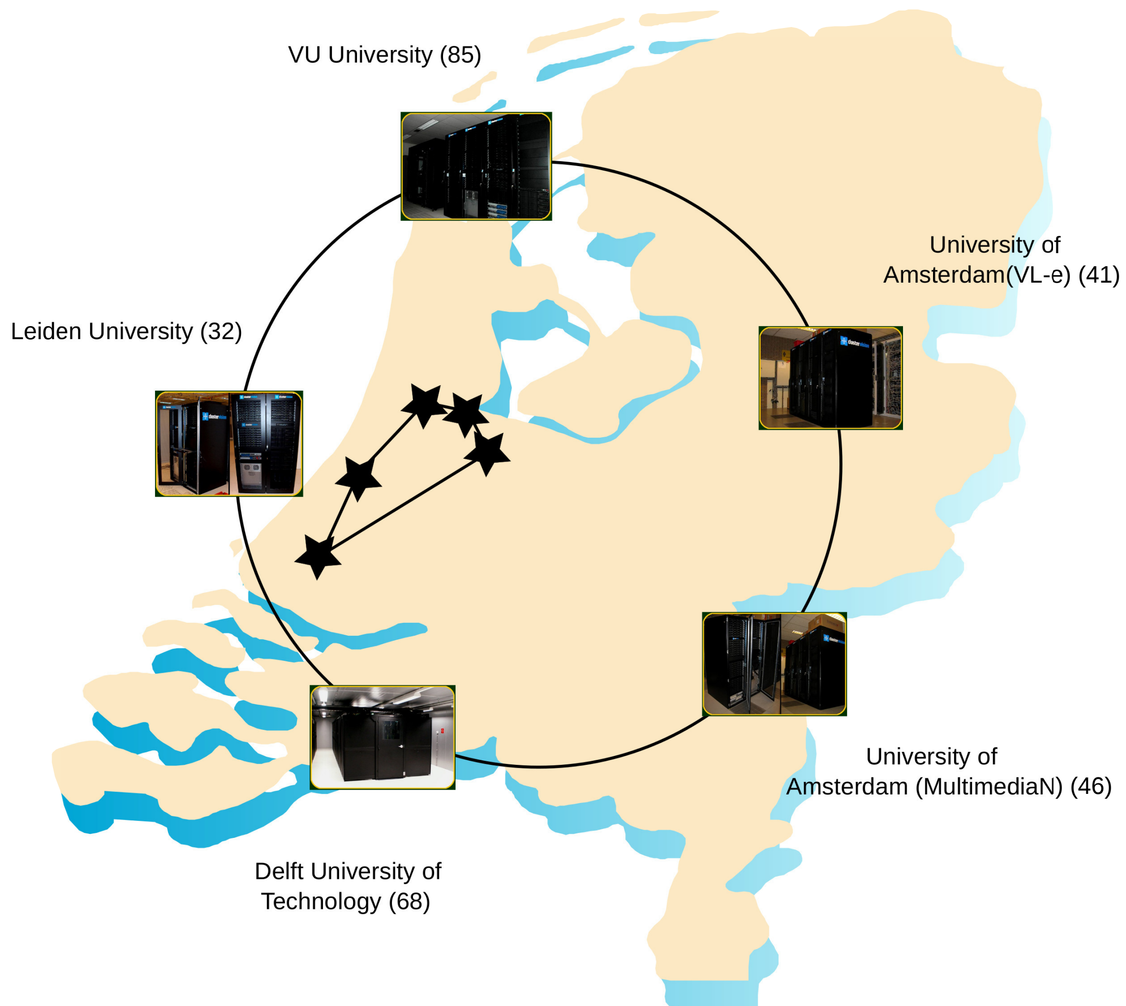


Figure 1.6: The DAS–3 multi–cluster.

Cluster (name)	Nodes (#)	Cores (type)	Speed (Ghz)	Memory (GB)	Storage (TB)	HDD (GB)	Network (type)
VU	85	dual	2.4	4	10	250	Myri–10G & GbE
LU	32	single	2.6	4	10	400	Myri–10G & GbE
UvA	41	dual	2.2	4	5	250	Myri–10G & GbE
TUD	68	single	2.4	4	5	250	only GbE
UvA–MN	46	single	2.4	4	3	1,500	Myri–10G & GbE

Table 1.4: DAS–3: “five clusters, one system”.

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

1.8.2 DAS–3: "five clusters, one system"

The Distributed ASCI Supercomputer 3 (DAS–3) is a multi–cluster consisting of five clusters, located at four academic institutions across the Netherlands [107]. 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 (GbE) interconnect and high speed Myri–10G interconnect. The TUD site only employs Gigabit Ethernet interconnect.

More specific details on the five sites are given in Table 1.4, while Table 1.5 lists average roundtrip measurements between four of the five DAS–3 sites on a lightly loaded network. Figure 1.6 shows the ring structure of the network topology of the DAS–3, as well as the (approximate) geographical locations of the five sites. 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.

1.9 Coupled vs. decoupled inner–outer iterations

The fact that there are essentially two distinct iteration processes in the proposed hybrid algorithm offers much freedom with respect to implementation.

The DAS–3 multi–cluster is designed for dedicated parallel computing and in order to preserve data locality, it is natural to perform the outer iteration and preconditioning iteration on the same set of nodes, resulting in a *coupled* iteration process. 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.

The GridSolve middleware allows for physically *decoupling* the outer iteration and the preconditioning iteration. It then becomes feasible to perform the inner iteration on unreliable (i.e., heterogeneous and distant) computational resources, while the outer iteration is performed on more stable (i.e., homogeneous and local) hardware, resulting in a partially fault–tolerant algorithm.

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 com-

plicated. The CRAC middleware is more suited for dedicated computational hardware where network connections between nodes may become temporarily unavailable.

1.10 Parallel iterative methods: building blocks revisited

The next step in implementing iterative algorithms on Grid computers is to revisit the four building blocks of subspace methods as mentioned in Section 1.3.3 on page 9 and discuss their efficient implementation on Grid computers. Where appropriate, each building block will be discussed in the context of the aforementioned two types of target architectures with references to chapters of the thesis.

1.10.1 Matrix–vector multiplication (or data distribution)

Generally speaking, the matrix–vector multiplication is the most expensive operation in parallel iterative methods. It is therefore important to create an efficient partitioning of the coefficient matrix for the matrix–vector multiplication. The problem of *load balancing* is to divide the computational work of the matrix–multiplication as evenly as possible under the constraint of minimal communication. For parallel sparse matrix–vector multiplication, this can be achieved by hypergraph partitioning algorithms used by software packages such as *Mondriaan* [142] and *PaToH* [35]. In the context of Grid computing, the load balancing algorithm may also need to incorporate properties related to the heterogeneity of the computational hardware into the partitioning process [132]. Also, the computational effort associated with the partitioning process itself is far from negligible and may be performed in parallel as well [52].

In this thesis, the coefficient matrix is partitioned using several different approaches, depending on the computational setting and the employed solution algorithms. In the following, these approaches will be discussed and motivated.

In Chapter 2 a synchronous method is used as a preconditioner and Section 2.2.2 describes a heterogenous block–row distribution of the matrix that takes into account the computational speed of the nodes. The rationale behind this choice is two–fold:

1. *Block–row* distribution: for the type of applications considered in this thesis, the number of non–zeros on each row of the coefficient matrix is roughly the same, so only nearest–neighbour communication is required when this partitioning is used.

2. *Heterogenous* distribution: in this completely synchronous (i.e., more traditional) context, “conventional” load balancing techniques for heterogeneous systems are more natural [132, 53].

In Chapter 3 an asynchronous iterative method is used as a preconditioner. Here, a homogeneous block–row distribution is employed. The reason for choosing a block–row distribution is the same as given above. The reason for choosing a homogeneous distribution is as follows. The purpose of load balancing is to divide the computational work as evenly as possible so that tasks finish roughly at the same time, avoiding idling of processors at synchronisation points. In contrast, the key point of asynchronous methods is that processors do not wait for information from other processors, which essentially negates the need for any type of load balancing. However, significant desynchronisation of the Jacobi processes could result in suboptimal convergence rates and therefore some form of load balancing may still be appropriate [13, 12].

In Chapter 4, both synchronous and asynchronous preconditioning techniques are investigated. In the asynchronous approach, the partitioning is equal to the one used in Chapter 3 for the same reasons as listed above. For the synchronous approach, instead of partitioning

the computational domain in strips like in Chapters 2 and 3, the domain is partitioned using a three–dimensional block partitioning. The reason for this partitioning is that in Chapter 4 the goal is to construct and test a parallel iterative method that has good scalability in the strong sense. A two–dimensional domain partitioning as used in Chapters 2 and 3 does not scale well in the strong sense [25].

Throughout this thesis, the data distribution used in the outer iteration is always equal to the data distribution used in the preconditioner. This is done for convenience only. Especially in a decoupled iteration approach such as used in Chapter 3, it may make sense to use more advanced partitioning algorithms in the outer iteration, such as the aforementioned hypergraph partitioners.

Note that efficient load balancing for the asynchronous preconditioning iteration can be problematic. The main reason is as follows. The bulk of the computational work in the preconditioning iteration consists of solving the block diagonal system in each Jacobi iteration step. These local linear systems are solved iteratively and in most cases inexactly. Furthermore, problem characteristics may cause erratic convergence rates. Therefore, the amount of work is hard to predict, which makes load balancing difficult.

1.10.2 Preconditioning

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 (IC) and block ILU [102].

2. *Domain decomposition techniques.* Many applications in scientific computing involve 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 [118].

3. *Multilevel techniques.* Solutions often contain both slowly–varying and fast–varying components. By solving the same problem at different scales in a recursive manner, both components can be efficiently captured. Examples of such methods are multigrid, deflation, and domain decomposition with coarse grid correction [67, 145].

In this thesis, preconditioners from all these classes are used in some form or another. To be more precise, in Chapter 2 Jacobi and block Jacobi preconditioning is used and in Chapter 3 an asynchronous iterative method is used as a preconditioner. These types of preconditioning fall into the class of domain decomposition techniques (class 2). In Chapter 3 the asynchronous iterative method is also combined with a coarse grid correction, resulting in a multi–level technique (class 3). In addition, IC and ILU preconditioning is used (class 1) when iteratively solving the local subdomains within the asynchronous preconditioning iteration.

Chapter 3 shows that asynchronous iterative methods combined with a deflation step can be efficiently parallelised on Grid computers and that they are highly effective preconditioners for large and difficult linear systems.

1.10.3 Convergence detection

Another 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 may be virtually impossible to construct a robust and efficient convergence detection algorithm.

On the other hand, if the preconditioning iteration is performed on dedicated but geographically separated hardware such as the DAS–3 architecture, sophisticated decentralised convergence detection algorithms could be employed, e.g. [14, 27, 143].

In this thesis, all these issues are circumvented by spending a *fixed* amount of time on each preconditioning step. The main disadvantage of this approach is that determining the ideal amount of said time may be extremely problem–dependent.

1.10.4 Vector operations

Vector updates are embarrassingly parallel, so they can be performed efficiently in parallel computing and in particular in Grid computing. In contrast, efficiently performing inner products in parallel computing is a difficult problem, but even more so in Grid computing. In this thesis, the focus is not so much on the efficient computation of inner products in Grid computing, but more on minimising the amount of synchronisation points that are induced by the inner products.

A notable exception is Section 4.4.2 on page 82, where the so–called IDR test matrix $\widetilde{\mathbf{R}}$ in IDR(s) is chosen so that the work, communication, and storage involving this matrix is minimised in multi–cluster environments. Using Open MPI, a method is presented for the efficient computation of the s combined inner products $\widetilde{\mathbf{R}}^*\mathbf{r}$ for a matrix $\widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$ and a vector $\mathbf{r} \in \mathbb{C}^n$ on a cluster of clusters. In such computational environments, the intercluster communication is more costly than the intracluster communication. In order to minimise communication across the intercluster links, each column of $\widetilde{\mathbf{R}}$ is chosen non–zero on one cluster and zero on the remaining clusters. By first combining the local results on a single cluster and then exchanging these partial results with the other clusters, the multiplication $\widetilde{\mathbf{R}}^*\mathbf{r}$ can be performed efficiently in this case.

Grid middleware such as MPICH–G2 provides more general topology–aware collective operations. For more information, see [86].

1.11 Applications

In this thesis, the main focus is on the efficient iterative solution of very large linear systems on Grid computers. In principle, the techniques presented in this thesis are applicable to many types of linear systems: sparse or dense, symmetric or non–symmetric, singular or non–singular, square or overdetermined.

The focus is on two important problems originating from computational fluid dynamics: large bubbly flow problems and large (more general) convection–diffusion problems. These bubbly flow problems are solved on very fine meshes in conjunction with large jumps in the coefficients, which results in symmetric and sparse linear systems that are severely ill–conditioned. On the other hand, the convection–diffusion problems result in non–symmetric and sparse linear systems of equations.

For applications from computational fluid dynamics, the so–called *Immersed Boundary Method* (IBM) is particularly 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 regular 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 this reason, only Cartesian meshes are considered in this thesis. For more discussion on IBMs, the reader is kindly referred to the excellent book chapter [80].

1.12 Further reading

For the interested reader, the classic book by Dimitri Bertsekas and John Tsitsiklis contains a wealth of information on parallel asynchronous iterative algorithms for solving linear systems using various applications [24]. More recently, Jacques Bahi, Sylvain Contassot–Vivier, and Raphaël Couturier wrote a book on parallel iterative algorithms for solving linear systems on Grid computing using synchronous and asynchronous iterative methods [8]. More extensive discussions on various aspects of parallel scientific computing may be found in the excellent book by Rob Bisseling [25].

For a comprehensive discussion on iterative methods for solving linear systems, the classic book by Gene Golub and Charles van Loan is greatly recommended [72], as well as the excellent book by Yousef Saad [102]. More on domain decomposition techniques can be found in [118, 133]. For more technical details on Grid hardware and Grid software technologies, the reader is referred to [23, 57, 55, 65]. The recent overview article on iterative methods by Valeria Simoncini and Daniel Szyld is also recommended [111]. Another excellent overview article by Michele Benzi discusses various types of preconditioning techniques [22].

Part III: General thesis remarks

1.13 Related work and main contributions

Throughout this chapter, numerous references are given to papers that are related (either directly or more indirectly) to the techniques used in this thesis. In this section, the contributions of this thesis are made explicit, highlighting the references that are the most relevant.

The study of asynchronous iterations — both theoretical and practical — dates back as far as 1969, when Chazan and Miranker published their seminal paper on chaotic relaxation methods [36]. Since then, asynchronous iterations have been used in a wide variety of scientific applications, such as inverse problems in geophysics and oil exploration [100], electrical power networks [18], network flow [20], and nonlinear problems [124, 10]. Some of the topics not covered in this thesis that could also be important are stopping criteria [14, 27, 143] and using overlapping subdomains [122, 68]. For a comprehensive overview paper and additional references on asynchronous iterative methods, see [70].

There exist several projects that aim to (among other things) efficiently solve linear systems on heterogeneous computing grids and we will mention two of these projects. Note that this list is by no means exhaustive and that it is meant to give an idea on the variety in projects.

- The GREMLINS[1] (GRid Efficient Methods for LINear Systems) project [46] aims to construct new algorithms for solving large sparse linear systems on heterogeneous and

[1] `http://info.iut-bm.univ-fcomte.fr/gremlins/`

distributed clusters. The CRAC library [47] used in this thesis (Section 1.7.2) was developed within the context of this project. Other papers associated with this project are [11, 14, 143, 15, 9].

- The GRAAL[2] project is about the design of algorithms and schedulers for distributed heterogeneous platforms. Part of this project is the DIET[3] (Distributed Interactive Engineering Toolbox) systems which is similar to GridSolve mentioned in Section 1.7.1. Also, it includes the MUMPS[4] (MUltifrontal Massively Parallel sparse direct Solvers for symmetric and unsymmetric, complex and real sparse matrices) project [4].

Despite the amount of research dedicated to tackling the difficult problems considered in this thesis, there are not many papers that present convincing results of efficiently running truly fine–grained parallel algorithms in Grid computing environments. One of the few exceptions is [144], where for any given position in the Awari board game the best possible move is computed in parallel on the DAS–3. For this type of application, the communication to computation is particularly high and asynchronous communication is used to obtain good performance. In [17], similar good results are reported for two other applications that search large state spaces: transposition driven search and model checking.

In particular, asynchronous iterative methods have never really been successfully applied to the solution of extremely large sparse linear systems. 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 favourable property in heterogeneous computing systems.

Although Krylov subspace methods such as the Conjugate Gradient method [81] 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 (flexible) Krylov subspace method, the best of both worlds is combined. It is shown in this thesis that this 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.

To the best of our knowledge, the idea of using an asynchronous preconditioner in a flexible method is an algorithmic innovation that has not been investigated in the context of Grid computing before. This constitutes one of the main contributions of this thesis.

The combination of four techniques in Chapter 4 results in an efficient iterative method for solving large sparse nonsymmetric linear systems on Grid computers, which is another main contribution of this thesis.

1.14 Scope and outline

This dissertation deals with the design, implementation, and analysis of efficient iterative solution methods for distributed heterogeneous computing systems. Different strategies are used to attack this important but difficult problem, depending on aspects such as application, middleware, algorithm, and target hardware.

Generally speaking, there are two main themes in this thesis: *asynchronous* preconditioning and "traditional" *synchronous* preconditioning. For the readers' convenience, an overview of the different methods and middleware that are used in this thesis is given in Tab 1.6.

[2] `http://graal.ens-lyon.fr/`
[3] `http://graal.ens-lyon.fr/~diet/`
[4] `http://graal.ens-lyon.fr/MUMPS/`

	synchronous preconditioning		asynchronous preconditioning		
chapter #	2	4	3	3	4
middleware	GridSolve	Open MPI	GridSolve	CRAC	CRAC
method	CG	IDR(s)	GCR	flexible CG	IDR(s)

Table 1.6: Overview of the iterative methods and middleware used in this thesis.

In Section 1.2 of this chapter it was argued that the ultimate numerical algorithm for Grid computing exhibits the following features: *robust*, *efficient*, *coarse–grained*, *asynchronous*, *minimal synchronisation*, *resource–aware*, *adaptive*, and *fault–tolerant*. In this section it is explicitly shown in which manner the algorithms presented in this thesis exhibit these features. This is done by giving a description of each thesis chapter where each chapter is put in the general context of the thesis. Chapters 1 through 4 of this thesis are all based on references that either have been published or that have been accepted for publication. Portions of Chapter 5 have been submitted for publication.

Chapter 1 contains a broad overview of the issues related to efficient iterative methods for Grid computing, motivating the choices made in this thesis. This chapter has appeared as [41]:

> T. P. Collignon and M. B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. In B. Koren and C. Vuik, editors, *Advanced Computational Methods in Science and Engineering. Springer Series Lecture Notes in Computational Science and Engineering*, volume 71, pages 79–106. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

The thesis continues by attacking the problem using more "traditional" approaches. In Chapter 2 a case study is described on iteratively solving large linear systems on Grid computers using standard block Jacobi (i.e., synchronous) preconditioning within the software constraints of the Grid middleware GridSolve and within the algorithmic constraints of preconditioned Conjugate Gradient–type methods. The algorithm presented in this chapter not only has *minimal synchronisation*, but is also *resource–aware* and *adaptive*. The contents of this chapter has appeared as [43, 40]:

> T. P. Collignon and M. B. van Gijzen. Two implementations of the preconditioned Conjugate Gradient method on heterogeneous computing grids. *International Journal of Applied Mathematics and Computer Science*, 20(1):109–121, 2010.

> T. P. Collignon and M. B. van Gijzen. 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*, pages 527–540, 2007.

An efficient and robust preconditioner is crucial for rapid convergence of iterative methods and in Chapter 3 the proposed *asynchronous* preconditioning approach is investigated in more detail. The partially asynchronous algorithm is implemented on two different types of Grid hardware applied to two different applications using two different types of Grid middleware. The results from this chapter show that the method is very *effective* in solving large linear systems on Grid computers and that the asynchronous preconditioner can *adapt* to changes in network load. In order to increase the *robustness* of the algorithm, the asynchronous preconditioning iteration is complemented with a coarse–grid correction in the form of a deflation step. In overall, this

chapter shows that the use of an asynchronous preconditioner results in a algorithm that not only has *minimal synchronisation*, but is also *efficient*, *robust*, *coarse–grained*, *asynchronous*, *adaptive*, and *fault–tolerant*. This chapter has appeared as [44, 42]:

> T. P. Collignon and M. B. van Gijzen. Fast iterative solution of large sparse linear systems on geographically separated clusters. *International Journal of High Performance Computing Applications*, 2011. (to appear).

> T. P. Collignon and M. B. van Gijzen. Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. In G. Kreiss, P. Lötstedt, A. Målqvist, and M. Neytcheva, editors, *Numerical Mathematics and Advanced Applications 2009: Proceedings of ENUMATH 2009, the 8th European Conference on Numerical Mathematics and Advanced Applications, Uppsala, July 2009*, pages 261–268. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

Chapter 4 brings everything together to focus on the key theme of this thesis: *designing efficient iterative methods for cluster and Grid computing*. The IDR(s) method and its variants are new and efficient iterative algorithms for solving large nonsymmetric linear systems. In this chapter, four strategies are used to minimise global synchronisation in IDR(s) methods. Firstly, an *efficient* and *robust* IDR(s) variant is presented that has a single global synchronisation point per matrix–vector multiplication step. Secondly, the so–called *IDR test matrix* in IDR(s) can be chosen freely and this matrix is constructed such that the work, communication, and storage involving this matrix is *minimised* in multi–cluster environments such as the DAS–3. Thirdly, a methodology is presented for *a priori* estimation of the optimal value of s in IDR(s). Finally, the proposed IDR(s) variant is combined with an asynchronous preconditioning iteration as analysed in Chapter 3. These four strategies result in IDR(s) variants that are very efficient in solving large non–symmetric linear systems on Grid computers. The contents of this chapter has appeared as [45, 138]:

> T. P. Collignon and M. B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 2011. (published online: 14 january 2011).

> M. B. van Gijzen and T. P. Collignon. Exploiting the flexibility of IDR(s) for Grid computing. In *The Proceedings of the 2nd Kyoto–Forum on Krylov Subspace Methods, Kyoto University, Kyoto, Japan*, March 2010.

In the last part of Chapter 4 an asynchronous preconditioner is used by IDR(s), which means that the IDR(s) method is treated as a *flexible* method. The original motivation for the work performed in Chapter 5 was to begin analysing mathematically the effect of a varying preconditioning operator on the convergence properties of IDR(s). This was done by interpreting the IDR(s) method as a special type of *deflation* method, which in fact revealed several interesting properties of IDR(s). This chapter has been submitted for publication as [39]:

> T. P. Collignon, G. L. G. Sleijpen, and M. B. van Gijzen. Interpreting IDR(s) as a deflation method. *Journal of Computational and Applied Mathematics*, 2011. Special Issue: Proceedings ICCAM–2010 (submitted).

Last but not least, Chapter 6 lists the general conclusions of this thesis and gives directions for future research.

notation	meaning
$\mathbf{I}$	identity matrix of appropriate dimension
$\mathbf{0}_{n,s}$	all–zero matrix of dimension $n \times s$
$\lvert\lvert\mathbf{v}\rvert\rvert$	Euclidean norm of $\mathbf{v}$
$\mathbf{A}^*$	adjoint of $\mathbf{A}$
$\mathcal{N}(\mathbf{A})$	nullspace of $\mathbf{A}$
$\operatorname{rank}\mathbf{A}$	rank of $\mathbf{A}$
$\sigma(\mathbf{A})$	set of eigenvalues of $\mathbf{A}$
$\operatorname{span}(\mathbf{A})$	range of $\mathbf{A}$
$\dim\mathcal{A}$	dimension of the vector space $\mathcal{A}$
$\mathbf{A}^\perp$	orthogonal complement to the column space of $\mathbf{A}$
$\mathbf{v} \perp \mathbf{A}$	$\mathbf{v}$ is orthogonal to all column vectors of $\mathbf{A}$
$\phi_{(m:n)}$	column vector $[\phi_m, \phi_{m+1}, \ldots, \phi_n]^\top$
MV	abbreviation of Matrix–Vector multiplication
$\widetilde{\mathbf{R}}$	s-dimensional *initial shadow residual* or *IDR test matrix* of IDR(s)
$\mathbf{B}$	"traditional" preconditioning matrix
$\mathbf{u}_x$	partial derivative of a function $\mathbf{u}$ with respect to x
s	number of vectors to generate in the subspaces $\mathcal{G}$ of IDR(s)
t	number of "standard" deflation vectors
$\mathcal{K}_r(\mathbf{A}, \mathbf{v})$	Krylov subspace of order r generated by a matrix $\mathbf{A}$ and a vector $\mathbf{v}$, i.e., $\operatorname{span}(\mathbf{v}, \mathbf{A}\mathbf{v}, \ldots, \mathbf{A}^{r-1}\mathbf{v})$
$\mathcal{G}$	IDR subspace
$\mathcal{S}$	Sonneveld subspace
ω_k	smoothing parameter in BiCGSTAB and IDR(s) in cycle k

Table 1.7: Notational conventions and nomenclature. Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ and $\mathbf{v} \in \mathbb{C}^n$.

1.15 Notational conventions and nomenclature

Matrices and vectors are written in boldface (e.g., $\mathbf{A}$ and $\mathbf{x}$), while scalars are written in italic face (e.g., n and γ). Vector spaces are written in calligraphic font (e.g., $\mathcal{G}$ and $\mathcal{S}$). A field is written in blackboard font (e.g., $\mathbb{C}$ and $\mathbb{N}$). A projection is always written as capital Π. For an overview of additional notational convections, see Table 1.7.

Chapter 2

Conjugate Gradient Methods for Grid Computing

This chapter has been published as:

T. P. Collignon and M. B. van Gijzen. Two implementations of the preconditioned Conjugate Gradient method on heterogeneous computing grids. *International Journal of Applied Mathematics and Computer Science*, 20(1):109–121, 2010.

T. P. Collignon and M. B. van Gijzen. 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*, pages 527–540, 2007.

Overview

This chapter describes a case study on iteratively solving large sparse linear systems on Grid computers within the software constraints of the Grid middleware GridSolve (Section 1.7.1 on page 16) and within the algorithmic constraints of preconditioned Conjugate Gradient–type (PCG) methods. The goal of this chapter is to efficiently solve linear systems on Grid computers using techniques that are more "traditional", such as synchronous preconditioning. We identify the various bottlenecks induced by the middleware and the iterative algorithm. We consider the standard CG algorithm of Hestenes and Stiefel, and as an alternative the Chronopoulos and Gear variant, a formulation that is potentially better suited for grid computing since it requires only one synchronisation point per iteration, instead of two for standard CG. In addition, we improve the computation–to–communication ratio by maximising the work in the preconditioner. In addition to these algorithmic improvements, we also try to minimise communication overhead within the communication model currently used by the GridSolve middleware. We present numerical experiments on 3D bubbly flow problems using heterogeneous computing hardware that show lower computing times and better speed–up for the Chronopoulos and Gear variant of Conjugate Gradients. Finally, we suggest extensions to both the iterative algorithm and the middleware for improving the granularity.

2.1 Introduction

We study two different implementations of the Conjugate Gradient (CG) method [81] on a heterogeneous computational grid. We use the GridSolve library [58], which is mature Grid middleware for accessing remote computational resources. Load balancing is achieved using a simple resource–aware data partitioning strategy. The number of synchronisation points in the CG algorithm which is in its standard implementation equal to two, can be reduced to one by using the implementation that has been proposed by Chronopoulos and Gear [37]. Chronopoulos and Gear claim stability, based on numerical experiments using this variant.

We apply our approach to the bubbly flow problem which is an important example of a moving boundary problem. Our numerical experiments show that by minimising the number of synchronisation points and by devoting more work to the preconditioning phase, speed–up can be achieved for the solution of systems of equations, despite the fact that for this application the tasks are tightly coupled.

The remainder of this chapter is organised as follows. In the next section we describe in detail our architecture–aware Conjugate Gradient algorithm. This includes a description of the test problem and several details concerning our implementation of a sparse iterative solver on Grid computers. Section 2.3 contains experimental results and in Section 2.4 we give concluding remarks and some suggestions for improvements.

2.2 Heterogeneous sparse linear solvers in GridSolve

2.2.1 Motivation

We want to simulate general moving boundary problems using Grid computers. Examples of said problems are the swimming of fish, airflow around wind turbine rotor blades, and bubbly flows. These simulations involve numerically solving the governing fluid equations on a structured grid, where the most expensive part usually consists of solving a large sparse linear system $\mathbf{Ax} = \mathbf{b}$ at each time step. When using a pressure–correction method [141] to solve the governing equations

for bubbly flows on a very fine 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,

$$\begin{cases} -\nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla \mathbf{p}(\mathbf{x}) \right) = \mathbf{f}(\mathbf{x}), & \mathbf{x} \in \Omega, \\ \frac{\partial}{\partial \mathbf{x}} \mathbf{p}(\mathbf{x}) = \mathbf{g}(\mathbf{x}), & \mathbf{x} \in \partial\Omega, \end{cases} \tag{2.1}$$

for some functions $\mathbf{f}$ and $\mathbf{g}$. Here, Ω and $\partial\Omega$ denote the computational domain and boundary respectively, while $\mathbf{p}$ and ρ represent the pressure and density. In this chapter we will consider the 3D test problem taken from [135, 129]. It is a two–phase bubbly flow problem where we have two separate fluids Γ_0 and Γ_1, representing water (high–density phase) and water vapour (low–density phase) respectively. The corresponding density function has a large jump, defined by

$$\rho(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in \Gamma_0; \\ \tau, & \mathbf{x} \in \Gamma_1, \end{cases} \tag{2.2}$$

where we typically have $\tau = 10^{-3}$. Such a discontinuity in the coefficient results in an ill–conditioned system, making it a difficult problem for iterative methods. For the purpose of this chapter we restrict ourselves to a cubical unit domain with a single bubble with radius 0.25 located in the center of the computational domain. For more details on applying the pressure–correction method to bubbly flows the reader is referred to [135].

Applying standard finite differences to (2.1) on a structured $n_x \times n_y \times n_z$ mesh results in the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.3}$$

where $\mathbf{A}$ is an $n \times n$ block pentadiagonal symmetric positive semi–definite (SPSD) sparse matrix and $n = n_x n_y n_z$. This implies that the solution $\mathbf{x}$ is determined up to a constant. However, it can be shown that for our particular case this does not pose any problem for the iterative solver [128].

The reason that we chose to solve this system using a preconditioned Conjugate Gradient (PCG) method is twofold: (i) it is the obvious choice for large and sparse SP(S)D systems, and (ii) the CG method consists of three basic computational kernels (i.e., matrix–vector multiplication, inner product, and vector update), which are simple to implement and relatively straightforward to parallelise on (dedicated) parallel computers. Also, we combine CG with a (block) Jacobi preconditioner due to its attractive parallelisation properties.

2.2.2 Resource–aware load balancing

We are interested in solving large sparse linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$ using GridSolve with architecture–aware dynamic load balancing. For this purpose it is insufficient to let the agent return a sorted list based on problem complexity and available resources, as is normally being done in GridSolve. Instead, we need to use a slightly different approach. Referring to Figure 1.5 on page 17, suppose that the client (i.e., component 1 in Figure 1.5) wishes to use s servers (component 3) to solve a linear system. The scheduler in the GridSolve agent (component 2) has been enhanced so that it creates a simple (non–homogeneous) partitioning of the computational work over s servers using information about currently available resources. It then returns the partitioning and a list of said servers to the client, after which the client initiates a series of non–blocking calls *explicitly specifying* the size and location of each task. Thus we ensure that the computational task is

Algorithm 2.1 Resource–aware Preconditioned Conjugate Gradient Algorithm; s servers

1: Agent partitions work based on available computational resources;
2: Client sets initial values and uploads initial vectors such as $\mathbf{r}_0$ to the IBP data depot; Set $k = 0$;
3: **while** CG not converged and $k < k_{\max}$ **do**
4: Client assigns CG tasks to s servers and waits until tasks have completed;
5: Client repartitions work if significant change in workload and/or computational resources has occurred;
6: Set $k = k + 1$;
7: **end while**
8: Client reads final answer from IBP depot;

being performed on the intended server, in accordance with our partitioning. Unfortunately the fault–tolerance mechanism within the original GridSolve is now being circumvented, because the tasks cannot be resubmitted to another server should a task fail.

Algorithm 2.1 shows the general resource–aware CG algorithm. Note that the tasks use DSI file handles to manipulate the vectors on the IBP depot (component 4). The specific structure of the CG tasks will be discussed in the next section. After each iteration step the client may decide to repartition the work and assign the work to different computational servers in the network accordingly.

2.2.3 Partitioning algorithm and CG schemes

The matrix originating from the 2D discretisation of a Poisson problem on a structured grid is a block tridiagonal matrix. This matrix has a similar structure to the 3D discretisation of our test problem. In both cases, the block–rows roughly contain the same number of non–zeros, so we chose a simple non–homogeneous block–row partitioning. For illustration purposes, Figure 2.1 shows a heterogeneous partitioning using four servers on a 8×8 grid for a 2D Poisson problem. The partitioning of the block pentadiagonal matrix from the corresponding 3D problem is performed similarly.

The input and output vectors, shown at the top and left side respectively, are distributed in the same manner. More specifically, the effective speed of s servers is calculated using (1.8), together with the total flop count of a single CG iteration step. The size of each task is then determined accordingly.

The standard preconditioned Conjugate Gradient method was implemented first and is shown in Algorithm 2.2 (cf. Algorithm 1.2 on page 10). There are two natural synchronisation barriers, namely the two inner products for computing α and ρ. Note that synchronisation consists of three separate steps. First, at the end of a subtask the relevant data is updated on the depot (e.g., line 5). The subtask then returns the partial inner product to the client (line 6). Finally, at the start of the next subtask, it reads the relevant data from the depot (line 7). As a result, the depot contains a full copy of the vectors $\mathbf{x}, \mathbf{r}, \mathbf{z}, \mathbf{p}$, and $\mathbf{q}$ at all times. This ensures that in case of a server failure during an iteration step, all essential data are preserved on the IBP depot and the iteration process may continue (possibly at a later time) without any problem.

The scheme as it is depicted in Algorithm 2.2 suggests that there is an additional synchronisation point at line 23. By simply rearranging terms this can be avoided. Note that in the context of Grid computing and iterative methods, the number of synchronisation points should be kept to an absolute minimum. In our case, synchronisation does not only involve returning the partial inner products to the client, but also the (expensive) transfer of large vectors to and

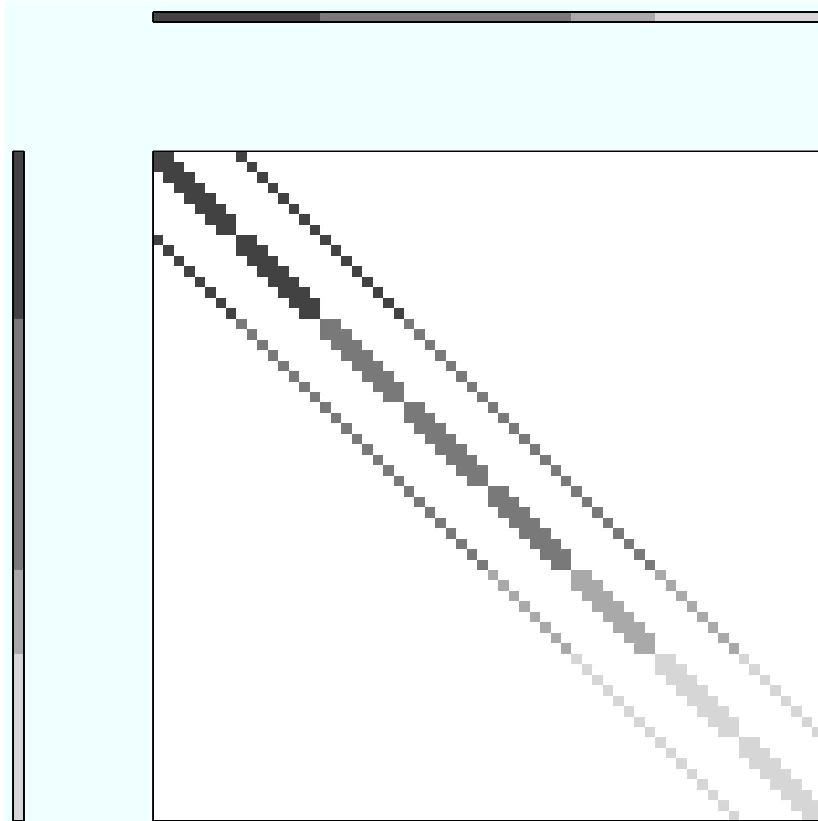


Figure 2.1: Heterogeneous block–row partitioning for $s = 4$ and $k = 8$ for a 2D Poisson problem.

from the depot. Therefore, this rearrangement of terms is neither trivial nor futile.

To increase the granularity we have also implemented the Chronopoulos–Gear variant of preconditioned CG [37, 56], which has a single synchronisation point, see Algorithm 2.3. This scheme introduces an additional $2n$ flops in each iteration step compared to the original scheme. Generating the matrix resulting from discretising the Poisson equation with varying density requires a significant amount of computation. Since this matrix has to be regenerated in each iteration step, this increases the granularity even further.

Preconditioning The efficiency of iterative methods depends heavily on the quality of the preconditioner, especially for very large systems. In parallel computing, efficient parallelisation of a sophisticated preconditioner is a difficult problem, but becomes even more difficult in the context of Grid computing. We therefore chose two traditional preconditioners that do not require any communication in the parallel context, which are Jacobi (diagonal scaling) and block Jacobi. For block Jacobi, the subdomains are solved accurately using standard CG with Incomplete Cholesky preconditioning.

Algorithm 2.2 Preconditioned CG; Task i with three subtasks.

INPUT: File handles to vectors $\mathbf{x}, \mathbf{r}, \mathbf{z}, \mathbf{p}, \mathbf{q}$ on IBP data depot, parameter k, preconditioner $\mathbf{B}$.
OUTPUT: Approximation $\mathbf{x}^i$.
1: *// Each server i performs the following:*
2: Read $\mathbf{r}^i$ from the depot
3: Solve $\mathbf{z}^i$ from $\mathbf{B}\mathbf{z}^i = \mathbf{r}^i$
4: Compute $\rho^i = (\mathbf{r}^i, \mathbf{z}^i)$
5: Update $\mathbf{z}^i$ on the depot
6: –SYNCHRONIZE– (Client sums ρ^i)
7: Read $\mathbf{z}^i$ and $\mathbf{p}^i$ from the depot
8: **if** $k = 1$ **then**
9: Set $\mathbf{p}^i = \mathbf{z}^i$
10: **else**
11: Set $\beta^i = \rho/\rho_{\text{old}}$
12: Set $\mathbf{p}^i = \mathbf{z}^i + \beta^i \mathbf{p}^i$
13: **end if**
14: Compute $\mathbf{q}^i = \mathbf{A}\mathbf{p}^i$
15: Compute $\mu^i = (\mathbf{p}^i, \mathbf{q}^i)$
16: Update $\mathbf{p}^i$ and $\mathbf{q}^i$ on the depot
17: –SYNCHRONIZE– (Client sums μ_i)
18: Compute $\alpha = \rho/\mu$
19: Read $\mathbf{x}^i, \mathbf{r}^i, \mathbf{p}^i$, and $\mathbf{q}^i$ from the depot
20: Set $\mathbf{x}^i = \mathbf{x}^i + \alpha\mathbf{p}^i$
21: Set $\mathbf{r}^i = \mathbf{r}^i - \alpha\mathbf{q}^i$
22: Update $\mathbf{x}^i$ and $\mathbf{r}^i$ on the depot
23: Check convergence; continue if necessary
24: Clients sets $\rho_{\text{old}} = \rho$

2.2.4 Implementation details

In this section we will discuss some specific issues concerning the various implementations. In the normal operation of GridSolve in combination with DSI, if an input parameter of a task is a DSI file handle, the middleware automatically retrieves the relevant data from the IBP depot before the task is started on the server. For our purposes a task needs full control over a DSI file, so instead we pass the DSI file handle explicitly.

Also, in the current implementation of IBP, reading and writing from and to the IBP depot are blocking operations [155]. Although read operations by different tasks can be performed on the same DSI file concurrently, write operations cannot, even when the write regions do not overlap. In the Chronopoulos and Gear scheme a task has to perform six write operations sequentially. Hence if a single DSI file is used to store data, large communication imbalance may occur in this case. We try to overcome this imbalance by using separate DSI files for each vector and letting each task update the vectors in a random order. By using the DSI functionality, it is also theoretically possible to interrupt the CG iteration process and restart at a later date, using possibly different computational resources.

At the end of each iteration step of the Chronopoulos and Gear variant, it may happen that DSI data is inadvertently overwritten. Specifically, we cannot guarantee that every task has finished reading the data from the previous iteration before other tasks have updated the new data. We therefore use two different DSI files representing the previous and current data and

Algorithm 2.3 Preconditioned CG; Chronopoulos and Gear variant; Task i.

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

let the client swap the corresponding file handles at the end of each iteration step.

Furthermore, each server node in our experimental setup has ATLAS [146] as a BLAS implementation which is used for the various `axpy` and inner product operations. Each task recomputes its portion of the sparse coefficient matrix every iteration step and stores it using the incremental compressed row storage (ICRS) format. The implementation of this format in C is somewhat faster than that of CRS [25].

To avoid the additional overhead of communicating matrix elements, we used a matrix–free approach. This is naturally suited for linear systems with specific classes of coefficient matrices, such as Poisson and Toeplitz matrices.

2.3 Numerical experiments

2.3.1 Overview

In the previous sections we discussed several implementations and various suggestions for increasing the granularity. In this section we will perform numerical experiments and investigate the effect of these suggestions on the performance. The experiments are divided into three parts. In the first part we investigate the difference between the standard CG method and the Chronopoulos and Gear variant. Also, we experiment with some features of the DSI mechanism. The implementation with the best results is then used for the remainder of the experiments. The second part describes experiments in a heterogeneous computing environment and in the last part we conduct overall performance experiments using two types of preconditioning.

The residual at iteration step k is defined as $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$. As the starting vector we take $\mathbf{x}_0 = \mathbf{0}$ and we use the termination criteria $||\mathbf{r}_k||_2/||\mathbf{b}||_2 < 10^{-6}$.

In parallel and distributed computing, speed–up can be investigated by solving a problem of fixed size using an increasing number of nodes (i.e., *strong scaling*). In the context of Grid computing, it is more natural to fix the problem size *per server* and investigate the scalability of the algorithm by adding more servers in order to solve bigger problems (i.e., *weak scaling*). We will analyse the algorithm using both approaches.

2.3.2 Target hardware

In order to properly investigate the effectiveness of the proposed algorithm on Grid hardware, two different Grid testbeds are used: a local non–dedicated cluster with varying workloads and a dedicated multi–cluster of geographically separated clusters.

The first testbed is a local cluster of computers, which is a multi–user system consisting of nodes with different processors and dynamic workloads. The servers in the network are ten single core (AMD Athlon 64 Processor 3700 at 2.4GHz) and two dual core CPU nodes (Intel Core 2 CPU 6700 at 2.66GHz) with 3 GB and 8 GB of memory respectively and running Linux 2.6. For more details on the local cluster, see Section 1.8.1 on page 19.

In some cases we aim to perform controlled and repeatable experiments so we use idle processors and as a result the partitioning is fixed and homogeneous throughout these experiments. In most of the experiments we measure the wall clock times of five CG steps for different values of n.

The second testbed is the Distributed ASCI Supercomputer 3 (DAS–3), which is a cluster of five geographically separated clusters spread over four academic institutions in the Netherlands [107]. 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 and high speed Myri–10G interconnect. The TUD site only employs Gigabit Ethernet interconnect. Although each separate cluster is relatively homogeneous, the system as a whole can be considered heterogeneous. For more details on the DAS–3, see Section 1.8.2.

On both testbeds, the client, the IBP server, and the agent are running on the same node while the servers are started on the remaining nodes. In a typical Grid environment the client program would be located on the users' desktop machine. On the local cluster the depot is started on a randomly chosen node, while on the DAS–3 the depot is located on the head node of the VU site. As a result we expect significant communication overhead.

2.3.3 Preliminary testing

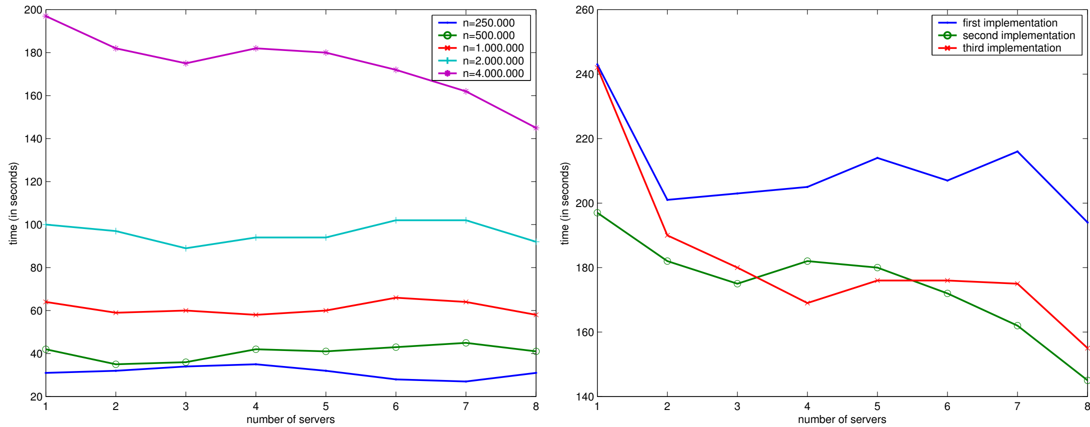
In these set of experiments on the local cluster we differentiate between the following three implementations:

(i). Standard preconditioned CG using a single DSI file;

(ii). Chronopoulos and Gear scheme using a single DSI file; and

(iii). Chronopoulos and Gear CG using six separate DSI files for each vector which are manipulated in a random order.

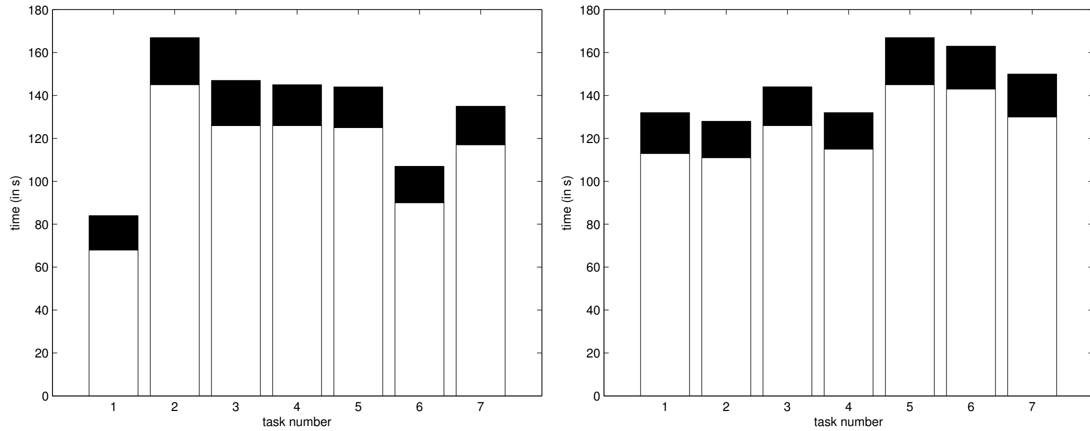
Jacobi preconditioning is used in every experiment and we fix the number of CG iterations to five. Figure 2.2(a) shows the total wall clock time of the second implementation for different values of n using up to eight servers. It also demonstrates that communication overhead is particularly an issue for small n, which is hardly surprising. In this case using more servers does



(a) Different problem sizes, implementation **(ii)**, up to eight servers.

(b) $n = 4 \times 10^6$, all three implementations, up to eight servers.

Figure 2.2: Wall clock times of CG implementations in GridSolve on the local cluster.



(a) Utilising a single DSI file.

(b) Utilising multiple DSI files.

Figure 2.3: Breakdown of wall clock time of tasks in communication (bottom part) and computation, for $n = 4 \times 10^6$ and using seven servers.

not result in improved execution times, and even results in larger wall clock times due to the communication overhead. For large n this implementation performs slightly better. The other implementations give similar results for small n and we will therefore concentrate on results for large systems.

In Figure 2.2(b) results are given of the three different implementations for $n = 4 \times 10^6$. Here we clearly see the improved performance of the Chronopoulos and Gear variants for a large number of servers. In other words, our attempts to improve the granularity resulted in speed–up, albeit modest. Nevertheless, in the context of Grid computing these are encouraging results.

Although we observe that using separate DSI files for the vectors only improved the overall running time of the Chronopoulos and Gear scheme for some number of servers, we note that in this case the tasks finish at roughly the same time, in contrast to the case of using a single DSI file. This is illustrated in Figure 2.3 where the wall clock times of the separate tasks for seven servers are shown after five CG steps of Chronopoulos and Gear, broken down in computation and communication. Note that Figure 2.3(b) shows that by using separate DSI files the communication becomes more balanced, which is an encouraging result. Unfortunately, the total wall clock time is not reduced.

When using a single DSI file, Figure 2.3(a) reveals unbalanced wall clock times for each task, which can be explained as follows. Although the client initiates a sequence of non–blocking calls, at the end of (in particular) the first task the updates to the DSI file appears to block subsequent updates by other tasks. These figures also clearly reveal the amount of communication overhead.

For the remainder of the experiments we will use the third implementation, which uses separate DSI files for each vector.

2.3.4 Heterogeneous environment

In this section we perform heterogeneous experiments on the two testbeds. On the local cluster we artificially simulate workload, which affects the partitioning of the computational work. On the DAS–3 no workload is simulated, but the differences in processor speed has a similar effect on the partitioning.

It is not trivial to perform repeatable experiments in a heterogeneous computing environment. Instead, we will give a brief qualitative analysis of the processes involved and present a typical execution of the resource–aware partitioning scheme and its effect on the CG iteration process.

On the local cluster, we artificially simulate varying workload by running a special process on each server. This process alternates between repeatedly performing a large matrix–vector multiplication and idling for a random number of seconds.

Note that the current resource–aware partitioning scheme is incompatible with block Jacobi preconditioning, because in this case the work done by each server is disproportional to the number of rows. As a result, Jacobi preconditioning is used for these experiments.

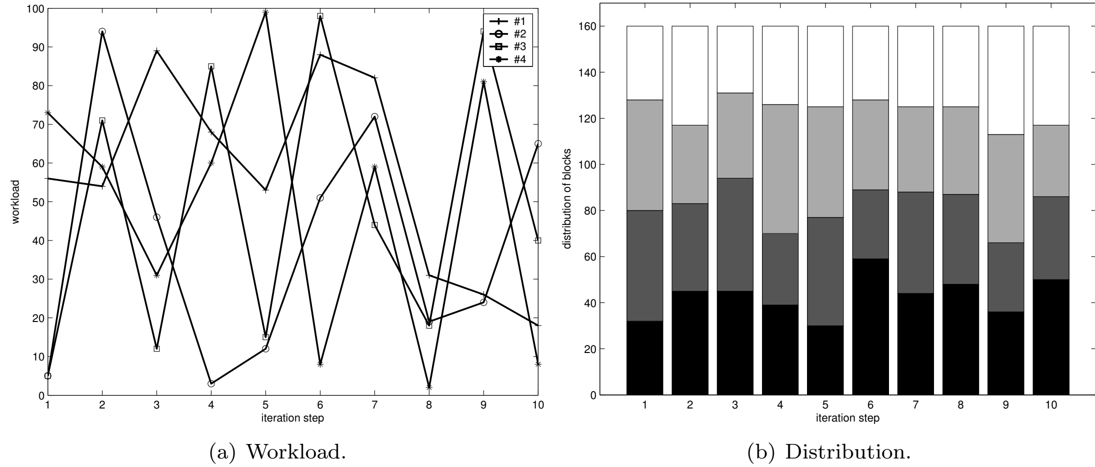
We fix the number of servers to four with approximately one million equations per server. In Figure 2.4(a) the workload of each server is shown at the beginning of each iteration step as observed by the GridSolve agent. Figure 2.4(b) shows the corresponding distribution of the matrix and vector row blocks, where the tasks are numbered from top to bottom.

The graphs clearly show the effect of the varying workloads on the distribution. For example, at the sixth iteration step, server number four is only slightly occupied and as a result task number four has the largest size.

To investigate the effect of the partitioning strategy on the execution time of the algorithm in an artificial heterogeneous computing environment, we measured the wall clock times of five iteration steps using either a homogeneous partitioning or a heterogenous partitioning.

However, extensive experiments showed that the latter strategy only had a moderate effect on the total execution time. Due to the low computation to communication ratio the current partitioning algorithm mostly affects the amount of communication of each task with the depot. As processor workload barely has influence on these kind of operations, the *total* execution time does not decrease significantly when using a heterogeneous partitioning on heterogeneous computational resources.

There are two possible solutions within the current context. Either incorporate network bandwidth information in the partitioning algorithm, or try to increase the computation to



(a) Workload. (b) Distribution.

Figure 2.4: Heterogeneous experiments with the 3D bubbly flow problem (local cluster).

communication ratio in combination with an appropriate computational resource–aware partitioning strategy.

Since the DAS–3 is a dedicated machine, the nodes have zero workload. Therefore, the partitioning on the DAS–3 is based solely on the heterogeneity in the processor speeds and is fixed throughout the whole iteration process. Not surprisingly, the effect on the wall clock times of five iteration step is similar to that of the local cluster results.

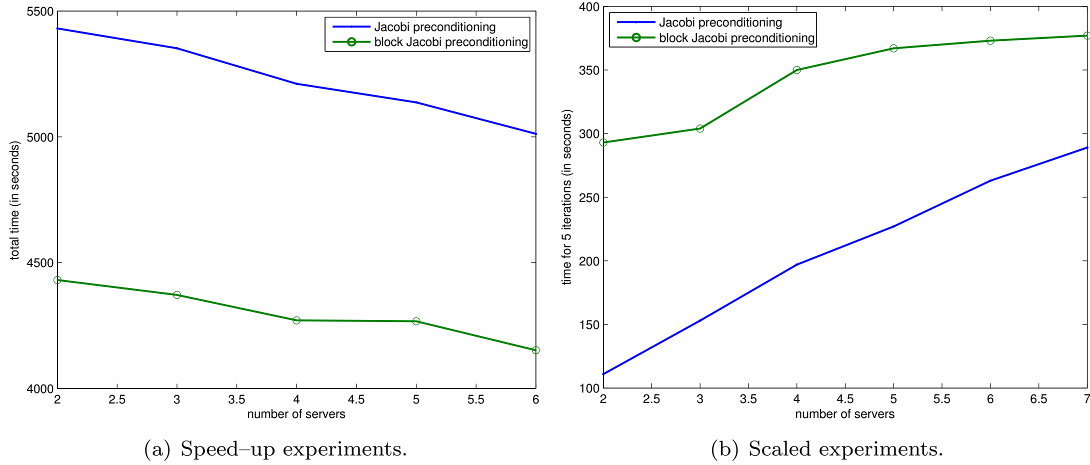
2.3.5 Parallel performance

In the previous sections, we investigated various aspects of the algorithm separately. In this section, we present overall parallel performance results using two preconditioning techniques on both the local cluster and on the DAS–3, *without any workload*. First, we fix the problem size to $n = 120^3$ and investigate speed–up using up to six servers on the local cluster. Figure 2.5(a) shows the total wall clock time until convergence is obtained. In Figure 2.6(a) speed–up results are given on the DAS–3 for $n = 25^3$. A server is started on a randomly chosen node on each cluster of the DAS–3. The client is located on the head node of the VU site.

Using a more sophisticated preconditioning technique like block Jacobi improves the computation to communication ratio and reduces the total number of CG iterations. However, it also negatively influences the manner in which the total number of CG iterations depends on the number of subdomains. This is in contrast to using diagonal scaling as a preconditioner, where the total number of iterations is independent of the number of subdomains.

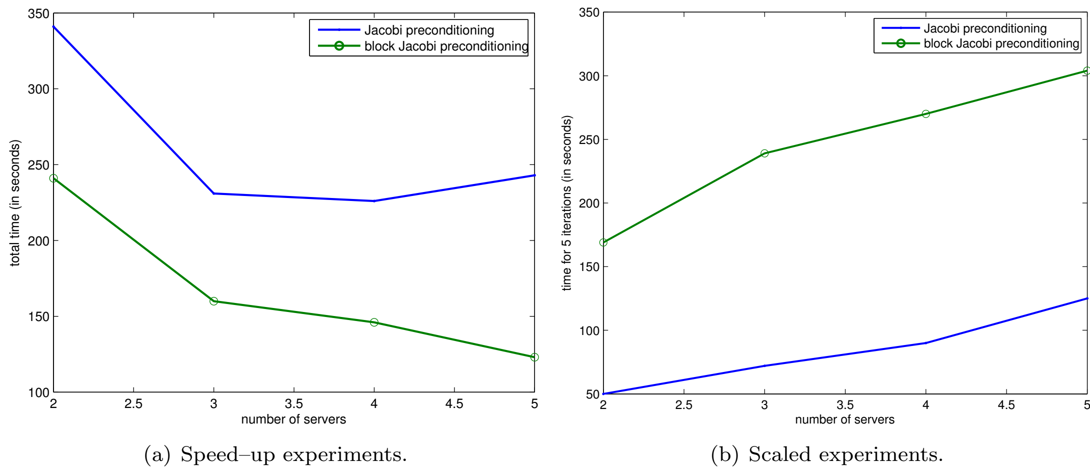
We investigate the scalability of the algorithm by setting the problem size per server to 1,000,000 equations and performing *five CG steps* using both Jacobi and block Jacobi as a preconditioner. This experiment gives an indication on how fast the communication overhead grows. The results for the local cluster are given in Figure 2.5(b), while Figure 2.6(b) shows results for the DAS–3.

Using Jacobi preconditioning results in an unfavourable computation to communication ratio and the results show that (communication) overhead grows quite fast, which is not a surprising result. On the other hand, the block Jacobi preconditioning has a more favourable ratio, which is indicated by the reduced increase in execution time.



(a) Speed–up experiments. (b) Scaled experiments.

Figure 2.5: Comparison of two preconditioning techniques (local cluster).



(a) Speed–up experiments. (b) Scaled experiments.

Figure 2.6: Comparison of two preconditioning techniques (DAS–3).

2.4 Concluding remarks

2.4.1 Conclusions

The efficient iterative solution of large sparse linear systems on aggregated computational resources is a difficult problem. While the parallel implementation of iterative methods in the context of dedicated parallel computing is relatively well–understood, the design of efficient iterative algorithms for the solution of large linear systems on non–dedicated and heterogeneous networks of computers is still in its infancy and much research is needed.

The key algorithmic constraint of CG methods in Grid computing is the inner product. To be more specific, the computation of an inner product in parallel iterative algorithms induces a

global synchronisation point. While such an operation can be complex even on dedicated and homogeneous parallel computers connected by a high–speed network, it may become the critical bottleneck in a non–dedicated and heterogeneous computing environment. For example, within the context of GridSolve the non–persistent data model forces us to transfer at each synchronisation point large amounts of data over a possibly unreliable and slow network connection.

In this work we have described two implementations of the preconditioned Conjugate Gradient method using the mature Grid middleware GridSolve. We have evaluated the implementations on heterogeneous computing hardware, applied to a realistic test problem. Using the middleware we have also implemented a simple architecture–aware partitioning algorithm to divide the computational work. Furthermore, by using multiple DSI files we have attempted to decrease the communication overhead within the bridge communication model currently used by GridSolve. And finally, we have increased the granularity by (i) using the Chronopoulos and Gear variant of CG which only has a single synchronisation point per iteration step and (ii) by devoting more work to the preconditioning phase.

We have explored the combination of Grid computing and iterative methods within the software constraints of the Grid middleware GridSolve. The main bottlenecks — in both middleware and iterative method — were identified and algorithmic and software modifications for improving the granularity were proposed and implemented, resulting in moderate improved performance and speed–up. Although the experimental results were less than optimal, they can be considered encouraging in the context of iterative solvers and Grid computing.

2.4.2 Suggestions for improvements

Naturally, there is room for improvement and we will give some suggestions.

The current implementation of GridSolve forces us to use bridge communication. SmartGridSolve [29] is an extension of GridSolve, which performs similarly to SmartNetSolve [30], allowing for communication between the computational servers as well as data persistence. By combining this with sophisticated (possibly weighted) hypergraph partitioning techniques such as used in *Mondriaan* [142] we hope to greatly improve our load balancing algorithm. Another possible improvement is incorporating information about network throughput into the partitioning algorithm.

Furthermore, possible hardware and software solutions to reduce communication overhead include fast network connections to the IBP depot and using distributed IBP data depots [21].

The local subdomains in the block Jacobi preconditioner are solved accurately using another iterative method. In [31] interesting results have been obtained with inaccurate subdomain solutions. Applying this same strategy to our application would require us to use a method that can handle a variable preconditioner, such as the flexible CG method [6]. As shown in the next chapters, efficient parallelisation of such a method on Grid computers introduces additional difficulties.

2.4.3 General remarks

In metacomputing, using the appropriate middleware is of critical importance. Many different types of Grid middleware exists and choosing the correct middleware depends on the application, target hardware, and the (numerical) algorithm. The main reason for choosing GridSolve to solve the current application is two–fold: (i) the GridSolve middleware is specifically targeted to numerical computations and (ii) GridSolve allows for easy access to remote computational resources.

In this chapter we have tried to realise the full potential of a *completely synchronous* parallel subspace method for solving large sparse linear systems on Grid computers. Synchronous parallel iterative algorithms are methods where at each iteration step information is needed from the previous iteration step. As shown previously and as exemplified by the experimental results, the fine–grain nature and potentially large number of synchronisations of said methods raises many efficiency issues on Grid computers and limits the applicability of this approach to large–scale problems.

An efficient and effective preconditioner is crucial for fast convergence of iterative methods. Such a preconditioner is generally speaking the most difficult part to parallelise, especially in heterogeneous environments as found in Grid computing. Within the fully synchronous context considered in this chapter, we have maximised the amount of work that can be devoted to the employed preconditioning, *without* introducing additional synchronisation points.

Parallel *asynchronous* iterative algorithms exhibit features that are extremely well–suited for Grid computing, such as lack of synchronisation points and coarse–graininess. Unfortunately, they also suffer from slow (block Jacobi method–like) convergence rates. We propose using said asynchronous methods as a coarse–grain preconditioner in a flexible subspace method, where the preconditioner is allowed to change in each iteration step. By combining a slowly converging asynchronous inner method and a fast converging synchronous outer method, we aim to reap the benefits and awards of both techniques.

Desynchronising 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. The resulting partially asynchronous inner–outer method is investigated extensively in the next chapters, with highly promising experimental results.

Chapter 3

Asynchronous Iterative Methods as Preconditioners

This chapter has been published as:

T. P. Collignon and M. B. van Gijzen. Fast iterative solution of large sparse linear systems on geographically separated clusters. *International Journal of High Performance Computing Applications*, 2011. (to appear).

T. P. Collignon and M. B. van Gijzen. Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. In G. Kreiss, P. Lötstedt, A. Målqvist, and M. Neytcheva, editors, *Numerical Mathematics and Advanced Applications 2009: Proceedings of ENUMATH 2009, the 8th European Conference on Numerical Mathematics and Advanced Applications, Uppsala, July 2009*, pages 261–268. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

Part I	Part II
CRAC	GridSolve
coupled iteration processes	decoupled iteration processes
dedicated cluster of clusters (DAS–3)	non–dedicated local clusters
2D and 3D bubbly flows	3D convection–diffusion
symmetric linear systems	nonsymmetric linear systems
flexible CG (plus deflation, see Part III)	GMRESR

Table 3.1: Differences in computational setting.

Overview

This chapter investigates experimentally an efficient iterative algorithm for solving large sparse linear systems on Grid computers. 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. Section 1.4 on page 11 contains a detailed description of this approach. This chapter consists of three parts.

In the first part (Section 3.1–3.3), the asynchronous preconditioner is combined with the flexible Conjugate Gradient (CG) method in order to solve on the DAS–3 multi–cluster large linear systems originating from large 2D and 3D bubbly flow problems. Both the outer iteration and the inner iteration are performed using the same (dedicated) computational resources, resulting in a *coupled* iteration approach. The algorithm is implemented using the CRAC middleware, which is specifically designed for easy implementation of (partially) asynchronous iterative algorithms on a cluster of (possibly geographically separated) clusters.

In the second part (Section 3.4–3.6), an asynchronous iterative method is used as a preconditioner in the flexible method GMRESR to solve large linear systems originating from a 3D convection–diffusion problem. In this case, the outer iteration is performed on stable and homogeneous computational resources, while the inner asynchronous iteration is performed on volatile, heterogeneous, and non–dedicated computational resources. This *decoupled* iteration approach is implemented using the GridSolve middleware, resulting in an adaptive and a partially fault–tolerant algorithm. For more discussion on this coupled/decoupled approach, see Section 1.9 on page 21.

In the final part (Section 3.7–3.11), the asynchronous preconditioning iteration of flexible CG from the first part is combined with a deflation step. It is shown that this results in increased robustness of the total algorithm. In addition, a short experimental study is performed in Section 3.10 on using asynchronous iterative methods as smoothers. Section 3.11 contains the general conclusions of this chapter.

For the readers' convenience, Table 3.1 presents a short overview of the differences in computational setting of part I and II.

Part I: Coupled iterations

3.1 Introduction

In this part the partially asynchronous method is implemented using a coupled iteration approach.

Algorithm 3.1 Flexible Conjugate Gradients (pure truncation strategy)

INPUT: Parameters $m_{\max}, \varepsilon_{\text{in}}, T_{\max}$; Set $m_k = \min(k, m_{\max})$; Initial guess $\mathbf{x}_0$; Set $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
OUTPUT: Approximate solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$.

1: **for** $k = 0, 1, \ldots,$ until convergence **do**
2: Evaluate $\mathbf{u} = \mathcal{B}(\mathbf{r}_k, \varepsilon_{\text{in}}, T_{\max})$; // *Preconditioning step: Algorithm 3.2*
3: Compute $\mathbf{u}_k = \text{ortho–mgs}(\mathbf{u}, \mathbf{c}_i, \mathbf{u}_i, k, m_k)$; // *Orthogonalisation step: Algorithm 3.3*
4: Compute $\mathbf{c}_k = \mathbf{A}\mathbf{u}_k$; // *Matrix–vector multiplication*
5: Compute $\alpha_k = \dfrac{\mathbf{u}_k^* \mathbf{r}_k}{\mathbf{u}_k^* \mathbf{c}_k}$;
6: Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{u}_k$;
7: Update $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{c}_k$;
8: **end for**

The target hardware consists of the DAS–3 Grid computer, 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 [128]. This application involves the solution of large sparse symmetric and positive definite systems, which leaves the flexible Conjugate Gradient method [6, 96] as the method of choice for the outer iteration.

The algorithm is implemented using the CRAC library, which was developed within the GREMLINS project [47, 46]. 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. For a more detailed description of CRAC, see Section 1.7.2 on page 18.

The experimental results on the DAS–3 multi–cluster demonstrate that the proposed algorithm is very 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 load varies heavily.

3.2 Parallel implementation details

Listed in Algorithm 3.1 is the flexible CG method using a pure truncation strategy. 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. In the following, the implementation of these phases will be discussed.

3.2.1 Asynchronous preconditioning

In standard preconditioned CG, the preconditioner is a fixed symmetric and positive definite matrix $\mathbf{B}$ such that solving the residual equation $\mathbf{B}\mathbf{u} = \mathbf{r}$ is “cheap” in some sense. In the proposed algorithm, the preconditioning operation in line 2 of Algorithm 3.1 consists of an asynchronous iterative method applied to the system $\mathbf{A}\mathbf{u} = \mathbf{r}_k$ 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

Algorithm 3.2 Asynchronous block Jacobi iteration for CRAC task i.

OUTPUT: $\mathbf{u}_i = \mathcal{B}(\mathbf{r}_i, \varepsilon_{\text{in}}, T_{\max})$

1: Wait until $\mathbf{r}_i$ is updated; Set $\mathbf{u}_i = \mathbf{0}$;
2: Perform IC(0) decomposition of $\mathbf{A}_{ii}$;
3: **while** $t_{\text{elapsed}} < T_{\max}$ **do**
4: Compute $\mathbf{v}_i = \mathbf{r}_i - \sum_j \mathbf{A}_{ij}\mathbf{u}_j$;
5: Solve $\mathbf{A}_{ii}\mathbf{p}_i = \mathbf{v}_i$ with accuracy ε_{in};
6: Update $\mathbf{u}_i = \mathbf{u}_i + \mathbf{p}_i$;
7: Exchange $\mathbf{u}_i$ asynchronously with neighbours;
8: **end while**

nonlinear) process,

$$\mathbf{u} = \mathcal{B}(\mathbf{r}_k, \varepsilon_{\text{in}}, T_{\max}), \qquad \mathcal{B} : \mathbb{R}^n \to \mathbb{R}^n, \tag{3.1}$$

which varies from one iteration step k to the next. In Algorithm 3.2 the specific steps are shown that are performed by the asynchronous preconditioning iteration processes (cf. Algorithm 1.1 on page 8).

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 performed in the outer iteration.

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

Note that the "standard" block Jacobi preconditioner corresponds to a *single* iteration step of the synchronous block Jacobi iteration from Algorithm 1.1 on page 8 with initial guess $\mathbf{x}^{(0)} \equiv \mathbf{0}$. In contrast, the preconditioning step from (3.1) consists of *multiple* asynchronous block Jacobi "iterations steps". These two types of preconditioning will be compared in the numerical experiments. Also note that using the term "iteration step" for an asynchronous iterative method is a slight abuse of language.

3.2.2 Orthogonalisation

The main difference with standard preconditioned CG is the additional orthogonalisation step in line 3 of Algorithm 3.1. The newly obtained search direction vector $\mathbf{u}$ is orthogonalised with respect to the $\mathbf{A}$–inner product (i.e., $\langle \mathbf{x}, \mathbf{y} \rangle_{\mathbf{A}} \equiv \mathbf{x}^*\mathbf{A}\mathbf{y}$) against a number of previous search directions.

For practical implementations of flexible methods a truncation or restart strategy has to be applied. In this chapter 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 [96].

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. The hope is that a low truncation parameter $m_{\max}$ is sufficient, thus keeping the number of expensive synchronisations to a minimum.

Algorithm 3.3 Modified Gram–Schmidt.

OUTPUT: $\mathbf{u}_k = \text{ortho–mgs}(\mathbf{u}, \mathbf{c}_i, \mathbf{u}_i, k, m_k)$;

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

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 [48, 26].

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 Algorithm 3.3 for our case.

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

3.3 Numerical experiments

3.3.1 Motivation

For completeness, we reproduce here the bubbly flow problem from Section 2.2.1 on page 32. Note that 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 [141] to solve the governing equations for bubbly flows on a very fine 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,

$$\begin{cases} -\nabla \cdot \left(\frac{1}{\rho(\mathbf{x})} \nabla p(\mathbf{x}) \right) = f(\mathbf{x}), & \mathbf{x} \in \Omega, \\ \frac{\partial}{\partial \mathbf{x}} p(\mathbf{x}) = g(\mathbf{x}), & \mathbf{x} \in \partial\Omega, \end{cases} \tag{3.2}$$

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 chapter the test problem from [135, 129] 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

$$\rho(\mathbf{x}) = \begin{cases} 1, & \mathbf{x} \in \Gamma_0; \\ \tau, & \mathbf{x} \in \Gamma_1, \end{cases} \tag{3.3}$$

FCG($m_{\max}$)	wall clock time (sec.)	iterations	number of 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 3.2: Influence of parameter m ($T_{\max} = 5$s, five nodes, 2D problem).

where typically $\tau = 10^{-3}$. Such a discontinuity in the coefficient results in an ill–conditioned linear system, making it a difficult problem for iterative methods. For the purpose of this chapter 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 [135].

Both 2D and 3D experiments will be performed. Applying standard finite differences to (3.2) on a structured $n_x \times n_y$ or $n_x \times n_y \times n_z$ mesh results in the linear system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{n \times n}$ 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 $\mathbf{x}$ is determined up to a constant. It can be shown that this does not pose any problems for the iterative solver [128].

3.3.2 Target hardware and experimental setup

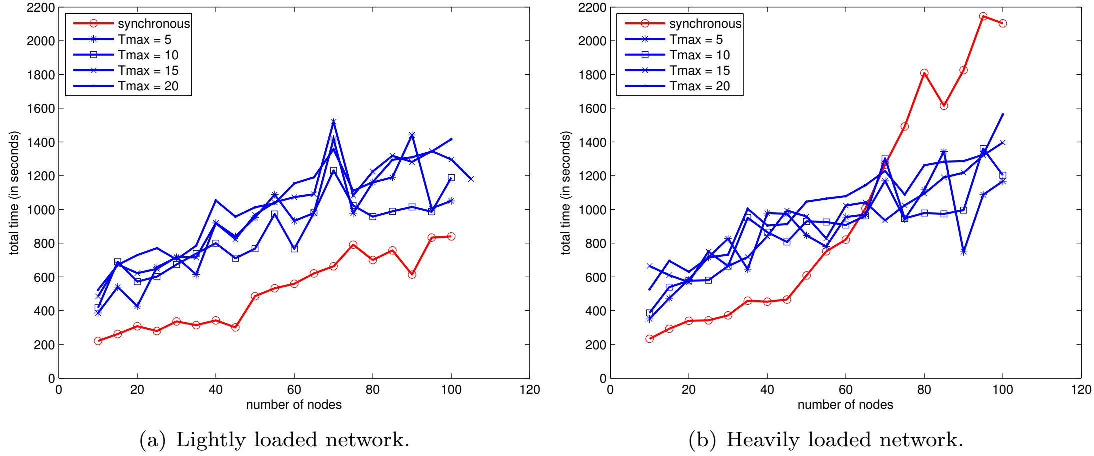
The Distributed ASCI Supercomputer 3 (DAS–3) is a multi–cluster consisting of five clusters, located at four academic institutions across the Netherlands [107]. The five sites are connected through SURFnet, which is the academic and research network in the Netherlands. Four of the five local clusters are equipped with both Gigabit Ethernet interconnect and high speed Myri–10G interconnect. The TUD site only employs Gigabit Ethernet interconnect. For more information on the DAS–3, see Section 1.8.2 on page 21. 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. Since a coupled iteration approach is used, it is natural to use the same data distribution for both iteration processes. For more background on the employed data partitioning, see Section 1.10.1 on page 22.

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 $\varepsilon_{\text{in}} = 10^{-1}$ using standard CG preconditioned with an Incomplete Cholesky decomposition (without fill–in, i.e., IC(0)) [102].

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 $\varepsilon_{\text{outer}} = 10^{-8}$.

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 (i.e., weak scalability). The nodes are evenly divided between the five clusters with increments of five nodes, starting with a single node on each cluster.



(a) Lightly loaded network.

(b) Heavily loaded network.

Figure 3.1: Total execution time (3D problem).

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

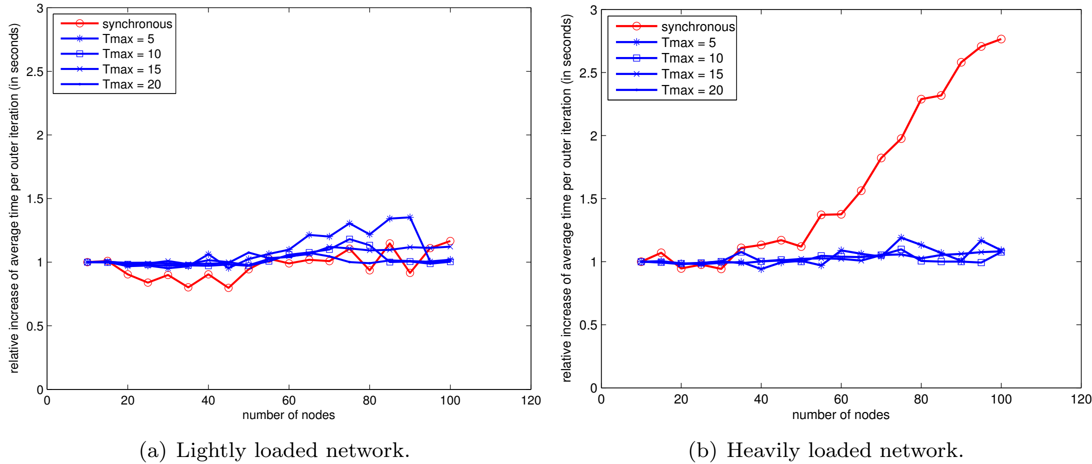
Since the DAS–3 is solely intended for research purposes, the maximum allowed time for a single job is sixty minutes. All the timing results shown are wall clock times. For comparison studies, fully synchronous preconditioning is also performed, which involves performing a single block Jacobi iteration step per preconditioning phase with zero initial guess. This corresponds to the standard block Jacobi preconditioner. The effectiveness of the asynchronous preconditioner depends on multiple (and random) factors, so these experiments are performed three times 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 Table 3.2. 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) is a Richardson iteration preconditioned with an asynchronous iterative method. In other words, this corresponds to the (for all intents and purposes) *completely* asynchronous Jacobi iteration and Table 3.2 shows that such a fully asynchronous method is impractical for this application. For $m_{\max} = 1$, the method corresponds to standard preconditioned CG, which also does not perform well. For $m_{\max} > 1$, the performance of the method starts to improve significantly. The results show 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.

3.3.3 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.



(a) Lightly loaded network.

(b) Heavily loaded network.

Figure 3.2: Relative increase of time per outer iteration step (3D problem).

2. Experiments using a 2D test problem and varying network load for showing that asynchronous preconditioning can outperform synchronous preconditioning independent of the amount of network activity.

3D experiments

Figure 3.1(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 3.1(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. Figure 3.2 shows the *relative increase* of the *average* times per outer iteration for both the single and the multi–cluster case. To be more precise, it shows the increase in time per iteration relative to the time per iteration on 10 nodes.

The results given in Figure 3.2(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.

As for the loaded network results, Figure 3.2(b) shows that the relative increase in time per

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

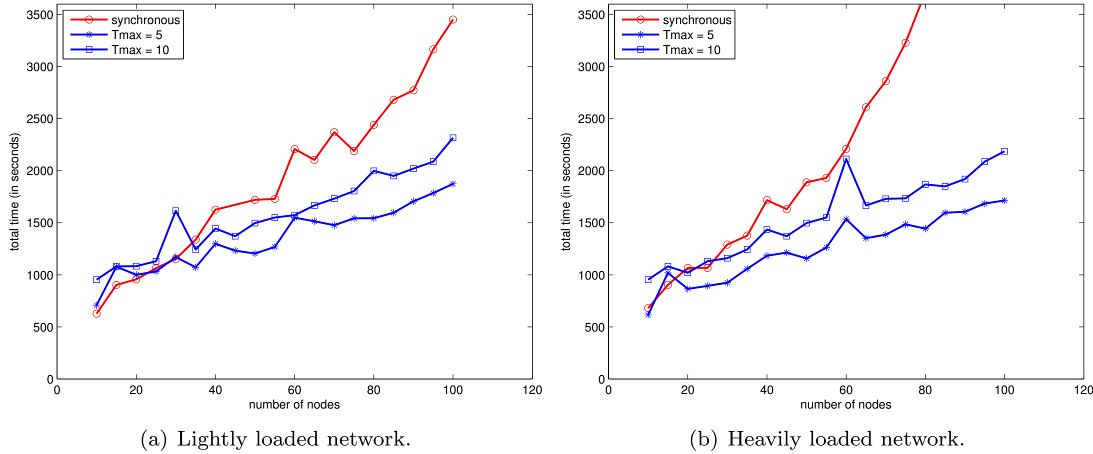
number of nodes	synchronous	async. (lightly loaded)	async. (heavily loaded)
10	286	77	76
20	601	91	96
30	805	139	120
40	1060	124	128
50	1388	130	134
60	1473	138	176
70	1881	151	157
80	1905	174	164
90	2082	175	162
100	2332	195	175

Table 3.4: Outer iterations for synchronous and asynchronous preconditioning (2D problem).

outer iteration for synchronous preconditioning is far greater than with asynchronous preconditioning.

(ii) Number of outer iterations Table 3.3 lists the total number of outer iterations for synchronous and asynchronous preconditioning with $T_{\max} = 15\text{s}$. 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 Figure 3.1(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.



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

Figure 3.3: Total execution time (2D problem).

2D experiments

In Figure 3.3 results are given using a 2D test problem for $T_{\max} \in \{5, 10\}$ on both a lightly loaded network and a heavily loaded network. Again, synchronous preconditioning is also included. Note that in this case there is less communication between the subdomains.

The numerical results show that even when the network is lightly loaded (i.e., Figure 3.3(a)), synchronous preconditioning is outperformed by asynchronous preconditioning when using more than 40 nodes. For 100 nodes, the total execution time for synchronous preconditioning is almost twice as long as for asynchronous preconditioning on a lightly loaded network. Not surprisingly, on a heavily loaded network the synchronous preconditioning performs even worse (i.e., Figure 3.3(b)). Similar to the 3D experiments, the effectiveness of the asynchronous preconditioning is practically unaffected by the increased network activity.

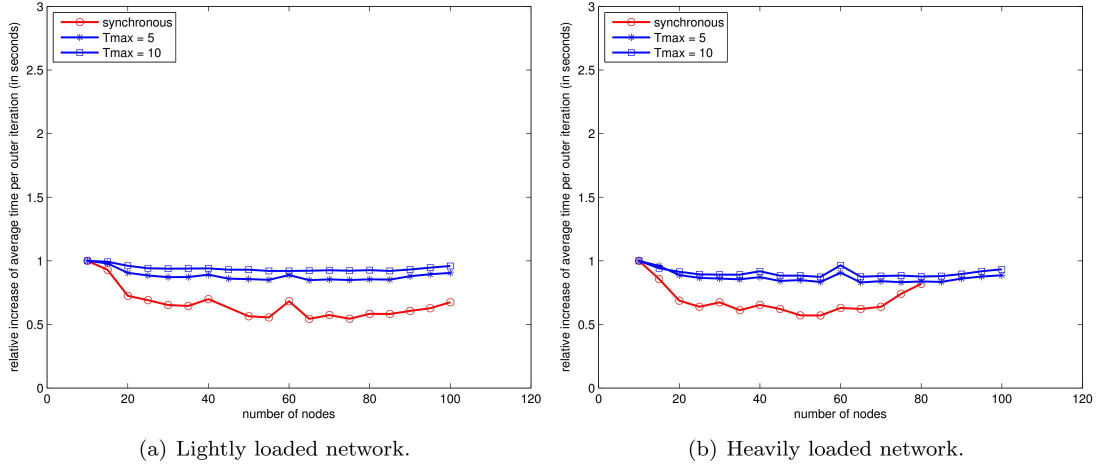
Similar to Figure 3.2, Figure 3.4 gives the relative "increase" in time per outer iteration step, where it can be seen that there is initially a *decrease* for synchronous preconditioning, followed by a small increase. We do not have a complete explanation for this behaviour. Despite the fact that synchronous preconditioning does not show a relatively large increase in time per outer iteration like in Figure 3.2(b), it is still outperformed by asynchronous preconditioning. This can be explained by examining the number of outer iterations for $T_{\max} = 10$s, which are shown in Table 3.4 for a lightly and heavily loaded network.

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 for both a lightly loaded and heavily loaded network. As a result, asynchronous preconditioning is also very effective for this test case.

3.3.4 Discussion

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

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



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

Figure 3.4: Relative increase of time per outer iteration step (2D problem).

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

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. 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.

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.

The experimental results also show that factors such as heterogeneity of the hardware and variations in network activity hardly seem to have any impact on the effectiveness of the preconditioner.

Part II: Decoupled iterations

3.4 Introduction

In this part the inner–outer algorithm described in Chapter 1 is implemented using the Grid middleware GridSolve [58, 154], which allows for convenient decoupling of the two iteration processes. The algorithm is a combination of the flexible iterative method GMRESR [137] and an asynchronous iterative method as preconditioner. The outer iteration is performed sequentially on the (stable) client machine, while the inner preconditioning iteration is performed on (unstable) heterogeneous computing hardware.

Since global synchronisation is an expensive operation, the bulk of the computational work is performed by the asynchronous preconditioning iteration. In this way, efficient use is made of the available computational resources. Purely for comparison purposes, we have also evaluated a parallel implementation of the outer iteration, resulting in a coupled iteration process as in the first part.

Algorithm 3.4 GMRESR (truncated version)

INPUT: Parameters $m, \varepsilon_{\text{in}}, T_{\max}$; Initial guess $\mathbf{x}_0$; Set $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$.
OUTPUT: Approximate solution to $\mathbf{Ax} = \mathbf{b}$.

1: **for** $k = 0, 1, \ldots,$ until convergence **do**
2: Evaluate $\mathbf{u} = \mathcal{B}(\mathbf{r}_k, \varepsilon_{\text{in}}, T_{\max})$; // *Preconditioning step: Algorithm 3.6*
3: Compute $\mathbf{c} = \mathbf{Au}$; // *Matrix–vector multiplication*
4: Compute $[\mathbf{c}_k, \mathbf{u}_k]$ = ortho–cgs$(\mathbf{c}, \mathbf{u}, \mathbf{c}_i, \mathbf{u}_i, k, m)$; // *Orthogonalisation step: Algorithm 3.5*
5: Compute $\gamma = \mathbf{c}_k^* \mathbf{r}_k$;
6: Update $\mathbf{x}_{k+1} = \mathbf{x}_k + \gamma \mathbf{u}_k$;
7: Update $\mathbf{r}_{k+1} = \mathbf{r}_k - \gamma \mathbf{c}_k$;
8: **end for**

We choose GMRESR as the flexible (outer) method, partly because the orthogonalisation process can be easily truncated, which is essential for practical implementations. The truncated variant of GMRESR is shown in Algorithm 3.4. The preconditioning step in line 2 consists of computing some approximate solution to $\mathbf{Au} = \mathbf{r}_k$ using an asynchronous iterative method. The obtained search direction is then orthogonalised in line 4 against m previous search directions.

Numerical experiments for a large 3D convection–diffusion problem demonstrate the effectiveness of the algorithm.

3.5 Parallel implementation details

3.5.1 Brief description of GridSolve

For completeness, we briefly repeat the key components of GridSolve. The middleware consists of the following components.

(i) The client, which can remotely execute tasks on computational servers using information provided by the agent.

(ii) The agent, which actively monitors server properties such as CPU speed, memory size, computational services, workload, and availability.

(iii) The computational servers, which can run predefined tasks. Any data that are read or generated locally during the execution of a task are lost after the task completes, unless the data are stored on the IBP data depot.

(iv) The IBP data depot, which acts as a storage device and is accessible by the client and the servers. The client uploads (downloads) data to (from) the IBP data depot which is in close proximity to the computational servers and tasks can then read (write) data from (to) the depot. Therefore, using the IBP data depot induces bridge communication between the client and the servers.

For a more detailed description of GridSolve, see Section 1.7.1 on page 16.

3.5.2 Decoupled iterations

The coarse–grain nature of the asynchronous preconditioning iteration makes this operation naturally suited for distributed computing. Moreover, the preconditioning step can be performed

Algorithm 3.5 Classical Gram–Schmidt

OUTPUT: $[\mathbf{c}_k, \mathbf{u}_k] = \text{ortho–cgs}(\mathbf{c}, \mathbf{u}, \mathbf{c}_i, \mathbf{u}_i, k, m)$;

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

Algorithm 3.6 Asynchronous block Jacobi iteration task for each GridSolve server i.

OUTPUT: $\mathbf{u}_i = \mathcal{B}(\mathbf{r}_i, \varepsilon_{\text{in}}, T_{\max})$

1: Read $\mathbf{r}_i$ from IBP depot; Set $\mathbf{u}_i = \mathbf{0}$;
2: Perform ILU decomposition of $\mathbf{A}_{ii}$;
3: **while** $t_{\text{elapsed}} < T_{\max}$ **do**
4: Read relevant part of $\mathbf{u}$ from IBP depot;
5: Compute $\mathbf{v}_i = \mathbf{r}_i - \sum_j \mathbf{A}_{ij}\mathbf{u}_j$;
6: Solve $\mathbf{A}_{ii}\mathbf{p}_i = \mathbf{v}_i$ approximately with accuracy ε_{in};
7: Update $\mathbf{u}_i = \mathbf{u}_i + \mathbf{p}_i$;
8: Write $\mathbf{u}_i$ to IBP depot;
9: **end while**

on unreliable hardware. Stalling of one of the preconditioning servers will result in a less effective preconditioning operation, but the main solution method will not break down.

However, the other operations (i.e., the matrix–vector multiplication, orthogonalisation, and vector operations) are relatively fine–grain and need to be performed on stable hardware. It may therefore be natural to perform the outer iteration on the (reliable) client machine. This approach has an obvious limitation. Depending on the problem size and the number of servers, the outer iteration may become a computational bottleneck. We have therefore also implemented a parallel outer iteration using techniques described in Chapter 2.

Similar to the previous part, the matrix is partitioned using a homogeneous one–dimensional block–row distribution, both in the preconditioning iteration and in the outer iteration. The vectors are distributed accordingly. What follows are various implementation issues pertaining to performing the outer iteration in sequential or parallel. Note that in both cases the inner preconditioning is performed in parallel on heterogeneous computing hardware.

Sequential outer loop

All of the operations — with exception of the preconditioning iteration — are performed on the client machine. There is a single GridSolve task for the preconditioning step, which implies that there is a single global synchronisation point in each outer iteration step. The client machine begins by updating the complete residual on the IBP data depot. Algorithm 3.6 shows the specific steps performed by each server i in the preconditioning phase (cf. Algorithm 1.1 on page 8).

At the beginning of task i, the appropriate portion of the residual is read and the task starts

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

Parallel outer iteration

In this case, the only data that is communicated between the client and the computational nodes are the results of the (partial) inner products. The classical Gram–Schmidt algorithm (CGS) shown in Algorithm 3.5 was chosen for the orthogonalisation step, since it has favourable parallel properties.

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

A disadvantage of this approach is that every GridSolve task should be performed on reliable hardware. That is, should any of the tasks fail, it is likely that important intermediate information is lost, halting the entire outer iteration process.

3.6 Numerical experiments

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

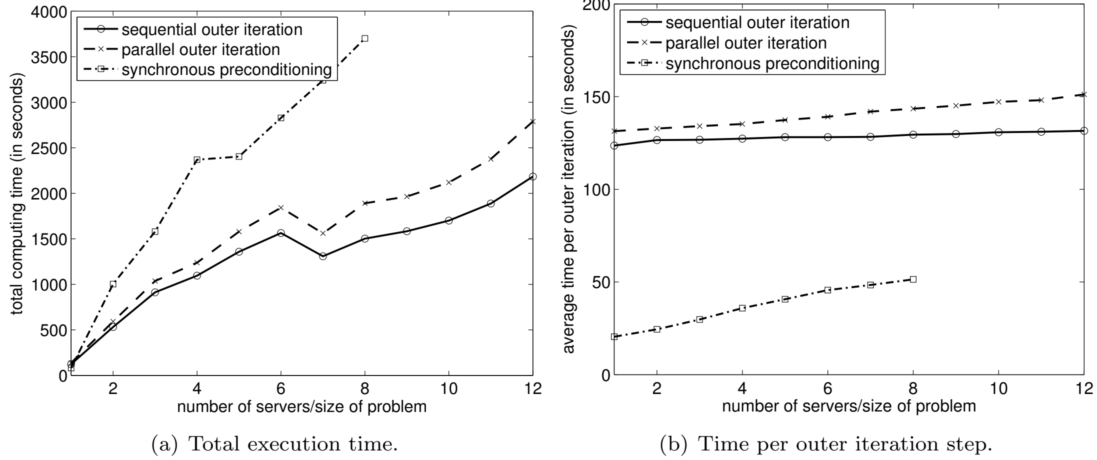
$$\begin{cases} -\nabla^2\mathbf{u} + (2\mathbf{p}\cdot\nabla)\mathbf{u} = \mathbf{f}, & \mathbf{u}\in\Omega, \\ \mathbf{u} = \mathbf{g}, & \mathbf{u}\in\partial\Omega, \end{cases} \tag{3.4}$$

where $\mathbf{p} = [1, 2, 3]$, Ω is the domain, and $\mathbf{f}, \mathbf{g}$ are given vectors. Discretisation by the finite difference scheme with a seven point stencil on a uniform $n_x \times n_y \times n_z$ mesh results in a sparse linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ is of order $n = n_x n_y n_z$. Centered differences are used for the first derivatives. The grid points are numbered using the standard (lexicographic) ordering, resulting in a heptadiagonal coefficient matrix. The right–hand side vector $\mathbf{b}$ is generated from the constant solution $\mathbf{x} = \mathbf{1}$.

3.6.1 Target hardware and experimental setup

The experiments are performed using a local cluster, which is a multi–user system. It is moderately heterogeneous in design, consisting of twelve nodes: six Intel 2.20 GHz machines, two Intel 2.66 GHz machines, and four AMD Athlon 2.20 GHz machines. The nodes are equipped with memory in the range 2–4 GB and the cluster is interconnected through 100 MB/s Ethernet links. The experiments are performed on a typical work day, while other users perform their computations. For more details on this cluster, see Section 1.8.1 on page 19.

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



(a) Total execution time.

(b) Time per outer iteration step.

Figure 3.5: Experiments on a large heterogeneous cluster with 250,000 equations per server.

method [120] with $s = 4$ and preconditioned with ILU. In the context of Grid computing, it is natural to fix the problem size per server and investigate the scalability of the algorithm by adding more servers in order to solve bigger problems. For each experiment, we take $n_x = n_y = n_z$ such that the number of equations of unknowns per server is approximately 250,000.

The outer iteration is performed either sequentially on the client machine (decoupled approach) or in parallel using the same nodes as the preconditioning nodes (coupled approach). The complete linear system is solved with relative tolerance $\varepsilon = 10^{-8}$. To limit memory requirements, the truncation parameter is kept small ($m = 5$).

3.6.2 Experimental results and discussion

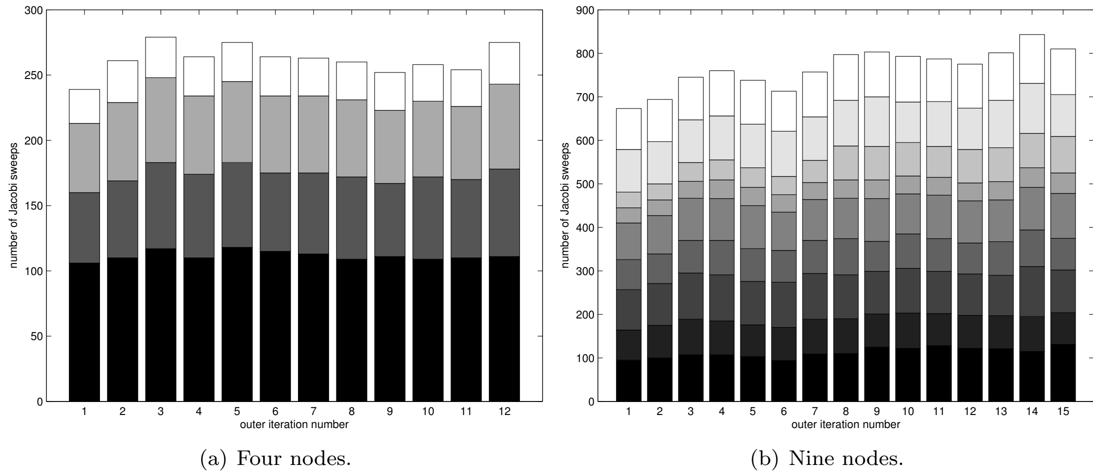
Five executions of the algorithm are performed, each time using a different and random set of servers. Figure 3.5 shows experimental results obtained using up to twelve servers (i.e., for problem sizes between 250,000 and three million), using both the sequential and parallel outer iteration. For comparison, results for standard (synchronous) block Jacobi preconditioning with inexact subdomain solves (using the parallel outer iteration) are also included.

Figure 3.5(a) shows the average execution times of the total iteration processes. The results show that the execution time of the asynchronous method increases when servers are added. This can be attributed almost completely to the increase in the number of outer iterations, which is approximately 4 for 2 servers and 11 for 8 servers. Using the same arguments as in Section 3.3.4, this increase in outer iterations is a result of the following two effects:

1. The coefficient matrix becomes increasingly ill–conditioned due to the increase of the problem size; and

2. the number of subdomains in asynchronous block Jacobi increases, which makes the preconditioner less effective.

Like before, factors that could also have a large impact on the effectiveness of the preconditioner are the heterogeneity of the hardware, the differences in workload, and fluctuations



(a) Four nodes.

(b) Nine nodes.

Figure 3.6: Jacobi sweeps performed by each server during outer iteration steps.

in network load. In the current computational environment and using the aforementioned parameters, the number of Jacobi sweeps during a single preconditioning step ranged between approximately 120 on a fully dedicated server and 30 on a fully occupied server. However, because the tasks are assigned to different servers in each outer iteration step, these effects are averaged out and the spread in total execution times remained within ten percent. Similar to the observations from the coupled approach in the first part, it seems that in the decoupled setting the asynchronous preconditioner is also robust with respect to the factors mentioned above.

Keeping the problem size per server fixed implies that — in the ideal case where overhead is negligible — the execution time per outer iteration remains constant. This is demonstrated in Figure 3.5(b), were we show the average times per outer iteration step. The results indicate that for the sequential outer loop the overhead is rather small. Also, for the parallel outer loop the increase in overhead due to the additional work and the (GridSolve) communication overhead is quite limited. In this case, the overhead grows more rapidly with increasing number of servers — compared to the sequential outer iteration.

For synchronous block Jacobi preconditioning, the total execution time grows significantly faster than for the asynchronous preconditioning if the number of servers is increased. This is a combination of two effects. Firstly, the number of iterations is higher for the synchronous preconditioner. That is, from 41 on 2 servers to 72 on 8 servers. A possible explanation is that, in contrast to asynchronous preconditioning, there is no exchange of information between the subdomains for synchronous preconditioning. Secondly, the time per iteration grows faster for the synchronous preconditioner. Since one synchronous preconditioning step requires much less computations than an asynchronous preconditioning step, the computation–to–synchronisation ratio is more favourable for asynchronous preconditioning. As a result, the asynchronous method outperforms the synchronous preconditioning technique. Moreover, this difference in performance becomes increasingly more significant for higher number of servers.

For illustrative purposes, Figure 3.6(a) and Figure 3.6(b) demonstrate the effect of varying workload and heterogeneity of hardware on a typical execution of the algorithm. It shows the number of Jacobi iterations performed by each server per outer iteration step, using four and nine servers respectively. Certain nodes exhibit an increased workload and its effect is clearly

visible.

Part III: Deflation and smoothing

In this part it is shown how deflation techniques can be combined with an asynchronous preconditioning iteration. In Section 3.7 the main motivations behind combining a deflation method with an asynchronous iterative method are discussed. Section 3.8 describes deflation methods and in Section 3.9 numerical experiments are performed that show the effectiveness of adding a deflation step to the flexible CG method from the first part of this chapter for solving the bubbly flow problem. In Section 3.10 the possibility of using asynchronous iterations as smoothers is briefly investigated experimentally. Section 3.11 formulates the general conclusions of this chapter.

3.7 Motivation

Asynchronous iterative methods methods are closely related to domain decomposition techniques. Large problem sizes implies a large number of processors, which corresponds to a large number of subdomains. To improve the robustness and scalability of the complete algorithm, some form of coarse grid correction such as deflation may be appropriate. The main idea is that high–frequency components of the residual are handled by the asynchronous preconditioning iteration, while low–frequency components can be handled by a coarse-grid correction step.

There are many types of deflation methods. In most (traditional) deflation variants, the "problematic" eigenvalues are deflated to zero. Perturbations such as roundoff errors, perturbed starting vectors, inaccurate preconditioning solves, and inaccurate Galerkin solves can transform these zero eigenvalues to near zero eigenvalues, which can hamper convergence severely. For our application, the so–called *adapted deflation variant 2* (A–DEF2) was chosen [127, Section 2.3.4], which was shown to be robust. In an adapted deflation method, the eigenvalues are shifted to one instead of zero. As a result, any perturbations will transform the unit eigenvalues into near unit eigenvalues, which are harmless to the convergence process. Since our preconditioning step is a highly variable process where the subdomains are solved inexactly, it is absolutely essential that the employed deflation method is robust with respect to such variations.

Another important choice are the deflation vectors, whose effectiveness depends on the application. For solving bubbly flow problems on Grid computers using our method, *subdomain deflation* using piecewise–constant, disjoint and orthogonal deflation vectors is a very effective choice. Some of the reasons are as follows (cf. [130, Section 4.2]):

- As mentioned before, asynchronous iterations are closely related to domain decomposition methods. In subdomain deflation, each of these subdomains can be made to correspond to one or multiple deflation vectors. The deflation vector consists of ones for grid points inside a subdomain and zero for grid points outside the subdomain.
- In the outer iteration, the deflation steps are parallel *synchronous* operations. Since subdomain deflation vectors are disjoint, the deflation steps can be performed efficiently in parallel, minimising the overhead in the outer iteration. In addition, it is shown below that (sequential) subdomain deflation can also be used within the asynchronous preconditioning iteration.
- It can be shown that subdomain deflation vectors approximate the eigenspace corresponding to the unfavourable eigenvalues of the coefficient matrix from the bubbly flow test problem.

The coefficient matrix $\mathbf{A}$ originating from the bubby flow problem is singular and symmetric positive semidefinite. Although the standard preconditioned CG method can handle singular matrices [83], applying a deflated CG method to a singular matrix will result in a singular Galerkin matrix. This means that solving the Galerkin systems using direct methods may pose difficulties. The coefficient matrix can be made invertible by perturbing the lower right element of the coefficient matrix, which in essence means that a Dirichlet condition is imposed on one point in the domain. This results in a near–singular coefficient matrix, which can increase the condition number. However, in [129] it was shown that applying deflated PCG either directly to the original singular system or to the modified near–singular system results in similar favourable convergence behaviour.

For ease of implementation, we use a near–singular matrix, since it allows for solving the coarse Galerkin systems using direct methods. For larger problem sizes (i.e., more subdomains), the number of deflation vectors will increase. As a result, the size of the Galerkin systems also increases and iterative methods may be employed to solve these systems (possibly inexactly). This allows us to work with singular coefficient matrices directly.

In our application, subdomain deflation is applied on two different levels: in the outer iteration and in the local subdomain solves of the asynchronous preconditioning iteration. These two strategies can contribute to the reduction of the total number of outer iterations. The strategies are connected in the following manner.

- *Outer deflation.* Without deflation, the total number of outer iterations is critically dependent on the number of bubbles present in the problem and the number of subdomains. Additionally, the effectiveness of the asynchronous preconditioner in heterogeneous computing environments may fluctuate considerably and thus the number of outer iterations. Not only will adding outer deflation remove unfavourable eigenvalues associated with the bubbles, it will also increase robustness with respect to inexact and fluctuating preconditioning steps. This reduces the number of outer iterations considerably. It is expected that this will only work if the asynchronous preconditioner handles the larger eigenvalues and the outer deflation handles the smaller eigenvalues.

- *Inner deflation.* By using deflation in the local subdomain solves, the number of iterations for solving the subdomains on each processor will be roughly the same and desynchronisation of the Jacobi processes may be reduced. As a result, it is expected that the operator $\mathcal{B}$ from (3.1) will increasingly approximate some SPD matrix $\mathbf{B}^{-1}$. This could also allow for a smaller truncation parameter, resulting in reduced memory requirements. In this case, load balancing also becomes less complicated.

3.8 Deflation methods

It is not our intention to give a complete discussion of deflation–type methods in this section. For more detailed information on deflation methods, see Chapter 5 of this thesis or [127].

In most deflation methods, two different kinds of preconditioners are combined: a traditional (first-level) preconditioner and a coarse-grid (second-level) correction step. The following definitions are needed to derive the A–DEF2 deflation method [127]. The deflation matrices $\Pi, \widehat{\Pi} \in \mathbb{R}^{n\times n}$, the correction matrix $\mathbf{Q} \in \mathbb{R}^{n\times n}$, and the Galerkin matrix $\sigma \in \mathbb{R}^{t\times t}$ are defined as follows:

$$\Pi \equiv \mathbf{I} - \mathbf{AQ} \quad \text{and} \quad \widehat{\Pi} = \Pi^* \equiv \mathbf{I} - \mathbf{QA} \quad \text{where} \quad \mathbf{Q} \equiv \mathbf{Z}\sigma^{-1}\mathbf{Z}^* \quad \text{and} \quad \sigma \equiv \mathbf{Z}^*\mathbf{AZ}, \tag{3.5}$$

where $\mathbf{Z} \in \mathbb{R}^{n\times t}$ is a full-rank matrix consisting of t deflation vectors.

Algorithm 3.7 Computation of $\mathbf{u} = \mathbf{P}^{\text{adef2}}\mathbf{r}_k$

1: $\mathbf{w}_1 = \mathbf{Q}\mathbf{r}_k$
2: $\mathbf{w}_2 = \mathcal{B}(\mathbf{r}_k, \varepsilon_{\text{in}}, T_{\max})$ (cf. (3.1))
3: $\mathbf{w}_3 = \widehat{\Pi}\mathbf{w}_2$
4: $\mathbf{u} = \mathbf{w}_1 + \mathbf{w}_3$

Algorithm 3.8 Computation of $\widehat{\Pi}\mathbf{y}$

1: $\mathbf{y}_1 = (\mathbf{AZ})^* y$
2: Solve $\sigma \mathbf{y}_2 = \mathbf{y}_1$
3: $\mathbf{y}_3 = \mathbf{Z}\mathbf{y}_2$
4: $\widehat{\Pi}\mathbf{y} = \mathbf{y} - \mathbf{y}_3$

The A–DEF2 method can be seen as the combination of two preconditioners. Let $\mathbf{C}_1$ and $\mathbf{C}_2$ be two preconditioners. Given an initial guess for the iterate $\mathbf{x}_0$, consider the two–step stationary iterative method for $k \in \mathbb{N}_0$:

$$\begin{aligned} \mathbf{x}'_k &= \mathbf{x}_k + \mathbf{C}_1(\mathbf{b} - \mathbf{A}\mathbf{x}_k); \\ \mathbf{x}_{k+1} &= \mathbf{x}'_k + \mathbf{C}_2(\mathbf{b} - \mathbf{A}\mathbf{x}'_k), \end{aligned} \tag{3.6}$$

which can be combined to obtain $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x}_k)$, where

$$\mathbf{P} = \mathbf{C}_1 + \mathbf{C}_2 - \mathbf{C}_2\mathbf{A}\mathbf{C}_1. \tag{3.7}$$

In the A–DEF2 deflation method, a standard preconditioner $\mathbf{B}^{-1}$ is applied first, followed by a coarse-grid correction step $\mathbf{Q}$. In other words, let $\mathbf{C}_1 \equiv \mathbf{B}^{-1}$ and $\mathbf{C}_2 \equiv \mathbf{Q}$. The complete deflation operator of A–DEF2 is then

$$\mathbf{P}^{\text{adef2}} = \widehat{\Pi}\mathbf{B}^{-1} + \mathbf{Q}. \tag{3.8}$$

Note that in our application, the preconditioning step $\mathbf{B}^{-1}\mathbf{r}$ is the result of applying the operator $\mathcal{B}(\mathbf{r})$ from (3.1).

For any full rank matrix $\mathbf{Z}$, we have $\mathbf{P}^{\text{adef2}}\mathbf{AZ} = \mathbf{Z}$, so that the matrix $\mathbf{P}^{\text{adef2}}\mathbf{A}$ has t unit eigenvalues. In "traditional" deflation methods, the operator $\mathbf{Q}$ is omitted from (3.8), which means that the eigenvalues are shifted to zero instead.

The deflation step can be easily added to the flexible CG method by replacing the preconditioning step in line 2 from Algorithm 3.1 with the operation $\mathbf{u} = \mathbf{P}^{\text{adef2}}\mathbf{r}_k$. In addition, a special starting vector $\mathbf{x}_0 = \mathbf{Q}\mathbf{b} + \widehat{\Pi}\mathbf{x}_0$ has to be used. The other operations of flexible CG remain the same.

The specific steps of the operation $\mathbf{u} = \mathbf{P}^{\text{adef2}}\mathbf{r}_k$ are shown in Algorithm 3.7. Note that steps 1 and 3 of Algorithm 3.7 are *synchronous* operations performed by the outer iteration, while step 2 is the asynchronous inner iteration. The operation $\widehat{\Pi}\mathbf{y}$ for some vector $\mathbf{y}$ is performed using the steps in Algorithm 3.8. The small matrix σ and the sparse matrix $(\mathbf{AZ})^*$ can be computed once and stored beforehand. When using subdomain deflation, the application of $\mathbf{Z}$ and $\mathbf{Z}^*$ to some vector can be computed efficiently without storing the matrix $\mathbf{Z}$ explicitly.

3.9 Numerical experiments

For the outer iteration, the A–DEF2 deflation method is used with flexible CG and preconditioned with an asynchronous iterative method. The small systems with σ in the deflation step

	10 nodes (5×10^6 equations)		20 nodes (10^7 equations)	
$T_{\max}$ (s)	without deflation	with deflation	without deflation	with deflation
0	170	134	301	230
5	55 (275)	45 (225)	107 (535)	71 (335)
10	35 (350)	26 (260)	41 (410)	43 (430)
15	25 (375)	20 (300)	42 (630)	34 (510)
20	18 (360)	19 (380)	40 (800)	28 (560)

Table 3.5: Number of outer iterations (3D problem).

are solved using a precomputed Cholesky decomposition. The deflation vectors correspond to the subdomains used in the asynchronous preconditioning iteration, which are horizontal slices across the z-axis. The number of deflation vectors is equal to the number of total grid points in the z-direction. That is, the height of the deflation subdomains is one grid point. The truncation parameter of flexible CG is set to $m_{\max} = 5$.

In the asynchronous preconditioning iteration, the local system of the subdomain is solved using standard CG preconditioned with incomplete Cholesky and using A–DEF2 deflation. Similar to the outer deflation, the number of (local) deflation vectors is equal to the number of grid points of the subdomain in the z-direction.

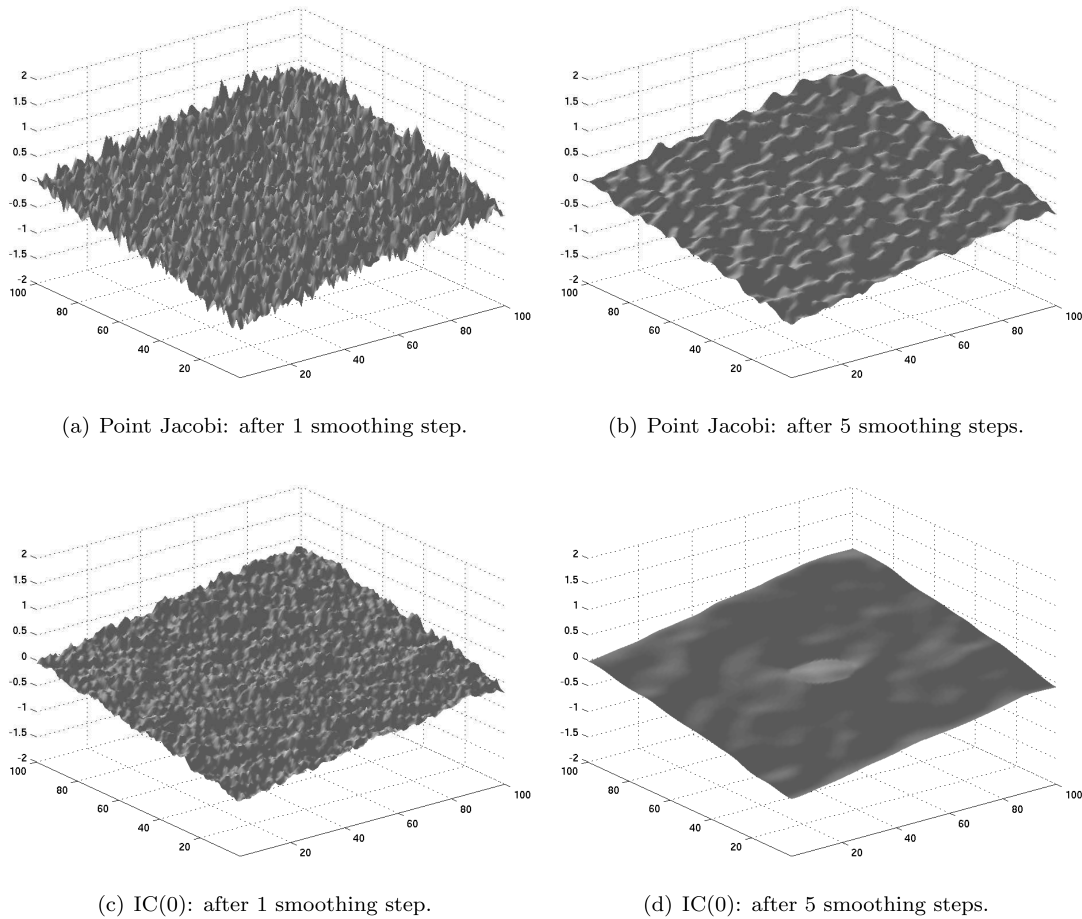
The deflation steps are implemented in a *sequential* outer iteration, while the asynchronous preconditioning iteration is performed in parallel. As before, the number of equations per node is fixed to 500,000 equations.

Table 3.5 shows experimental results for the deflation method A–DEF2 applied to the 3D bubbly flow problem on a lightly loaded network. Not surprisingly, the additional overhead of the deflation step in the (sequential) outer iteration results in a large computing time per iteration step. Therefore, only the number of iterations are relevant and experiments using only 10 and 20 nodes are performed (or equivalently, for problem sizes of five and ten million equations). The results shown are the minimum number of iteration steps of four different experimental runs. For completeness, the theoretical total computing time (i.e., zero overhead in the outer iteration) is shown between brackets.

This shows that the larger the problem size, the more effective the deflation step. A possible explanation is that for larger problem sizes, the number of unfavourable eigenvalues increases and a coarse grid correction step will become more effective. Also, the results show that the more asynchronous preconditioning, the less effective the deflation step. This is not surprising, since information is exchanged between the subdomains in the asynchronous preconditioning. This interfers with the main objective of a coarse grid correction, which is to tie the subdomains together as well.

In general, the deflation step has a rather small effect on the number of iterations. A possible explanation is that the bulk of the (predominately large) eigenvalues of the symmetric coefficient matrix are efficiently handled by the asynchronous inner iteration, while the relatively small number of difficult eigenvalues may be removed by a small number of (nondeflated) outer iterations. However, increasing the number of bubbles will result in an increase of outer iterations when using the same amount of asynchronous preconditioning. For this case, a deflation step (with appropriate deflation subdomains) will be more effective.

Currently, the deflation vectors correspond to the subdomains originating from the asynchronous preconditioning iteration, which are horizontal slices across the z-axis. Naturally, more sophisticated subdomains such as cuboids can be used for the deflation vectors, which will



(a) Point Jacobi: after 1 smoothing step.

(b) Point Jacobi: after 5 smoothing steps.

(c) IC(0): after 1 smoothing step.

(d) IC(0): after 5 smoothing steps.

Figure 3.7: Point smoothers.

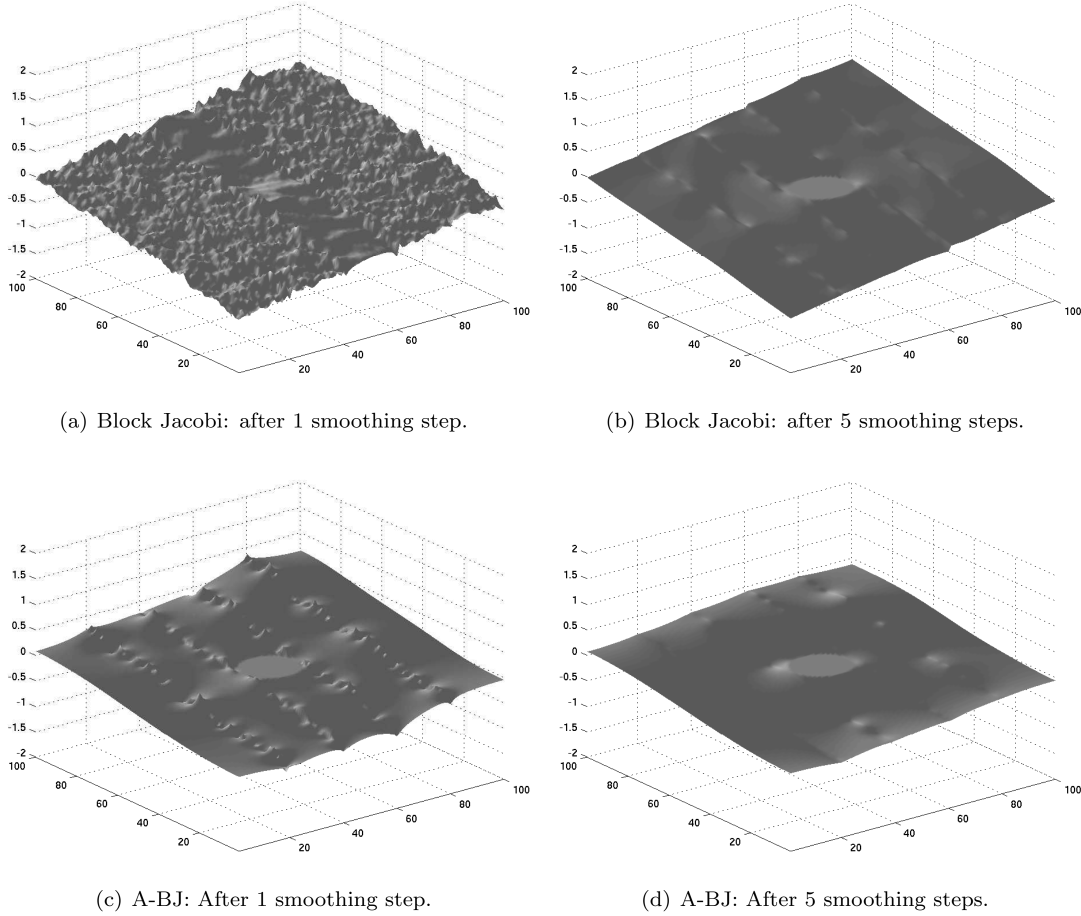
increase the effectiveness of the deflation step.

3.10 Using asynchronous iterative methods as smoothers

To further investigate the limited effect of the deflation step, we will investigate experimentally the smoothing properties of asynchronous iterative methods.

In multigrid, the preconditioner is often a weak fine–scale smoother (e.g., Jacobi or Gauss-Seidel) combined with a coarse–scale correction over a relatively large space. Since the asynchronous preconditioning iteration is a relatively strong fine-scale preconditioner as we will show below, using asynchronous relaxation in multigrid may not be entirely appropriate.

Since the preconditioning matrix $\mathbf{B}_k^{-1}$ is not explicitly available in each outer iteration step k, it is not trivial to investigate the effect of the asynchronous iteration on the spectrum $\sigma(\mathbf{B}_k^{-1}\mathbf{A})$. We will therefore compare and illustrate experimentally the error reducing properties of the following four smoothers: asynchronous block Jacobi iteration, standard (synchronous) block Ja-



(a) Block Jacobi: after 1 smoothing step.

(b) Block Jacobi: after 5 smoothing steps.

(c) A-BJ: After 1 smoothing step.

(d) A-BJ: After 5 smoothing steps.

Figure 3.8: Smoothing for block Jacobi and for asynchronous iterations (A-BJ).

cobi, standard point Jacobi, and Incomplete Cholesky decomposition without fill-in (i.e., IC(0)). Note that the first two methods are block smoothers and will be performed in parallel, while the latter two are point smoothers and are performed sequentially. We start with a random initial solution and solve for a singular system $\mathbf{Ax} = \mathbf{b}$ from a 2D bubbly flow problem of dimension $n = 100^2$ with a single bubble centered in the middle. Richardson is used as the outer iteration and for the block smoothers the domain is partitioned into five equally–sized rectangular strips.

For the asynchronous smoother, performing a single sweep (or smoothing or relaxation step) means in our case that the asynchronous iteration is stopped when one of the asynchronous processes has performed a maximum of five local iteration steps. Note that due to the randomness of the asynchronous iteration process, the number of iterations the other asynchronous processes has performed when the iteration is stopped will vary for each sweep.

Shown in Figure 3.7 and Figure 3.8 is the error $\mathbf{x} - \mathbf{x}_k$ after one sweep and after five sweeps using the four smoothing methods. Note that the bubble can be observed at the center. This shows that for this particular test case, point Jacobi (Figure 3.7(a) and Figure 3.7(b)) has less smoothing capabilities than the IC(0) method (Figure 3.7(a) and Figure 3.7(b)). For point

Jacobi, the error is still oscillatory even after five sweeps.

The block Jacobi method (Figure 3.8(a) and Figure 3.8(b)) has slightly better smoothing properties compared to the IC(0) method. After one sweep, the smoothness of the error is comparable to IC(0). However, after five sweeps the error for block Jacobi is completely smooth, while with IC(0) some oscillations can still be observed. Also, the five domains in block Jacobi can be seen clearly. For unknown reasons, the subdomain that contains the bubble is already completely smooth after a single sweep of the block Jacobi method (Figure 3.8(a)).

For the asynchronous method (Figure 3.8(c) and Figure 3.8(d)), the error after one smoothing step (Figure 3.8(c)) is comparable to block Jacobi after five (synchronous) sweeps (Figure 3.8(b)). Although the error *inside* the subdomains is extremely smooth, at the boundary oscillatory behaviour can clearly be observed. Also, slow oscillations can be seen within the subdomains, which suggests that using deflation techniques in the local subdomain solves may be beneficial.

After five sweeps the total error is reduced substantially in comparison to the other smoothers and the artifacts at the boundaries have disappeared. Since the error is quite smooth even after a small number of asynchronous sweeps, it is expected that a coarse-grid correction from a small dimensional subspace (e.g., equal to the number of subdomains) can be very effective. This further motivates the idea that it is more natural to combine asynchronous iterations with deflation techniques than with multigrid.

These results contradict the remarks in [59, p. 7], where it is argued that asynchronous relaxation methods will never really work as (standard) smoothers, because the small number of smoothing iterations will make it difficult to exchange enough information between neighbouring subdomains. However, [59] also states that the smoothing of the error will only occur *within* a subdomain, which is exactly what our experiments showed.

3.11 Conclusions

In the coupled iteration approach, the CRAC library allows for easy implementation of the partially asynchronous iterative algorithm on a multi–cluster. Also, extensive numerical experiments using approximately 100 nodes divided between five geographically separated and dedicated clusters showed that:

1. Using the partially asynchronous algorithm is more efficient than (i) using a fully synchronous method or (ii) using a fully asynchronous method;

2. The asynchronous preconditioner adapts to a computational environment in which the network is heavily loaded;

In the decoupled iteration approach, GridSolve allows for straightforward physical separation of the two iteration processes. The inner preconditioning iteration is performed on unreliable (heterogeneous and distant) computational resources, while the outer (and main) iteration is performed on stable (homogeneous and local) hardware. This results in an algorithm that is partially fault–tolerant. In this manner, efficient use of existing and non–dedicated resources can be achieved. Similar to the coupled approach, using the partially asynchronous algorithm is more efficient than using a fully synchronous method.

Also, GridSolve is originally intended for purely coarse–grain numerical algorithms, but we successfully applied it to a fine–grain iterative solution approach. Despite the inherent limitations of the employed middleware GridSolve and the extremely volatile nature of the computational resources, encouraging experimental results are obtained.

The numerical experiments showed that both approaches (coupled and decoupled) were robust with respect to heterogeneity in hardware and changes in network load. It can therefore be

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

The ideas presented in this chapter were applied in the context of Grid computing. However, the asynchronous preconditioning approach may also be of interest for parallel computing with multi–core processors, where connections between cores on the same processor are much faster than connections between the processors.

Complementing the asynchronous preconditioning step with a coarse grid correction is relatively straightforward. However, the subsequent decrease in number of outer iterations is rather mild. Nevertheless, the results are promising and it is expected that increasing the number of bubbles and using more sophisticated deflation vectors will increase the effectiveness of a deflation step.

Small test cases suggest that asynchronous smoothing can be beneficial for multi–level preconditioning and in particular for deflation–type methods. However, larger experiments did not corroborate these findings and more research is needed to explain this.

Chapter 4

IDR(s) for Grid Computing

This chapter has been published as:

T. P. Collignon and M. B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 2011. (published online: 14 january 2011).

M. B. van Gijzen and T. P. Collignon. Exploiting the flexibility of IDR(s) for Grid computing. In *The Proceedings of the 2nd Kyoto–Forum on Krylov Subspace Methods, Kyoto University, Kyoto, Japan*, March 2010.

Overview

IDR(s) is a family of fast algorithms for iteratively solving large nonsymmetric linear systems. With cluster computing and in particular with Grid computing, the inner product is a bottleneck operation. In this chapter, four techniques are investigated for alleviating this bottleneck. Firstly, a recently proposed IDR(s) algorithm that is efficient and stable is reformulated in such a way that it has a single global synchronisation point per matrix–vector multiplication. Secondly, the so–called IDR test matrix is chosen so that the work, communication, and storage involving this matrix is minimised in multi–cluster environments. Thirdly, a methodology is presented for *a priori* estimation of the optimal value of s. Finally, the reformulated IDR(s) variant is combined with the asynchronous preconditioning technique as described in Chapters 1 and 3. Numerical experiments applied to a 3D convection–diffusion problem are performed on the DAS–3 Grid computer, demonstrating the effectiveness of our approach.

4.1 Introduction

The recently proposed induced dimension reduction (IDR) method [120] and its variants are short recurrence Krylov subspace methods for iteratively solving large nonsymmetric linear systems

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{A} \in \mathbb{C}^{n\times n}, \quad \mathbf{x}, \mathbf{b} \in \mathbb{C}^{n}. \tag{4.1}$$

In [78, 115, 120] it is shown that for $s = 1$, IDR(s) is mathematically equivalent to the ubiquitous Bi–CGSTAB [136] method. For important classes of problems and for relatively small values of $s > 1$ (e.g., $s = 4$ or 6), the IDR(s) algorithms outperform Bi–CGSTAB, see for example [120, Section 6] and [140].

The IDR(s) method has attracted considerable attention and we will give a short overview of some IDR(s) related papers. The IDR approach to solving nonsymmetric linear systems is quite nonstandard and in [78, 115] the connections between the IDR(s) method and more traditional Krylov subspace methods are explained. Several algorithmic variations are proposed and analysed in for example [97, 98, 139, 140], while [45, 138] aims to optimise IDR(s) methods in a parallel and Grid computing context. In [117], the strengths of BiCGstab(ℓ) [113] and IDR(s) are combined, resulting in the superior IDRstab method. A related approach is used in [131]. In [112] the IDR(s) method is interpreted as a Petrov–Galerkin method and in [79] the IDR approach is used for eigenvalues computations. More recently, an in–depth convergence analysis of IDR(s) using statistical arguments was performed in [119]. Several interesting results related to the algebra of induced dimension reduction are discussed in [116].

The goal of this chapter is to construct an efficient IDR(s) algorithm for cluster and Grid computing. Global synchronisation is a bottleneck operation in such computational environments and to alleviate this bottleneck, four techniques are investigated:

(i.) The efficient and numerically stable IDR(s)-*biortho* method from [140] is reformulated in such a way that it has one global synchronisation point per MV. The resulting method is named IDR(s)-*minsync*;

(ii.) Using a parallel performance model, *a priori* estimation of the optimal parameter s and number of processors is performed to minimise the total computing time using only the problem size and machine–dependent parameters such as latency and bandwidth;

(iii.) Piecewise sparse column vectors for the IDR test matrix are used to minimise computation, communication, and storage involving this matrix in multi–cluster environments.

(iv.) The IDR(s)-minsync variant mentioned above is preconditioned using an asynchronous iterative method, which is an effective preconditioner in the context of Grid computing, as shown in Chapter 3.

Extensive experiments on a 3D convection–diffusion problem show that the performance model is in good agreement with the experimental results. The model is successfully applied to both a single cluster and to the DAS–3 multi–cluster. Comparisons between IDR(s)-biortho and IDR(s)-minsync are made, demonstrating superior scalability and efficiency of the reformulated method IDR(s)-minsync.

Techniques for reducing the number of synchronisation points in Krylov subspace methods on parallel computers has been studied by several authors [37, 49, 75, 76, 152, 151, 153, 87]. With the advent of Grid computing, the need for reducing global synchronisations is larger than ever. The combination of the four strategies used in this chapter results in an efficient iterative method for solving large nonsymmetric linear systems on Grid computers.

Note that the prototype method IDR(s)-*proto* from the original IDR(s) paper [120] also has a single global synchronisation point per MV, except when computing a new ω each $s + 1$st step, which requires two global synchronisations. However, the IDR(s)-biortho method exhibits superior numerical stability and has less floating point operations per IDR(s) cycle. Therefore, the IDR(s)-biortho method was used as a basis for the new IDR(s)-minsync variant.

The index k always refers to the kth IDR(s) *cycle*. Note that one IDR(s) cycle consists of $s + 1$ MVs.

This chapter is organised as follows. In Section 4.2 the IDR(s)-minsync variant with a single global synchronisation point per MV is presented. Section 4.3 explains how IDR(s) methods are parallelised in this chapter. Section 4.4 describes the parallel performance model that is used to estimate the optimal value of s. It also describes a technique of using piecewise sparse column vectors for the test matrix to minimise work involving this matrix on a multi–cluster. In Section 4.5 the IDR(s)-minsync variant is discussed within the context of asynchronous preconditioning and it is shown that the key recursions of IDR(s) methods remain consistent within this context. Section 4.6 contains extensive experimental results on the DAS–3, which show the effectiveness of the four strategies. Concluding remarks are given in Section 4.7.

4.2 IDR(s) variant with one synchronisation point per MV

The IDR(s)–based methods are new iterative algorithms for solving large nonsymmetric systems and much research is needed on efficient parallelisation on distributed memory computers. When applying the so–called IDR(s) theorem to derive practical algorithms, certain choices can be made. This freedom allows for efficient tuning of the numerical algorithm to specific computational environments. In this section the numerically stable IDR(s)-biortho method is reproduced from [140] and used as a basis for the IDR(s)-minsync method, which has a single global synchronisation point per MV.

Given an initial approximation $\mathbf{x}_0$ to the solution, all IDR(s) methods construct residuals $\mathbf{r}_k$ in a sequence $(\mathcal{G}_k)$ of shrinking subspaces that are related according to the following theorem.

Theorem 4.1 (Induced Dimension Reduction (IDR))**.** *Let $\mathbf{A} \in \mathbb{C}^{n\times n}$, let $\mathbf{B} \in \mathbb{C}^{n\times n}$ be a preconditioning matrix, let $\widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$ be a fixed matrix of full rank, and let $\mathcal{G}_0$ be any non–trivial invariant linear subspace of $\mathbf{A}$. Define the sequence of subspaces $(\mathcal{G}_k)$ recursively as*

$$\mathcal{G}_{k+1} \equiv (\mathbf{I} - \omega_{k+1}\mathbf{A}\mathbf{B}^{-1})(\mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}) \quad \textit{for} \quad k = 0, 1, \ldots, \tag{4.2}$$

where (ω_k) *is a sequence in* $\mathbb{C}$*. If* $\widetilde{\mathbf{R}}^\perp$ *does not contain an eigenvector of* $\mathbf{AB}^{-1}$*, then for all* $k \geq 0$

- $\mathcal{G}_{k+1} \subset \mathcal{G}_k$*;*
- $\dim \mathcal{G}_{k+1} < \dim \mathcal{G}_k$ *unless* $\mathcal{G}_k = \{\mathbf{0}\}$.

Ultimately, the residual is forced in the zero–dimensional subspace $\mathcal{G}_k = \{\mathbf{0}\}$ for some $k \leq n$. For a proof the reader is referred to e.g., [120, 115, 117].

Iterative algorithms based on the IDR(s) theorem can be seen as being divided into two steps, which constitute the kth cycle of an IDR(s) method (i.e., $s+1$ (preconditioned) matrix–vector multiplications):

1. The dimension reduction step: given s vectors in $\mathcal{G}_k$ and a residual $\mathbf{r}_k \in \mathcal{G}_k$, a residual $\mathbf{r}_{k+1}$ in the lower dimensional subspace $\mathcal{G}_{k+1} = (\mathbf{I} - \omega_{k+1}\mathbf{AB}^{-1})(\mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp) \subset \mathcal{G}_k$ is computed after selecting an appropriate ω_{k+1};
2. Generating s additional vectors in $\mathcal{G}_{k+1}$.

In the next chapter these two steps are discussed in a more general setting than in this chapter and it is argued that the dimension reduction step should be seen as consisting of two separate phases. That is, the projection used in forming a residual in $\mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$ is a key projection in IDR(s) and this should be seen as a separate step. However, for the purpose of this chapter, it is sufficient to consider an IDR(s) cycle as consisting of the two steps shown above.

It can be shown that in the generic case, IDR(s) methods terminate within $\frac{n}{s}$ dimension reduction steps in exact arithmetic, or equivalently, within $n(1+\frac{1}{s})$ (preconditioned) matrix–vector multiplications [120, Section 3]. In practical applications, the iteration process will exhibit much faster convergence rates according to $\frac{\widehat{n}}{s}$ IDR(s) cycles, where $\widehat{n} \ll n$.

Shown in Algorithm 4.1 is the (right) preconditioned IDR(s)-biortho variant [140], which not only has slightly less operations but is also numerically more stable than the IDR(s)-proto variant. The elements of the small $s \times s$ matrix $\mathbf{M}$ are denoted by $\mu_{i,j}$ for $1 \leq i,j \leq s$ and $\mathbf{M}$ is initially set to the identity matrix. The dimension reduction step (i.e., lines 32–37 in Algorithm 4.1) consists of one preconditioned matrix–vector product, two vector updates, and two inner products. Combined with the operations for constructing s vectors in $\mathcal{G}_k$ (i.e., lines 6–31 in Algorithm 4.1), this amounts to $s+1$ preconditioned matrix–vector products, $s(s+1)+2$ inner products, and $2(s(s+1)+1)$ vector updates per cycle of IDR(s). The computation of the (combined and separate) inner products is highlighted by boxes, which shows that there are $\frac{1}{2}(s(s+1))+2$ global synchronisation points per IDR(s) cycle.

In the IDR(s)-biortho method, certain bi–orthogonality conditions with the columns of $\widetilde{\mathbf{R}}$ are enforced that result in improved numerical stability and in reduced number of vector operations compared to IDR(s)-proto. To be more specific, let $\mathbf{r}_{n+1}$ be the first residual in $\mathcal{G}_{k+1}$. In IDR(s)-biortho, vectors for $\mathcal{G}_{k+1}$ are made to satisfy

$$\mathbf{g}_{n+j} \perp \widetilde{\mathbf{r}}_i, \quad i = 1, \ldots, j-1, \quad j = 2, \ldots, s, \tag{4.3}$$

and *intermediate* residuals (also in $\mathcal{G}_{k+1}$) are made to satisfy

$$\mathbf{r}_{n+j+1} \perp \widetilde{\mathbf{r}}_i, \quad i = 1, \ldots, j, \quad j = 1, \ldots, s. \tag{4.4}$$

In the implementation presented in Algorithm 4.1, these conditions are enforced using a modified Gram–Schmidt (MGS) process for oblique projection (lines 15–24 in Algorithm 4.1). The disadvantage of this approach is that the inner products cannot be combined, which poses

Algorithm 4.1 IDR(s)-biortho with bi–orthogonalisation of intermediate residuals.

INPUT: $\mathbf{A} \in \mathbb{C}^{n\times n}; \mathbf{x}, \mathbf{b} \in \mathbb{C}^n; \widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$; preconditioner $\mathbf{B} \in \mathbb{C}^{n\times n}$; parameter s; accuracy ε.
OUTPUT: Approximate solution $\mathbf{x}$ such that $||\mathbf{b} - \mathbf{A}\mathbf{x}|| \le \varepsilon$.

1: // *Initialisation*
2: Set $\mathbf{G} = \mathbf{U} = \mathbf{0} \in \mathbb{C}^{n\times s}; \mathbf{M} = [\mu_{i,j}] = \mathbf{I} \in \mathbb{C}^{s\times s}; \omega = 1$
3: Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
4: // *Loop over nested* $\mathcal{G}_k$ *spaces,* $k = 0, 1, \ldots$
5: **while** $||\mathbf{r}|| \ge \varepsilon$ **do**
6: // *Compute* s *linearly independent vectors* $\mathbf{g}_j$ *in* $\mathcal{G}_k$
7: $\boxed{\phi = \widetilde{\mathbf{R}}^*\mathbf{r}}$, $\phi = (\phi_1, \ldots, \phi_s)^\top$ // s *inner products (combined)*
8: **for** $j = 1$ to s **do**
9: Solve $\mathbf{M}\gamma = \phi$ for γ, $\gamma = (\gamma_j, \ldots, \gamma_s)^\top$
10: $\mathbf{v} = \mathbf{r} - \sum_{i=j}^{s} \gamma_j \mathbf{g}_i$
11: $\widetilde{\mathbf{v}} = \mathbf{B}^{-1}\mathbf{v}$ // *Preconditioning step*
12: $\mathbf{u}_j = \sum_{i=j}^{s} \gamma_i \mathbf{u}_i + \omega\widetilde{\mathbf{v}}$
13: $\mathbf{g}_j = \mathbf{A}\mathbf{u}_j$
14: // *Make* $\mathbf{g}_j$ *orthogonal to* $\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{j-1}$
15: **for** $i = 1$ to $j - 1$ **do**
16: $\boxed{\alpha = \widetilde{\mathbf{r}}_i^*\mathbf{g}_j/\mu_{i,i}}$ // $j - 1$ *inner products (separate)*
17: $\mathbf{g}_j = \mathbf{g}_j - \alpha\mathbf{g}_i$
18: $\mathbf{u}_j = \mathbf{u}_j - \alpha\mathbf{u}_i$
19: **end for**
20: // *Update column* j *of* $\mathbf{M}$
21: $\boxed{\mu_{i,j} = \widetilde{\mathbf{r}}_i^*\mathbf{g}_j}$ for $i = j, \ldots, s$ // $s - j + 1$ *inner products (combined)*
22: // *Make the residual orthogonal to* $\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_j$
23: $\beta = \phi_j/\mu_{j,j}$
24: $\mathbf{r} = \mathbf{r} - \beta\mathbf{g}_j$
25: $\mathbf{x} = \mathbf{x} + \beta\mathbf{u}_j$
26: // *Update* $\phi = \widetilde{\mathbf{R}}^*\mathbf{r}$
27: **if** $j + 1 \le s$ **then**
28: $\phi_i = 0$ for $i = 1, \ldots, j$
29: $\phi_i = \phi_i - \beta\mu_{i,j}$ for $i = j + 1, \ldots, s$
30: **end if**
31: **end for**
32: // *Entering* $\mathcal{G}_{k+1}$. *Note:* $\mathbf{r} \perp \widetilde{\mathbf{R}}$
33: $\widetilde{\mathbf{v}} = \mathbf{B}^{-1}\mathbf{r}$ // *Preconditioning step*
34: $\mathbf{t} = \mathbf{A}\widetilde{\mathbf{v}}$
35: $\boxed{\omega = (\mathbf{t}^*\mathbf{r})/(\mathbf{t}^*\mathbf{t})}$ // *Two inner products (combined)*
36: $\mathbf{r} = \mathbf{r} - \omega\mathbf{t}$
37: $\mathbf{x} = \mathbf{x} + \omega\widetilde{\mathbf{v}}$
38: **end while**

a bottleneck in parallel computing environments. By using a classical Gram–Schmidt (CGS) process for these projections, this bottleneck can be alleviated. The general idea is that *all* the inner products can be recursively computed with a one–sided bi–orthogonalisation process using solely *scalar updates*.

In the *standard* Gram–Schmidt algorithm, a full set of mutually orthogonal vectors is formed. In our case, the Gram–Schmidt process is used for orthogonalising a vector with respect to a *fixed* set of vectors (i.e., the IDR test matrix $\widetilde{\mathbf{R}}$). Classical GS is known to be less robust than modified GS. However, it is not immediately apparent whether using the classical GS process for orthogonalising a vector with respect to a fixed set of vectors is also less stable than using the modified GS process. In any event, in none of the experiments stability problems were encountered related to the use of a CGS process instead of a MGS process. For a more detailed discussion on using either CGS or MGS for the oblique projections, see [78].

Shown in Algorithm 4.2 is the reformulated variant IDR(s)-minsync. In the following, the two phases of the new IDR(s) variant are discussed separately, where we often specifically refer to line numbers of both Algorithm 4.1 and Algorithm 4.2.

1. The dimension reduction step. In the following, let $\mathbf{r}_n$ be the last intermediate residual in $\mathcal{G}_k$ before the dimension reduction step. In accordance with condition (4.4), this residual is orthogonal to all columns of $\widetilde{\mathbf{R}}$ (cf. $\mathbf{r}_{n+s+1}$ from (4.4)) and therefore we have $\mathbf{r}_n \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}$. Using the IDR Theorem (i.e., Theorem 4.1) the first residual $\mathbf{r}_{n+1}$ in $\mathcal{G}_{k+1}$ (also known as the *primary* residual, see Section 5.2.1 for more details) is thus computed as

$$\mathbf{r}_{n+1} = (\mathbf{I} - \omega_{k+1}\mathbf{A}\mathbf{B}^{-1})\mathbf{r}_n. \qquad \text{(line 35 of Algorithm 4.2)} \tag{4.5}$$

Premultiplying this expression with $\mathbf{A}^{-1}$ results in the corresponding recursion for the iterate

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \omega_{k+1}\mathbf{B}^{-1}\mathbf{r}_n, \qquad \text{(line 36 of Algorithm 4.2)} \tag{4.6}$$

which is essentially preconditioned Richardson. A typical (minimal residual) choice of ω_{k+1} is

$$\omega_{k+1} = \arg\min_{\omega} ||(\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}_n|| \tag{4.7}$$

$$= \frac{(\mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n)^*\mathbf{r}_n}{(\mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n)^*\mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n}. \tag{4.8}$$

By reordering operations, the computation of ω_{k+1} can be combined with the computation of $\phi = \widetilde{\mathbf{R}}^*\mathbf{r}$ in line 7 from Algorithm 4.1 as follows. Premultiplying the recursion for computing the new residual (4.5) with $\widetilde{\mathbf{R}}^*$ gives

$$\widetilde{\mathbf{R}}^*\mathbf{r}_{n+1} = \widetilde{\mathbf{R}}^*\mathbf{r}_n - \omega_{k+1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n \tag{4.9}$$

$$= -\omega_{k+1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n, \tag{4.10}$$

since $\widetilde{\mathbf{R}}^*\mathbf{r}_n = \mathbf{0}$ by construction. Setting $\mathbf{t}_n = \mathbf{A}\mathbf{B}^{-1}\mathbf{r}_n$, the computation of $\mathbf{t}_n^*\mathbf{r}_n, \mathbf{t}_n^*\mathbf{t}_n$, and $\widetilde{\mathbf{R}}^*\mathbf{t}_n$ can then be combined (line 34 of Algorithm 4.2).

2. Generating s additional vectors in $\mathcal{G}_{k+1}$. In addition to the standard orthogonalisation step performed in IDR(s) for computing a vector $\mathbf{v} \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}$ (line 9–11 in Algorithm 4.2), the goal is to enforce the extra orthogonality conditions (4.3) and (4.4) to the newly computed vectors $\mathbf{g}$ and residuals $\mathbf{r}$ in $\mathcal{G}_{k+1} \subset \mathcal{G}_k$. In practice, this means that there are now essentially *two* main

Algorithm 4.2 IDR(s)-minsync with bi–orthogonalisation of intermediate residuals and with a single synchronisation point per MV.

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

1: // *Initialisation*
2: $\mathbf{G} = \mathbf{U} = \mathbf{0} \in \mathbb{C}^{n\times s}; \mathbf{M}_l = \mathbf{I} \in \mathbb{C}^{s\times s}, \mathbf{M}_t = \mathbf{M}_c = \mathbf{0}$: see (4.18); $\omega = 1$
3: Compute $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
4: $\phi = \widetilde{\mathbf{R}}^*\mathbf{r}, \phi = (\phi_1, \ldots, \phi_s)^\top$
5: // *Loop over nested* $\mathcal{G}_k$ *spaces,* $k = 0, 1, \ldots$
6: **while** $||\mathbf{r}|| > \varepsilon$ **do**
7: // *Compute* s *linearly independent vectors* $\mathbf{g}_j$ *in* $\mathcal{G}_k$
8: **for** $j = 1$ to s **do**
9: // *Compute* $\mathbf{v} \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$
10: Solve $\mathbf{M}_l\gamma_{(j:s)} = \phi_{(j:s)}$
11: $\mathbf{v} = \mathbf{r} - \sum_{i=j}^{s} \gamma_i \mathbf{g}_i$
12: $\widetilde{\mathbf{v}} = \mathbf{B}^{-1}\mathbf{v}$ // *Preconditioning step*
13: $\widehat{\mathbf{u}}_j = \sum_{i=j}^{s} \gamma_i \mathbf{u}_i + \omega\widetilde{\mathbf{v}}$ // *Intermediate vector* $\widehat{\mathbf{u}}_j$
14: $\widehat{\mathbf{g}}_j = \mathbf{A}\widehat{\mathbf{u}}_j$ // *Intermediate vector* $\widehat{\mathbf{g}}_j$
15: $\boxed{\psi = \widetilde{\mathbf{R}}^*\widehat{\mathbf{g}}_j}$ // *s inner products (combined)*
16: Solve $\mathbf{M}_t\alpha_{(1:j-1)} = \psi_{(1:j-1)}$
17: // *Make* $\widehat{\mathbf{g}}_j$ *orthogonal to* $\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{j-1}$ *and update* $\widehat{\mathbf{u}}_j$ *accordingly*
18: $\mathbf{g}_j = \widehat{\mathbf{g}}_j - \sum_{\ell=1}^{j-1} \alpha_\ell \mathbf{g}_\ell, \quad \mathbf{u}_j = \widehat{\mathbf{u}}_j - \sum_{\ell=1}^{j-1} \alpha_\ell \mathbf{u}_\ell$
19: // *Update column* j *of* $\mathbf{M}_l$
20: $\mu^l_{i,j} = \psi_i - \sum_{\ell=1}^{j-1} \alpha_\ell \mu^c_{i,\ell}$ for $i = j, \ldots, s$
21: // *Make* $\mathbf{r}$ *orthogonal to* $\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_j$ *and update* $\mathbf{x}$ *accordingly*
22: $\beta = \phi_j / \mu^l_{j,j}$
23: $\mathbf{r} = \mathbf{r} - \beta\mathbf{g}_j$
24: $\mathbf{x} = \mathbf{x} + \beta\mathbf{u}_j$
25: // *Update* $\phi \equiv \widetilde{\mathbf{R}}^*\mathbf{r}$
26: **if** $j + 1 \leq s$ **then**
27: $\phi_i = 0$ for $i = 1, \ldots, j$
28: $\phi_i = \phi_i - \beta\mu^l_{i,j}$ for $i = j + 1, \ldots, s$
29: **end if**
30: **end for**
31: // *Entering* $\mathcal{G}_{k+1}$. *Note:* $\mathbf{r} \perp \widetilde{\mathbf{R}}$
32: $\widetilde{\mathbf{v}} = \mathbf{B}^{-1}\mathbf{r}$ // *Preconditioning step*
33: $\mathbf{t} = \mathbf{A}\widetilde{\mathbf{v}}$
34: $\boxed{\omega = (\mathbf{t}^*\mathbf{r})/(\mathbf{t}^*\mathbf{t}); \phi = -\widetilde{\mathbf{R}}^*\mathbf{t}}$ // *s + 2 inner products (combined)*
35: $\mathbf{r} = \mathbf{r} - \omega\mathbf{t}$
36: $\mathbf{x} = \mathbf{x} + \omega\widetilde{\mathbf{v}}$
37: $\phi = \omega\phi$
38: **end while**

orthogonalisations that need to be performed. Since $\mathcal{G}_{k+1} \subset \mathcal{G}_k$, these two orthogonalisations use vectors $\mathbf{g}$ that are either in both $\mathcal{G}_{k+1}$ and $\mathcal{G}_k$ or only in $\mathcal{G}_k$.

In the following, let $j = 1, 2, \ldots, s$ and let $\widetilde{\mathbf{R}}_j \equiv [\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_j]$. Suppose that after $n + j$ iterations we have exactly $s - j + 1$ vectors $\mathbf{g}_i, i = n + j - s - 1, \ldots, n - 1$ in $\mathcal{G}_k$ and $j - 1$ vectors $\mathbf{g}_i, i = n + 1, \ldots, n + j - 1$ in $\mathcal{G}_{k+1}$, which gives a total of s vectors $\mathbf{g}_i$. In addition, suppose that we have s corresponding vectors $\mathbf{u}_i$ such that $\mathbf{g}_i = \mathbf{A}\mathbf{u}_i$ for all i.

From the dimension reduction step we have a residual $\mathbf{r}_{n+1} \in \mathcal{G}_{k+1}$. Also, let $\widehat{\mathbf{g}}_{n+j} \in \mathcal{G}_{k+1}$. Then the two main orthogonalisations that need to be performed are

$$\begin{cases} \mathbf{v}_{n+j} = \mathbf{r}_{n+j} - \sum_{i=j}^{s} \gamma_i \mathbf{g}_{n+i-s-1} & \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}; \quad \text{(“standard”, line 11 of Algorithm 4.2)} \\ \mathbf{g}_{n+j} = \widehat{\mathbf{g}}_{n+j} - \sum_{i=1}^{j-1} \alpha_i \mathbf{g}_{n+i} & \in \mathcal{G}_{k+1} \cap \widetilde{\mathbf{R}}_{j-1}^{\perp}. \quad \text{(additional, line 18 of Algorithm 4.2)} \end{cases} \tag{4.11}$$

The γ_i's are chosen such that $\mathbf{v}_{n+j} \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}$ and the α_i's are chosen such that $\mathbf{g}_{n+j} \in \mathcal{G}_{k+1} \cap \widetilde{\mathbf{R}}_{j-1}^{\perp}$ (i.e., condition (4.3)).

The intermediate update $\widehat{\mathbf{u}}_{n+j}$ for the iterate and the intermediate vector $\widehat{\mathbf{g}}_{n+j} \in \mathcal{G}_{k+1}$ using explicit multiplication by $\mathbf{A}$ are computed according to

$$\begin{cases} \widehat{\mathbf{u}}_{n+j} = \sum_{i=j}^{s} \gamma_i \mathbf{u}_{n+i-s-1} + \omega_{k+1} \mathbf{B}^{-1} \mathbf{v}_{n+j}; & \text{(line 13 of Algorithm 4.2)} \\ \widehat{\mathbf{g}}_{n+j} = \mathbf{A} \widehat{\mathbf{u}}_{n+j}. & \text{(line 14 of Algorithm 4.2)} \end{cases} \tag{4.12}$$

In the implementation of IDR(s)-biortho from Algorithm 4.1, the vector $\widehat{\mathbf{g}}_{n+j}$ is subsequently orthogonalised against $\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{j-1}$ using a MGS process (lines 15–19 in Algorithm 4.1), while $\mathbf{v}_{n+j}$ is orthogonalised using a CGS process (lines 9–10 in Algorithm 4.1).

By performing *both* oblique projections in (4.11) using a CGS process, it will be shown that in each iteration step j only s (combined) inner products have to be computed and that the rest of the inner products can be computed using scalar updates.

Using the orthogonality condition (4.3), define the $(s-j+1) \times (s-j+1)$ and $(j-1) \times (j-1)$ lower triangular matrices $\mathbf{M}_l$ and $\mathbf{M}_t$ as

$$\mathbf{M}_l \equiv \left[\mu_{i,k}^{l}\right] = \begin{cases} \widetilde{\mathbf{r}}_i^* \mathbf{g}_{n+k-s-1} & \text{for } j \le k \le i \le s; \\ 0 & \text{otherwise,} \end{cases} \qquad (\mathbf{g} \in \mathcal{G}_k) \tag{4.13}$$

and

$$\mathbf{M}_t \equiv \left[\mu_{i,k}^{t}\right] = \begin{cases} \widetilde{\mathbf{r}}_i^* \mathbf{g}_{n+k} & \text{for } 1 \le k \le i \le j-1; \\ 0 & \text{otherwise,} \end{cases} \qquad (\mathbf{g} \in \mathcal{G}_{k+1} \subset \mathcal{G}_k) \tag{4.14}$$

respectively. Using the orthogonality condition (4.4), define the $s \times 1$ column vectors ϕ and ψ as

$$\phi_i = \begin{cases} \widetilde{\mathbf{r}}_i^* \mathbf{r}_{n+j} & \text{for } j \le i \le s; \\ 0 & \text{otherwise,} \end{cases} \tag{4.15}$$

$$\psi_i = \widetilde{\mathbf{r}}_i^* \widehat{\mathbf{g}}_{n+j}, \quad \text{for } 1 \le i \le s. \qquad \text{(line 15 of Algorithm 4.2)} \tag{4.16}$$

Using these definitions, the following two small lower triangular systems have to be solved in order to perform the oblique projections (4.11):

$$\begin{cases} \mathbf{M}_l \gamma_{(j:s)} & = \phi_{(j:s)}; \qquad \text{(line 10 of Algorithm 4.2)} \\ \mathbf{M}_t \alpha_{(1:j-1)} & = \psi_{(1:j-1)}. \qquad \text{(line 16 of Algorithm 4.2)} \end{cases} \tag{4.17}$$

Here, the notation $\phi_{(m:n)}$ denotes the column vector $[\phi_m, \phi_{m+1}, \ldots, \phi_n]^\top$.

Most of the inner products that are computed during iteration step j can be stored in a single lower triangular matrix $\mathbf{M}$. To be more precise, define at the start of iteration step j the following $s \times s$ lower triangular matrix $\mathbf{M}$, consisting of the three submatrices M_t, M_l, and M_c:

$$\mathbf{M} \equiv \begin{bmatrix} \mathbf{M}_t & \varnothing \\ \mathbf{M}_c & \mathbf{M}_l \end{bmatrix} = \left[\begin{array}{ccc|ccc} \mu^t_{1,1} & 0 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & 0 & & & \vdots \\ \mu^t_{j-1,1} & \cdots & \mu^t_{j-1,j-1} & \ddots & & \vdots \\ \hline \mu^c_{j,1} & \cdots & \mu^c_{j,j-1} & \mu^l_{j,j} & 0 & 0 \\ \vdots & & \vdots & \vdots & \ddots & 0 \\ \mu^c_{s,1} & \cdots & \mu^c_{s,j-1} & \mu^l_{s,j} & \cdots & \mu^l_{s,s} \end{array}\right], \tag{4.18}$$

where $\mathbf{M}_c$ is defined as the $(s-j+1) \times (j-1)$ block matrix

$$\mathbf{M}_c \equiv \left[\mu^c_{i,k}\right] = \widetilde{\mathbf{r}}_i^* \mathbf{g}_{n+k} \quad \text{for} \quad j \le i \le s, \quad 1 \le k \le j-1. \tag{4.19}$$

We are now ready to compute the vector $\mathbf{v}_{n+j} \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$ as follows. If

$$\gamma_{(j:s)} = \mathbf{M}_l^{-1} \phi_{(j:s)}, \quad \mathbf{v}_{n+j} = \mathbf{r}_{n+j} - \sum_{i=j}^{s} \gamma_i \mathbf{g}_{n+i-s-1}, \quad \text{(lines 10–11 in Algorithm 4.2)} \tag{4.20}$$

then

$$\mathbf{v}_{n+j} \perp \widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_j. \tag{4.21}$$

Also, to compute the vector $\mathbf{g}_{n+j} \in \mathcal{G}_{k+1} \cap \widetilde{\mathbf{R}}_{j-1}^\perp$ and corresponding update $\mathbf{u}_{n+j}$, let

$$\alpha_{(1:j-1)} = \mathbf{M}_t^{-1} \psi_{(1:j-1)}; \tag{4.22}$$

$$\mathbf{g}_{n+j} = \widehat{\mathbf{g}}_{n+j} - \sum_{\ell=1}^{j-1} \alpha_\ell \mathbf{g}_{n+\ell}; \tag{4.23}$$

$$\mathbf{u}_{n+j} = \widehat{\mathbf{u}}_{n+j} - \sum_{\ell=1}^{j-1} \alpha_\ell \mathbf{u}_{n+\ell}, \tag{4.24}$$

then

$$\mathbf{g}_{n+j} \perp \widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{j-1} \quad \text{and} \quad \mathbf{u}_{n+j} \perp_{\mathbf{A}} \widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{j-1}, \tag{4.25}$$

which is exactly condition (4.3).

To efficiently compute the new column j of $\mathbf{M}$ using a scalar update, premultiply the recurrence for $\mathbf{g}_{n+j}$ with $\widetilde{\mathbf{r}}_i^*$ to obtain

$$\widetilde{\mathbf{r}}_i^* \mathbf{g}_{n+j} = \widetilde{\mathbf{r}}_i^* \widehat{\mathbf{g}}_{n+j} - \sum_{\ell=1}^{j-1} \alpha_\ell \widetilde{\mathbf{r}}_\ell^* \mathbf{g}_{n+\ell} \tag{4.26}$$

$$\mu^l_{i,j} = \psi_i - \sum_{\ell=1}^{j-1} \alpha_\ell \mu^c_{i,\ell} \quad \text{for} \quad i = j, \ldots, s, \tag{4.27}$$

where the second expression uses the block matrix $\mathbf{M}_c$ from (4.18).

To summarise, this gives for step j while referring to the line numbers in Algorithm 4.2:

$$\begin{aligned}
\psi &= \widetilde{\mathbf{R}}^*\widehat{\mathbf{g}}_j; && \text{(line 15, } s \text{ combined inner products)} \\
\alpha_{(1:j-1)} &= \mathbf{M}_t^{-1}\psi_{(1:j-1)}; && \text{(line 16, lower triangular system } \mathbf{M}_t^{-1}) \\
\mathbf{g}_{n+j} &= \widehat{\mathbf{g}}_{n+j} - \sum_{\ell=1}^{j-1}\alpha_\ell \mathbf{g}_{n+\ell}; && \text{(line 18, orthogonalise against } \widetilde{\mathbf{r}}_1,\dots,\widetilde{\mathbf{r}}_{j-1}) \\
\mathbf{u}_{n+j} &= \widehat{\mathbf{u}}_{n+j} - \sum_{\ell=1}^{j-1}\alpha_\ell \mathbf{u}_{n+\ell}; && \text{(line 18, } \mathbf{A}\text{–orthogonalise against } \widetilde{\mathbf{r}}_1,\dots,\widetilde{\mathbf{r}}_{j-1}) \\
\mu^l_{i,j} &= \psi_i - \sum_{\ell=1}^{j-1}\alpha_\ell \mu^c_{i,\ell}, \quad i = j,\dots,s. && \text{(line 20, new column } j \text{ of } \mathbf{M} \text{ using } \mathbf{M}_c)
\end{aligned}$$

In accordance with condition (4.4), the updated residual $\mathbf{r}_{n+j+1}$ can be made orthogonal to $\widetilde{\mathbf{r}}_1,\dots,\widetilde{\mathbf{r}}_j$ by

$$\mathbf{r}_{n+j+1} = \mathbf{r}_{n+j} - \frac{\phi_j}{\mu^l_{j,j}}\mathbf{g}_{n+j}, \qquad \text{(line 23 of Algorithm 4.2)} \tag{4.28}$$

since

$$\widetilde{\mathbf{r}}_j^*\mathbf{r}_{n+j+1} = \widetilde{\mathbf{r}}_j^*\mathbf{r}_{n+j} - \frac{\phi_j}{\mu^l_{j,j}}\widetilde{\mathbf{r}}_j^*\mathbf{g}_{n+j} \tag{4.29}$$

$$= \widetilde{\mathbf{r}}_j^*\mathbf{r}_{n+j} - \widetilde{\mathbf{r}}_j^*\mathbf{r}_{n+j} = 0. \tag{4.30}$$

Premultiply (4.28) with $\mathbf{A}^{-1}$ to obtain the corresponding update to the iterate

$$\mathbf{x}_{n+j+1} = \mathbf{x}_{n+j} + \frac{\phi_j}{\mu^l_{j,j}}\mathbf{u}_{n+j}. \qquad \text{(line 24 of Algorithm 4.2)} \tag{4.31}$$

Finally, premultiplying (4.28) with $\widetilde{\mathbf{r}}_i^*$ for $i = j+1,\dots,s$ gives the scalar update for the vector ϕ,

$$\phi_i = \phi_i - \frac{\phi_j}{\mu^l_{j,j}}\mu^l_{i,j}, \qquad i = j+1,\dots,s, \qquad \text{(line 28 of Algorithm 4.2)} \tag{4.32}$$

which concludes the generation of the s vectors for $\mathcal{G}_{k+1}$.

Therefore, in each iteration step j the s combined inner products $\widetilde{\mathbf{R}}^*\widehat{\mathbf{g}}_j$ in the vector ψ have to be computed. The remaining inner products $\widetilde{\mathbf{R}}^*\mathbf{r}$ in the vector ϕ and the inner products $\widetilde{\mathbf{R}}^*\mathbf{g}$ in the small matrix $\mathbf{M}$ can then be computed using solely scalar updates.

As with Algorithm 4.1, the computation of the (now solely combined) inner products is highlighted by boxes in Algorithm 4.2. The small systems in lines 10 and 16 of Algorithm 4.2 involving $\mathbf{M}_l$ and $\mathbf{M}_t$ respectively are lower triangular and can be solved efficiently using forward substitution. Note that the system in line 16 involves the first $j-1$ elements of the column vector ψ, while in line 20 the remaining elements are used. This shows that there is now a *single* global synchronisation point per MV. The number of operations is the same as the original IDR(s)-biortho method from Algorithm 4.1.

4.3 Parallelising IDR(s) methods

Similar to other iterative subspace methods, the IDR(s) variants in Algorithm 4.1 and Algorithm 4.2 are composed of several key building blocks (cf. Section 1.10 on page 22). These are (referring to line numbers of Algorithm 4.2): matrix–vector multiplications (lines 14 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 plus small system solves involving $\mathbf{M}$ (lines 10, 16, 20, 22, 28, 37). All these operations can be straightforwardly parallelised on distributed memory computers by making a block partitioning of the coefficient matrix and of the vectors:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1p} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{p1} & \mathbf{A}_{p2} & \cdots & \mathbf{A}_{pp} \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_p \end{bmatrix}. \tag{4.33}$$

Note that in the type of application we are interested in almost all off–diagonal blocks are zero. 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.
- The matrix–vector products are computed by performing local products with the sub-matrices. Multiplication with a diagonal block $\mathbf{A}_{ii}$ does not require communication, but multiplication with an off–diagonal block requires communication with another processor.
- The scalar operations and small system solves involving $\mathbf{M}$ are inexpensive and are not parallelised, but are performed by all processors.
- In some experiments performed in this chapter, the preconditioning operation consists of a parallel asynchronous preconditioning as described in Sect 4.5.

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 Algorithm 4.2 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 MV.

4.4 Choosing s and $\widetilde{\mathbf{R}}$

The parameter s and the test matrix $\widetilde{\mathbf{R}}$ can be chosen freely in IDR(s) methods. In this section this freedom is exploited in order to minimise parallel execution time. In Section 4.4.1 performance models will be given that are used to estimate the optimal s in IDR(s) methods. Section 4.4.2 describes a method of minimising the work and storage involving operations with the test matrix $\widetilde{\mathbf{R}}$ in a multi–cluster environment.

4.4.1 Parallel performance models for IDR(s)

Performance models are derived to estimate the total execution time of the IDR(s)-biortho variant and the IDR(s)-minsync variant. The model is applied to both a single cluster and to a multi–cluster. After first presenting a general model for the total execution time of parallel IDR(s) methods, specific expressions are given for the communication steps of the IDR(s) algorithms.

A general model. The overall performance model is based on message passing communication. It is assumed that no load imbalance occurs and as a result the parallel computing time for a fixed–sized problem on p processors can be described in general by

$$T_{\text{total}}(p) = \frac{T(1)}{p} + T_{\text{comm}}, \tag{4.34}$$

where T_{comm} denotes the total communication time of the algorithm and $T(1)$ the computation time on a single processor.

Generally speaking, there are two operations that require communication in parallel iterative methods, which are the inner product (both single and combined) and the matrix–vector product. The first operation requires global communication, whereas the second operation requires nearest neighbour communication for the current application. No preconditioning is included in the performance model.

The following simple linear model for the time to communicate a message of k bytes length is assumed,

$$T_{\text{mess}} = l + k/b, \tag{4.35}$$

in which l is the latency and b the bandwidth. Using this linear model, expressions for the communication time of inner products (denoted by T_{dots}) and matrix-vector multiplication (denoted by T_{mmult}) in each IDR(s) cycle can be derived. These expressions will in general depend on both p and s.

To formulate the complete performance model, the total number of IDR(s) cycles has to be determined. As mentioned before, it can be shown that in the generic case, IDR(s) methods terminate within $n(1 + \frac{1}{s})$ matrix–vector multiplications in exact arithmetic, or equivalently, within $\frac{n}{s}$ dimension reduction steps [120, Section 3]. In practical applications, the iteration process will exhibit much faster (superlinear) convergence rates according to $\frac{\widehat{n}}{s}$, where $\widehat{n} \ll n$. Note that the parameter $\widehat{n}$ can be estimated using a single experiment. However, it is shown below that this parameter is not needed to compute the optimal s.

To summarise, the total theoretical computing time on p processors for a given value of s is then (cf. (4.34))

$$T_{\text{total}}(p, s) = \frac{\widehat{n}}{s} \times \underbrace{\left[\frac{\widetilde{T}(1, s)}{p} + T_{\text{dots}}(p, s) + T_{\text{mmult}}(p, s) \right]}_{\text{time of a parallel IDR}(s)\text{ cycle}}, \tag{4.36}$$

where $\widetilde{T}(1, s)$ is the computing time of one *sequential* IDR(s) cycle. This quantity is a function of s and can be estimated by counting the number of floating point operations and using the value for the computational speed of a single processor.

By minimising (4.36), the optimal value of s and p for solving the test problem in a minimal amount of time can be estimated. Note that since the parameter $\widehat{n}$ is a constant, it does not play any role in minimising (4.36) and only *a priori* information is needed.

Models for different architectures. The performance model (4.36) will be applied to a single cluster and to a multi–cluster for both IDR(s) variants as follows.

For the single cluster model, the values for latency, bandwidth, and computational speed of that cluster are used. In this way, the model is used to compute the optimal s and corresponding number of *nodes* p in a cluster.

In multi–cluster architectures, *inter*cluster latencies are often several orders of magnitude higher than *intra*cluster latencies. For the DAS–3 multi–cluster, these latencies differ by three orders of magnitude, while the values for the bandwidth differ by one order of magnitude. In the model for multi–clusters, intracluster latency and bandwidth is therefore excluded and each cluster is treated as a *single entity*. In this way, the model is used to compute the optimal s and corresponding number of *clusters* P in the DAS–3 multi–cluster. The intercluster latency and bandwidth of the DAS–3 multi–cluster are then used by the performance model. This also means that in the performance model, the computational speed of a cluster is taken to be the combined computational speed of all the nodes in that cluster.

In the following, specific expressions for the communication time of the inner products T_{dots} and the matrix–vector multiplication T_{mmult} will be presented.

The communication time of a properly implemented inner product on a single (tightly–coupled) cluster is given by

$$T_{\text{dot}} = \lceil \log_2(p) \rceil (l + 8/b). \tag{4.37}$$

In the algorithms, some inner products can be combined. The communication time of the combined broadcasting of c partial inner products is then

$$T_{\text{cdot}} = \lceil \log_2(p) \rceil (l + 8c/b). \tag{4.38}$$

For the multi–cluster model, we note that the network topology of the DAS–3 multi–cluster is structured like a ring (see Section 4.6.1 for more details). However, the number of clusters that is used is always relatively small, so it is assumed that the hierarchical model for the inner product also applies to the multi–cluster setting.

A minimal residual strategy is used to compute ω, so the *total* communication time $T_{\text{dots}}(p,s)$ spent on inner products in each cycle of IDR(s)-biortho is then

$$T_{\text{dots}}^{\text{biortho}}(p,s) = \lceil \log_2(p) \rceil \left[\left(\tfrac{1}{2} s(s+1) + 2 \right) l + 8 \left(s^2 + s + 2 \right) / b \right] \tag{4.39}$$

$$= \lceil \log_2(p) \rceil \left[\mathcal{O}(s^2) l + \mathcal{O}(s^2)/b \right] \tag{4.40}$$

and for the new IDR(s)-minsync variant it is

$$T_{\text{dots}}^{\text{minsync}}(p,s) = \lceil \log_2(p) \rceil \left[(s+1) l + 8 \left(s^2 + s + 2 \right) / b \right] \tag{4.41}$$

$$= \lceil \log_2(p) \rceil \left[\mathcal{O}(s) l + \mathcal{O}(s^2)/b \right]. \tag{4.42}$$

The expressions (4.40) and (4.42) show that for relatively large values of bandwidth b, the communication time spent on inner products in each cycle grows differently with s in the two variants. For larger values of latency l, the time per cycle is almost linear in s for IDR(s)-minsync, while the time per cycle behaves quadratically in s for IDR(s)-biortho.

Expressions for the communication time of the matrix–vector multiplication $T_{\text{mmult}}(p,s)$ can be derived in a similar manner. For a single cluster, the cubical domain is partitioned into rectangular cuboids and each subdomain is assigned to a single node. Supposing that $n_x = n_y = n_z$, the communication time spent on the $s+1$ matrix–vector multiplications in each IDR(s) cycle for an interior subdomain is (in both IDR(s) variants)

$$T_{\text{mmult}}(p,s) = 6(s+1) \left(l + \frac{8 n_x^2}{b \sqrt[3]{p}} \right). \qquad \text{(single cluster)} \tag{4.43}$$

Considering again the fact that the DAS–3 network has a ring structure, the domain partitioning in the multi–cluster setting can be seen as two–dimensional. That is, each subdomain is assigned to a single cluster. Also, each interior subdomain has two neighbouring subdomains, which gives for the matrix–vector multiplication (again in both IDR(s) variants)

$$T_{\text{mmult}}(s) = 2(s+1)(l + 8n_x n_y/b). \qquad \text{(multi–cluster)} \tag{4.44}$$

Note that this expression is independent of the number of clusters.

Using this information, the expression (4.36) can then be minimised which allows for *a priori* estimation of the optimal parameter s and corresponding number of nodes/clusters using only problem and machine–based parameters.

This also shows that the performance model can be as complex as one desires, depending wholly on the type of architecture and application. For example, the Multi–BSP model from [134] is a hierarchical performance model for modern multi–core architectures and could be used to find the optimal s of IDR(s) methods on such architectures.

In Section 4.6 the performance models given here will be compared to numerical experiments on the DAS–3 multi–cluster. For experiments using nodes from one cluster, the model for a single cluster is used. When using more than one cluster, we switch to the performance model for multi–clusters.

4.4.2 Using piecewise sparse column vectors for $\widetilde{\mathbf{R}}$

In multi–cluster environments such as the DAS–3, the latency between clusters may be several orders of magnitude larger than the intracluster latency. The majority of the inner products in the algorithm consist of computing $\widetilde{\mathbf{R}}^*\mathbf{r}$ for some vector $\mathbf{r}$. The $n \times s$ matrix $\widetilde{\mathbf{R}}$ can be chosen arbitrarily and this freedom can be exploited [120, Section 4.1]. By using sparse column vectors for $\widetilde{\mathbf{R}}$, the total cost of the inner products may be reduced significantly in the context of a multi–cluster environment. However, such a strategy may influence the robustness of the algorithm and this phenomenon will be illustrated experimentally.

The outline of the algorithm for the computation of $\widetilde{\mathbf{R}}^*\mathbf{r}$ using sparse column vectors for $\widetilde{\mathbf{R}}$ is as follows. Each cluster in the multi–cluster is considered as one (large) subdomain. The columns of $\widetilde{\mathbf{R}}$ are chosen in such a manner that they are nonzero on one of these subdomains and zero on the other subdomains.

A *coordinator* node is randomly chosen on each cluster and each node computes its local inner product with its local part of $\mathbf{r}$. A reduction operation is then performed locally on each cluster and the result is gathered on the coordinator node. The coordinators exchange the partial inner products across the slow intercluster connections, combining them to make the total inner product. Finally, this result is broadcasted locally within each cluster. Therefore, the number of times that data is sent between the clusters is reduced considerably.

For ease of implementation, the value s is chosen as an integer multiple of the total number of clusters γ in the grid. Using sparse column vectors decreases the computational work and the storage requirements on each cluster. Instead of computing s inner products of length n, the total computational cost is reduced to s inner products of length n/γ. Note that this approach is valid for arbitrary s.

As an example, suppose that there are three clusters in the multi–cluster and that each cluster has two nodes, giving six nodes $\{a, b, c, d, e, f\}$ in total. If $s = 6$, the computation of $\widetilde{\mathbf{R}}^*\mathbf{r}$

$$
\begin{array}{|l|}
\hline
\textbf{for } j = 1 \text{ to } s \textbf{ do} \\
\quad \mathbf{v} = \mathbf{r} - \mathbf{G}\gamma \\
\quad \mathbf{u} = \omega\mathbf{v} + \mathbf{U}\gamma \\
\quad \mathbf{g} = \mathbf{A}\mathbf{B}^{-1}\mathbf{u} \\
\quad \mathbf{r} = \mathbf{r} - \mathbf{g} \\
\quad \mathbf{y} = \mathbf{y} + \mathbf{u} \\
\textbf{end for} \\
\mathbf{r} = \mathbf{r} - \omega\mathbf{A}\mathbf{B}^{-1}\mathbf{r} \\
\mathbf{y} = \mathbf{y} + \omega\mathbf{r} \\
\hline
\end{array}
\quad
\begin{array}{|l|}
\hline
\textbf{for } j = 1 \text{ to } s \textbf{ do} \\
\quad \mathbf{v} = \mathbf{r} - \mathbf{G}\gamma \\
\quad \mathbf{B}^{-1}\mathbf{u} = \omega\mathbf{B}^{-1}\mathbf{v} + \mathbf{B}^{-1}\mathbf{U}\gamma \\
\quad \mathbf{g} = \mathbf{A}\mathbf{B}^{-1}\mathbf{u} \\
\quad \mathbf{r} = \mathbf{r} - \mathbf{g} \\
\quad \mathbf{B}^{-1}\mathbf{y} = \mathbf{B}^{-1}\mathbf{y} + \mathbf{B}^{-1}\mathbf{u} \\
\textbf{end for} \\
\mathbf{r} = \mathbf{r} - \omega\mathbf{A}\mathbf{B}^{-1}\mathbf{r} \\
\mathbf{B}^{-1}\mathbf{y} = \mathbf{B}^{-1}\mathbf{y} + \omega\mathbf{B}^{-1}\mathbf{r} \\
\hline
\end{array}
\quad
\begin{array}{|l|}
\hline
\textbf{for } j = 1 \text{ to } s \textbf{ do} \\
\quad \mathbf{v} = \mathbf{r} - \mathbf{G}\gamma \\
\quad \underline{\mathbf{u}} = \omega\mathbf{B}^{-1}\mathbf{v} + \underline{\mathbf{U}}\gamma \\
\quad \mathbf{g} = \mathbf{A}\underline{\mathbf{u}} \\
\quad \mathbf{r} = \mathbf{r} - \mathbf{g} \\
\quad \mathbf{x} = \mathbf{x} + \underline{\mathbf{u}} \\
\textbf{end for} \\
\mathbf{r} = \mathbf{r} - \omega\mathbf{A}\mathbf{B}^{-1}\mathbf{r} \\
\mathbf{x} = \mathbf{x} + \omega\mathbf{B}^{-1}\mathbf{r} \\
\hline
\end{array}
$$

Figure 4.1: Varying preconditioner in IDR(s), shown for a single IDR cycle.

using the sparse column vectors $\widetilde{\mathbf{r}}_1, \dots, \widetilde{\mathbf{r}}_6$ for $\widetilde{\mathbf{R}}$ has the following form:

$$
\widetilde{\mathbf{R}}^*\mathbf{r} = \left[\begin{array}{cc|cc|cc}
\widetilde{\mathbf{r}}_1^a & \widetilde{\mathbf{r}}_1^b & & & & \\
\widetilde{\mathbf{r}}_2^a & \widetilde{\mathbf{r}}_2^b & & & & \\
& & \widetilde{\mathbf{r}}_3^c & \widetilde{\mathbf{r}}_3^d & & \\
& & \widetilde{\mathbf{r}}_4^c & \widetilde{\mathbf{r}}_4^d & & \\
& & & & \widetilde{\mathbf{r}}_5^e & \widetilde{\mathbf{r}}_5^f \\
& & & & \widetilde{\mathbf{r}}_6^e & \widetilde{\mathbf{r}}_6^f
\end{array}\right] \times \left[\begin{array}{c}
\mathbf{r}^a \\ - \\ \mathbf{r}^b \\ \hline \mathbf{r}^c \\ - \\ \mathbf{r}^d \\ \hline \mathbf{r}^e \\ - \\ \mathbf{r}^f
\end{array}\right]. \tag{4.45}
$$

In this case, parts of two columns of $\widetilde{\mathbf{R}}$ are nonzero on one cluster and zero on the remaining two clusters. Therefore, two partial local inner products are computed by each node and the results are gathered on one of the two nodes of the cluster. These results are then exchanged between the three clusters and broadcasted locally to the two nodes in each cluster.

4.5 Combining IDR(s) with asynchronous preconditioning

Asynchronous 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 (for more details, see Chapters 1 and 3). Referring to the IDR(s)-minsync algorithm in Algorithm 4.2, asynchronous preconditioning can be performed by applying a parallel asynchronous iterative method to the system $\mathbf{A}\widetilde{\mathbf{v}} = \mathbf{v}$ (line 12 in Algorithm 4.2) and to $\mathbf{A}\widetilde{\mathbf{v}} = \mathbf{r}$ (line 32 in Algorithm 4.2) for a fixed amount of time $T_{\max}$.

The asynchronous preconditioning step consists of a random (typically nonlinear) process (cf. (3.1) on page 48),

$$
\widetilde{\mathbf{v}} = \mathcal{B}(\mathbf{r}), \qquad \mathcal{B} : \mathbb{C}^n \to \mathbb{C}^n, \tag{4.46}
$$

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?

In the following it is shown that the recursions of IDR(s) remain valid within the context of a varying preconditioner. In addition, it also shows how a right preconditioner is introduced

into IDR(s) methods. Given in Figure 4.1 are the *simplified* recursions of IDR(s) methods (e.g., without the additional bi–orthogonalisation steps of Algorithm 4.1 and Algorithm 4.2), showing the three steps of incorporating a right preconditioner in IDR(s). To be more precise, applying a right preconditioner $\mathbf{B}^{-1}$ to the system $\mathbf{Ax} = \mathbf{b}$ gives

$$\mathbf{AB}^{-1}\mathbf{y} = \mathbf{b}, \quad \mathbf{x} = \mathbf{B}^{-1}\mathbf{y}. \tag{4.47}$$

Applying the simplified IDR(s) recursions to the preconditioned system (4.47) gives the leftmost part of Figure 4.1, where the final solution is obtained by $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$. The main problem with this approach is that if the preconditioner $\mathbf{B}$ changes in each iteration step, the computed iterates $\mathbf{y}$ do not correspond to the computed residuals $\mathbf{r}$.

This inconsistency can be remedied by premultiplying $\mathbf{y}$ with $\mathbf{B}^{-1}$, scaling back the iterates (middle part of Figure 4.1). Again the final solution is obtained by $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$. Note that as a result, the update $\mathbf{u}$ is also scaled back. Defining $\underline{\mathbf{u}} = \mathbf{B}^{-1}\mathbf{u}$ for the new updates and setting $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$ gives the correct right-preconditioned recursions for IDR(s) (rightmost part of Figure 4.1). The iterate $\mathbf{x}$ and residual $\mathbf{r}$ are now computed in a consistent manner (and basically independent of how the update $\underline{\mathbf{u}}$ is constructed).

Note that according to the theoretical finite termination property of IDR(s), finite termination at the exact solution should occur in the generic case within $n + n/s$ MVs [120]. However, this no longer holds if IDR(s) is used as a flexible method. Nevertheless, the method is still finite for $s = n$, since in that case a complete set of basis vectors for $\mathbb{C}^n$ is generated and the method terminates at the exact solution after n MVs.

4.6 Numerical experiments

In this section, four parallel implementations of IDR–based methods will be investigated:

(i) The IDR(s)-biortho variant given in Algorithm 4.1 using a dense matrix $\widetilde{\mathbf{R}}$ (with $\mathbf{B} \equiv \mathbf{I}$);

(ii) The IDR(s)-minsync variant given in Algorithm 4.2 using a dense matrix $\widetilde{\mathbf{R}}$ (with $\mathbf{B} \equiv \mathbf{I}$);

(iii) The IDR(s)-minsync variant given in Algorithm 4.2 using a sparse matrix $\widetilde{\mathbf{R}}$ as described in Section 4.4.2 (with $\mathbf{B} \equiv \mathbf{I}$);

(iv) The IDR(s)-minsync variant given in Algorithm 4.2 using a dense matrix $\widetilde{\mathbf{R}}$ and an asynchronous preconditioner as described in Sect 4.5 (with $\mathbf{B} \equiv \mathcal{B}$ from (4.46)).

Note that in exact arithmetic, the first two variants produce residuals that are identical in every iteration step. Also, the first three variants do not include preconditioning.

This section is divided into two parts. In the first part (Section 4.6.1–4.6.4), general information is given about the experimental setup which involves some small sequential experiments. The second part (Section 4.6.5–4.6.8) contains results from large parallel experiments using the four variants given above.

In Section 4.6.1 and Section 4.6.2 a description is given of the target hardware and test problem, respectively. In Section 4.6.3 the parameter $\widehat{n}$ for this test problem using unpreconditioned IDR(s) is estimated, along with the (true) computational speed of a single core. In Section 4.6.4 the optimal parameter s using again unpreconditioned IDR(s) for a particular computational environment is computed *a priori* using the performance model from Sect 4.4.1.

In Section 4.6.5 the performance model is compared to the numerical results using variants (i) and (ii). In Section 4.6.6 the experimental results are investigated more closely by comparing the time per cycle to the performance model using variants (i), (ii), and (iii). In Section 4.6.7,

	LU site
1–way latency MPI (μsec)	2.7
max. throughput (MB/sec)	950
Gflops (HPL benchmark [101])	6.9
	DAS–3
WAN latencies (μsec) (average)	990
WAN bandwidth (MB/sec)	5,000

Table 4.1: Specifications DAS–3: all values (except WAN latencies) are obtained from [16].

both strong and weak scalability results are given, again using using variants (i), (ii), and (iii). Finally, Section 4.6.8 contains experimental results using variant (iv).

Experimental setup

4.6.1 Target hardware

The numerical experiments are performed using the distributed ASCI Supercomputer 3 (DAS–3), which is a cluster of five clusters, located at four academic institutions across the Netherlands [107]. The five sites are connected through SURFnet, which is the academic and research network in the Netherlands. Each local cluster is equipped with both 10 Gbps Ethernet and high speed Myri–10G interconnect. However, the TUD site only employs the Ethernet interconnect. If an experiment includes the TUD site, the other sites will automatically switch to the slower Ethernet interconnect. For more details on the five DAS–3 sites, see Section 1.8.2 on page 21.

The network topology of the DAS–3 cluster is structured like a ring, connecting the five sites as follows: TUD, LU, VU, UvA, UvA–MN, and again to the TUD site. Although nodes on some sites may contain multiple cores, we always employ a single core on each node for our computations. Table 4.1 lists values obtained from [16] on latency and bandwidth for the LU site and for the wide–area bandwidth on all five clusters. These values were corroborated by the authors using the Intel MPI Benchmarks suite (IMB v2.3). The value for the WAN latency in Table 4.1 is the average value from several IMB benchmarks performed at different times during the day and is similar to (albeit somewhat below) the values given in [144].

The three unpreconditioned variants are implemented using Open MPI v1.2.1 [71] and level 3 optimisation is used by the underlying GNU C compiler (see Sect 1.7.3 on page 19). The preconditioned variant (iv) is implemented using the CRAC library [47], which is specifically designed to build (partially) asynchronous applications (see Sect 1.7.2 on page 18). All four implementations are matrix–free: the coefficient matrix is not explicitly formed and stored.

4.6.2 Test problem

Consider the following three–dimensional elliptic partial differential equation taken from [113]:

$$\nabla^2 \mathbf{u} + w\mathbf{u}_x = \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{z}), \tag{4.48}$$

defined on the unit cube $[0,1] \times [0,1] \times [0,1]$. The predetermined solution

$$\mathbf{u} = \exp(\mathbf{xyz}) \sin(\pi\mathbf{x}) \sin(\pi\mathbf{y}) \sin(\pi\mathbf{z}) \tag{4.49}$$

# nodes	$p_x \times p_y \times p_z$	# equations
1	$1 \times 1 \times 1$	128^3
2	$2 \times 1 \times 1$	
4	$2 \times 2 \times 1$	
8	$2 \times 2 \times 2$	
16	$4 \times 2 \times 2$	
32	$4 \times 2 \times 4$	
64	$4 \times 4 \times 4$	
96	$6 \times 4 \times 4$	
128	$8 \times 4 \times 4$	

Table 4.2: Processor grids and problem size for the strong scalability experiments.

# nodes	$p_x \times p_y \times p_z$		# equations
	first strategy	second strategy	
30	$5 \times 3 \times 2$	$1 \times 6 \times 5$	398^3
60	$5 \times 4 \times 3$	$2 \times 6 \times 5$	501^3
90	$5 \times 6 \times 3$	$3 \times 6 \times 5$	574^3
120	$5 \times 6 \times 4$	$4 \times 6 \times 5$	631^3
150	$5 \times 6 \times 5$	$5 \times 6 \times 5$	680^3

Table 4.3: Processor grids and problem sizes for the weak scalability experiments.

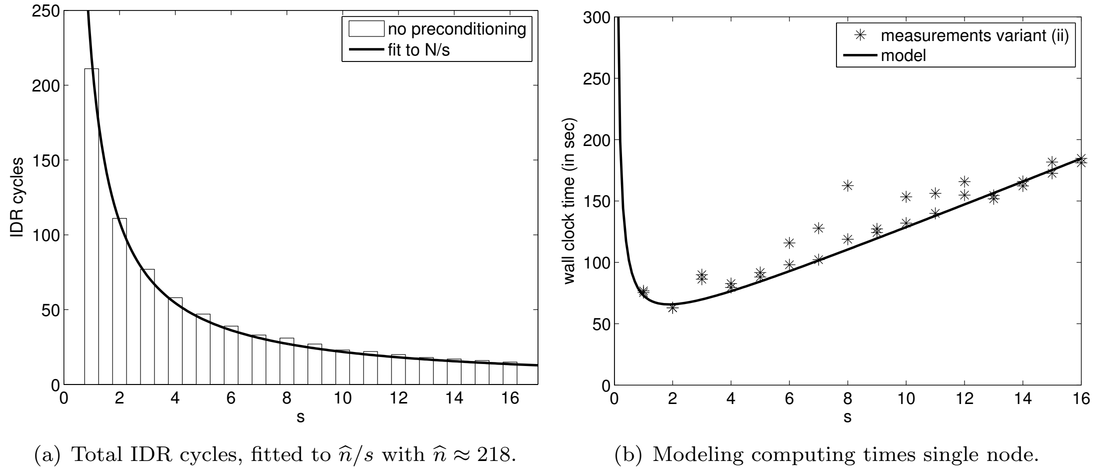
defines the vector $\mathbf{f}$ and Dirichlet boundary conditions are imposed accordingly.

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 $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ 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 and when not specified otherwise then the convection coefficient w is set to 100. The IDR test matrix $\widetilde{\mathbf{R}}$ consists of s orthogonalised random vectors. The (outer) iteration is terminated when $||\mathbf{r}_n||/||\mathbf{r}_0|| \leq \varepsilon \equiv 10^{-6}$ and the initial guess is set to $\mathbf{x}_0 \equiv \mathbf{0}$. At the end of the iteration process convergence is verified by comparing the true residual with the iterated final residual.

Unpreconditioned variants ($\mathbf{B} \equiv \mathbf{I}$). The experiments for investigating strong scalability are performed using the four sites that employ the fast interconnect. On each cluster, 32 nodes are used, which gives a total of 128 nodes for the largest experiment. Experiments that use less than 32 nodes are performed on the LU site and in each subsequent experiment 32 nodes are added with each additional site, in the following order: VU, UvA, and UvA–MN. In this way, the ring structure of the DAS–3 wide–area network is obeyed. The number of grid points in each direction is $n_x = n_y = n_z \equiv 128$, which gives a total problem size of approximately two million equations.

For the weak scalability experiments, 30 nodes per site are used and the TUD cluster is included, which means that in this case the slower interconnect is used on every cluster. The number of equations per node is set to approximately two million equations, yielding the problem sizes shown in Table 4.3.

The computational domain is partitioned using a three–dimensional block partitioning, where



(a) Total IDR cycles, fitted to $\widehat{n}/s$ with $\widehat{n} \approx 218$.

(b) Modeling computing times single node.

Figure 4.2: Estimating model parameters: $\widehat{n}$ and processor speed.

the subdomains are arranged in a Cartesian grid $p_x \times p_y \times p_z$. The nodes are numbered according to

$$p(i,j,k) = p_z^k + (p_y^j - 1)p_z + (p_x^i - 1)p_y p_z \tag{4.50}$$

with $p_x^i \times p_y^j \times p_z^k \in [1, p_x] \times [1, p_y] \times [1, p_z]$. The nodes are mapped sequentially across the clusters, one cluster after another. In the multi–cluster experiments, the domain is partitioned along the x–direction. This partitioning divides the domain into slices and in our experiments, each slice is mapped onto a cluster. Partitioning in this way ensures that adjacent slices in the domain correspond to adjacent clusters in the DAS–3 multi–cluster. Also, this partitioning corresponds to the multi–cluster performance model as discussed in Section 4.4.1.

Using this strategy, Table 4.2 and Table 4.3 list the dimensions of the processor grid for each number of nodes used in the strong and weak scalability experiments, respectively.

As shown in Table 4.3, the problem size for the weak scalability experiments will be increased using two different strategies. In the first strategy the nodes are equally divided between the five DAS–3 sites, starting with $30/5 = 6$ nodes per site for the smallest experiment. This means that there are always five slices, one for each cluster. Also, this allows comparing the use of a dense test matrix $\widetilde{\mathbf{R}}$ and of a sparse test matrix $\widetilde{\mathbf{R}}$.

In the second strategy, one whole site is added each time, starting with the LU site and ending with the TUD site. Here, the number of slices corresponds to the number of clusters. Also, for this experiment only a dense test matrix $\widetilde{\mathbf{R}}$ is used.

Preconditioned variant ($\mathbf{B} \equiv \mathcal{B}$ from (4.46)). 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 (fixed) problem size in this case is $n = 180^3 \approx 6,000,000$ equations and the local systems in the preconditioning iteration are solved using (truncated) GCR(100) [60] with relative accuracy 10^{-1}.

4.6.3 Estimating parameters of performance model

Although the parameter $\widehat{n}$ is not needed to compute the optimal s, we will use it estimate the computational speed of a single core.

	variant (i)		variant (ii)	
# nodes	optimal s	wall clock time	optimal s	wall clock time
1	2	66.14	2	66.14
2	2	33.15	2	33.15
4	2	16.64	2	16.64
8	2	8.378	2	8.376
16	2	4.235	2	4.233
32	2	2.156	2	2.153
64	2	1.111	2	1.107

Table 4.4: *A priori* estimation of s for the LU cluster.

In order to estimate $\widehat{n}$ for the test problem, the total number of dimension reduction steps for $s \in \{1, 2, \ldots, 16\}$ are shown in Figure 4.2(a). These experimental data are obtained using variant (ii) on a single LU node. Variant (i) gives the same results and the data are fitted to the curve $\widehat{n}/s$, giving $\widehat{n} \approx 218$. This shows that the number of IDR(s) cycles behaves in accordance with the theoretical estimate. Interestingly, the value for $\widehat{n}$ is of the same order of magnitude as the number of grid points in each direction $n_x = n_y = n_z \equiv 128$. Note that a *single* (possibly parallel) experiment is sufficient to estimate the parameter $\widehat{n}$.

In order to estimate the effective computational speed of a single core, wall clock times of two executions of variant (ii) on an LU node for each value of s are shown in Figure 4.2(b). Since variants (i) and (ii) have the same number of operations, only results from the second variant are given. The theoretical computing time of the algorithm on a single node is equal to (cf. (4.36))

$$T_{\text{total}}(1, s) = \frac{\widehat{n}}{s} \times \widetilde{T}(1, s). \tag{4.51}$$

Using the obtained value of $\widehat{n}$, the value for the computational speed is estimated by fitting the measurements to the model (solid line in Figure 4.2(b)), which gives a value of 3×10^{-1} Gflops. According to the HPL benchmark (i.e., Table 4.1), the peak performance of a single core is 6.9 Gflops. However, it is not uncommon that only a fraction of the processor's peak performance can be attained in practice. These results also indicate that when using a single core, setting $s = 2$ results in the fastest computing times.

4.6.4 *A priori* estimation of optimal parameter s

By using the expression for the theoretical computing time (4.36) from Section 4.4.1, the optimal parameter s that minimises the total execution time can be computed. Note that every parameter needed to compute these optimal values can be obtained *a priori*. In particular, the parameter $\widehat{n}$ is not required.

As described in Section 4.4.1, estimates for both a single cluster (the LU site) and for a multi–cluster (the DAS–3) will be given. Using the data from Table 4.1, the optimal (rounded value of) s for a given (theoretical) number of nodes of the LU cluster are computed and shown in Table 4.4 for variants (i) and (ii), respectively. Similarly, Table 4.5 shows the optimal s for a given (theoretical) number of clusters in the DAS–3 multi–cluster — with 32 nodes per cluster — for variants (i) and (ii), respectively. Also shown are the corresponding wall clock times.

According to Table 4.4, the optimal s for using one node on the LU cluster is $s = 2$ for both variants. This is in agreement with results from Figure 4.2(b) from Section 4.6.3, where

	variant (i)		variant (ii)	
# clusters	optimal s	wall clock time	optimal s	wall clock time
1	2	2.058	2	2.058
2	2	2.231	3	1.968
3	2	2.203	3	1.772
4	2	2.255	4	1.69
5	2	2.325	4	1.657
6	2	2.398	5	1.639
7	2	2.469	5	1.633

Table 4.5: *A priori* estimation of s for the DAS–3 multi–cluster.

the computing times on a single node were given (i.e., latency equal to *zero*). For that case the optimal value was also $s = 2$.

Table 4.4 also shows that for all number of nodes the optimal value of s is always $s = 2$. In this case, latency is low and the communication cost for generating the vectors of the nested subspaces $\mathcal{G}_k$ is relatively small. Apparently the reduction in number of cycles for larger s is not worth the increased cost of generating more vectors in $\mathcal{G}_k$. Also, there is practically no difference in optimal s and wall clock time for both variants. The reason is that for $s = 1$, both variants are equal in cost and that for very low $s > 1$, the differences in cost are only marginal.

For the multi–cluster model the latency value is much higher. In this case, the differences between the two variants become more apparent, as illustrated by Table 4.5. For variant (i), $s = 2$ is again the optimal value for all number of clusters, but the wall clock time grows with increasing number of clusters. In contrast, using variant (ii) not only gives lower wall clock times, it also shows that increasing the number of clusters (and correspondingly the value of s) results in improved execution times.

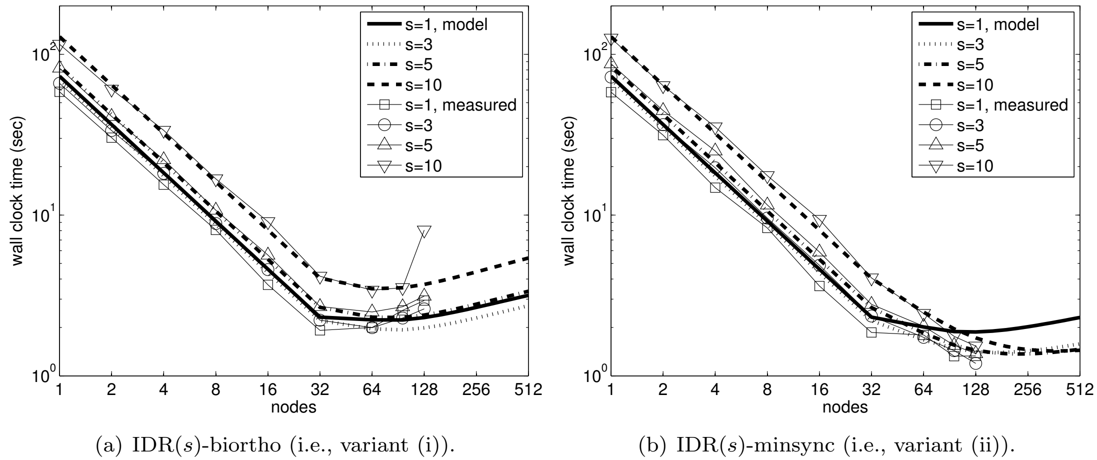
Numerical results

4.6.5 Validation of the parallel performance model

Shown in Figure 4.3 using a log–log scale are the predicted total computing times defined by (4.36) in Section 4.4.1 for variants (i) and (ii) using $s \in \{1, 3, 5, 10\}$ and using up to (in theory) 512 nodes (i.e., sixteen clusters). Experiments that use up to 32 nodes employ the LU site and corresponding parameters in the performance model. In the larger experiments, a cluster consisting of 32 nodes is added each time and the corresponding multi–cluster performance model is used. Also shown are the measured wall clock times using up to 128 nodes (i.e., four clusters of the DAS–3) for $s \in \{1, 3, 5, 10\}$.

Communication overhead on the LU site (i.e., using less than 32 nodes) is relatively small. As a result, the total wall clock time scales almost linearly with the number of nodes in both variants. This holds for both the model and the measurements. Note that Table 4.4 also shows this linear scaling behaviour for $s = 2$. However, when more clusters are added (i.e., using more than 32 nodes), the effect of communication begins to play a larger role. In this case, the optimal s and number of nodes differ for both variants.

According to the measurements from Figure 4.3(a), the optimal value of s for solving this test problem using variant (i) lies between $s = 1$ and $s = 3$. The corresponding number of nodes lies between 32 and 64 nodes (i.e., between one and two clusters). This is in accordance with the predictions from Table 4.5, which showed that the estimated optimal value of s is $s = 2$ using



(a) IDR(s)-biortho (i.e., variant (i)).

(b) IDR(s)-minsync (i.e., variant (ii)).

Figure 4.3: Performance model results for variants (i) and (ii) using 32 nodes per cluster.

one cluster.

The measurements from Figure 4.3(b) show that for variant (ii) using 128 nodes (i.e., four clusters) gives the minimum computing times. The trend seems to be that using more than four clusters only results in a marginal reduction in wall clock time. The corresponding value of s is between $s = 3$ and $s = 5$. Indeed, the predictions from Table 4.5 show that using more than four clusters results in small reductions in total time. Correspondingly, the estimated optimal value of s is $s = 4$.

These results nicely illustrate the fact that there is an optimal value of s and number of nodes for solving this test problem in a minimal amount of time.

4.6.6 Comparing the time per IDR(s) cycle to the performance model

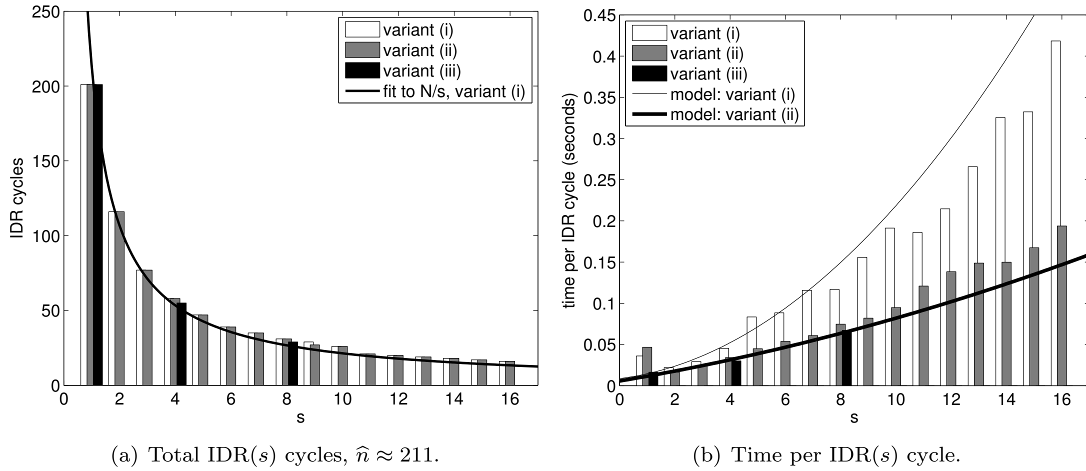
To investigate the relation between the value of s and the time per IDR(s) cycle, the following experiment is performed. Figure 4.4 shows results of experiments using the three unpreconditioned variants and using a total of 64 nodes, divided equally between the four DAS–3 sites that employ the fast interconnect.

When using sparse column vectors for $\widetilde{\mathbf{R}}$ (i.e., variant (iii)), the iteration did not converge for values of $s > 8$ for this particular test case due to numerical stability issues. For this reason and because s has to be a multiple of the number of clusters in the grid for variant (iii), only results for $s = 1$, 4, and 8 are given.

For completeness, the total number of IDR cycles for $s \in \{1, \dots, 16\}$ is shown in Figure 4.4(a), which is practically identical for all three unpreconditioned variants. As before, the total number of IDR cycles is fitted to the curve $\widehat{n}/s$, which gives in this case $\widehat{n} \approx 211$. Naturally, this value is almost identical to the previously obtained value for $\widehat{n}$ from Section 4.6.3.

More interestingly, Figure 4.4(b) shows the wall clock time per IDR cycle for increasing values of s. As mentioned in Sect 4.4.1, the performance model (i.e., expression (4.42)) predicts that for larger bandwidth values, the time spent on inner products per cycle in variant (ii) scales almost linearly with s. The measurements are in agreement with this prediction.

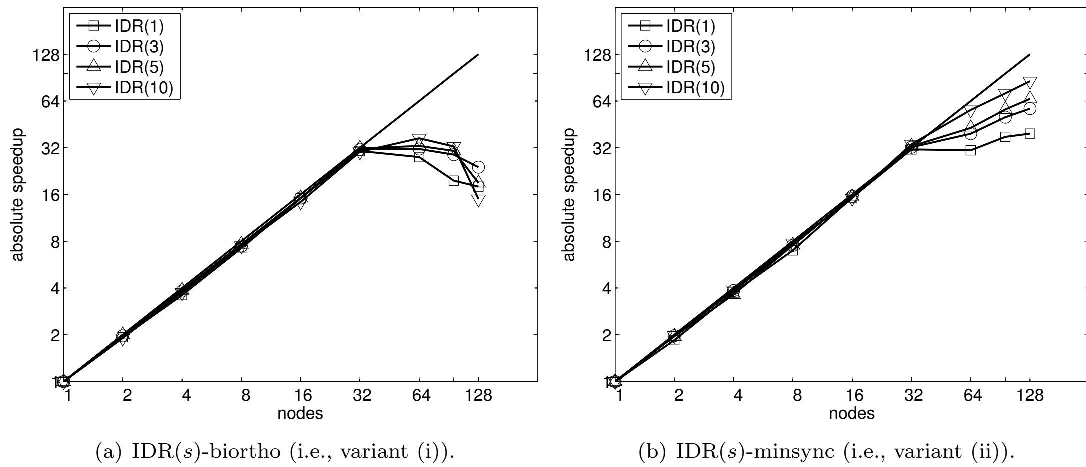
Similarly, expression (4.40) from Sect 4.4.1 shows that the time per IDR(s) cycle in variant (i) has a more quadratic behaviour in s, which is also in agreement with the measurements from



(a) Total IDR(s) cycles, $\widehat{n} \approx 211$.

(b) Time per IDR(s) cycle.

Figure 4.4: Investigating s–dependence for $s \in \{1, \ldots, 16\}$ using 64 nodes, four sites, and fast network.



(a) IDR(s)-biortho (i.e., variant (i)).

(b) IDR(s)-minsync (i.e., variant (ii)).

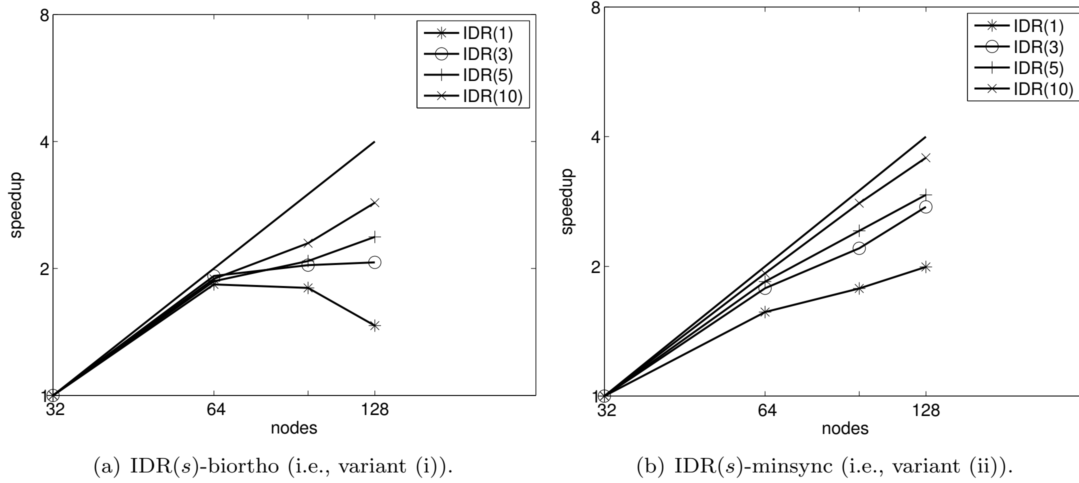
Figure 4.5: Strong scalability results, scaled to number of iterations, $n = 128^3$.

Figure 4.4(b). As a result, there is a significant increase in time per iteration with increasing s for variant (i).

In general, the performance model is in good agreement with the measurements. The outlier for $s = 1$ for variants (i) and (ii) seems related to C compiler optimisations, since *disabling* these optimisations *reduced* the time per cycle. For some reason, variant (iii) conforms well to the performance model for $s = 1$ when using the compiler optimisations.

4.6.7 Parallel speedup results

In this section strong and weak scalability of the three unpreconditioned variants is investigated.



(a) IDR(s)-biortho (i.e., variant (i)). (b) IDR(s)-minsync (i.e., variant (ii)).

Figure 4.6: Strong scalability results, scaled to number of iterations, $n = 256^3$.

Strong scalability

The standard definition for strong scalability S is used, i.e.,

$$S(p) = \frac{T(1)}{T(p)}, \tag{4.52}$$

where $T(1)$ is the execution time of the parallel algorithm on one node and p is the number of nodes. Figure 4.5 shows strong scalability results using variants (i) and (ii) for $s \in \{1, 3, 5, 10\}$, in Figure 4.5(a) and Figure 4.5(b) respectively. The scalability results of variant (iii) is similar to that of variant (ii) and are therefore omitted. Optimal speedup $S(p) = p$ is also shown.

The near to linear speedup of both variants using up to 32 nodes on the LU site is not surprising, considering the fact that in this case communication overhead is almost negligible. However, as more sites are added, the results show that IDR(s)-minsync (i.e., variant (ii)) scales much more favourably than IDR(s)-biortho (i.e., variant (i)). Since for $s = 1$ variants (i) and (ii) almost have the same implementation, speedup is roughly identical. In addition, the results show that variant (i) exhibits the same (bad) scalability for all values of s, while increasing s in variant (ii) gives significant gains in scalability.

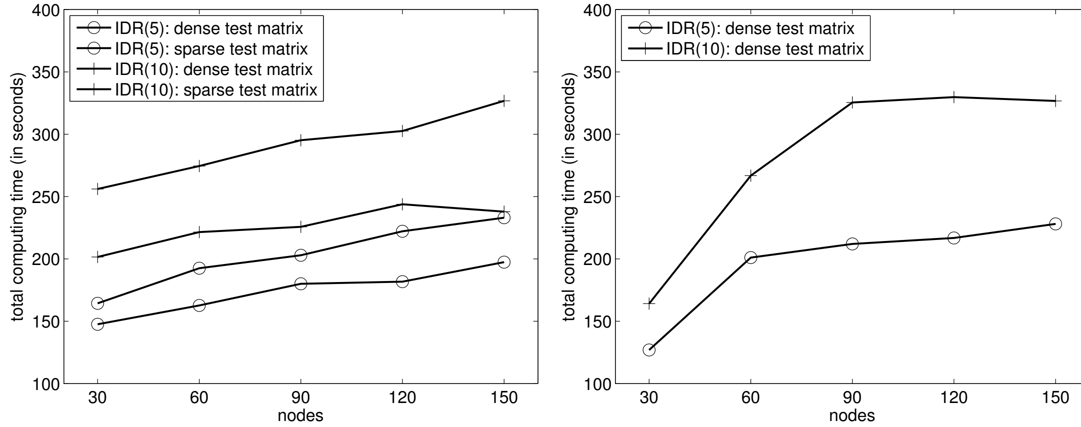
For completeness, Figure 4.6 shows strong scalability results for a bigger test problem, where $n = 256^3 \approx 17,000,000$. In this case, the smallest experiment uses $p = 32$ nodes on one cluster (i.e., $T(1) \equiv T(32)$ in (4.52)). Similar to the results in Figure 4.5, the single synchronisation point in IDR(s)-minsync results in superior parallel speedup compared to IDR(s)-biortho, in particular for higher values of s. Also, the total number of iterations of both variants for this test problem is shown in Table 4.6, which shows that the number of iterations is almost equal.

Weak scalability

To investigate the weak scalability of IDR(s)-minsync, the number of equations per node is fixed to approximately two million and a fixed number of iterations of 275 is performed. The TUD site is also used in this case and the number of nodes in each experiment is $30, 60, \ldots, 150$. The corresponding total problem sizes are listed in Table 4.3. Note that for variant (iii), the value

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

Table 4.6: Total number of iterations.



(a) Comparing sparse and dense test matrices (first strategy).

(b) Using a dense test matrix (second strategy).

Figure 4.7: Weak scaling experiments on the DAS–3.

of s has to be a multiple of the number of clusters in the grid. In the ideal case, the time per iteration is constant for increasing number of nodes.

As explained in Section 4.6.2, the problem size is increased using two different strategies. Figure 4.7 shows the weak scalability results for $s \in \{5, 10\}$. In Figure 4.7(a) results are given when employing the first strategy, showing that using the sparse matrix $\widetilde{\mathbf{R}}$ gives increased gains in execution time for increasing s.

Figure 4.7(b) gives results using the second strategy. Not surprisingly, adding the second cluster results in a large jump in execution time, because the relative increase in communication time is rather high in this case. However, adding subsequent clusters show weak scalability results comparable to Figure 4.7(a).

4.6.8 Results for IDR(s) with asynchronous preconditioning

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 asynchronous 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 very expensive global synchronisation. In order to simulate such an environment, network load will be varied artificially. Similar to the experiments performed in Chapter 3, the experiments here are performed on both a *lightly loaded* global network and on a *heavily loaded* global network.

The results are displayed in Figure 4.8. If a result for a particular experiment is missing, it means that the iteration did not converge.

We can make the following observations:

- In all experiments with preconditioning, after the convergence criterion was satisfied, the required accuracy was indeed achieved. This shows that the asynchronous 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 asynchronous 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 asynchronous preconditioner is robust against changes in network load: little change in performance can be observed between the results with preconditioning for 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 asynchronous 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.

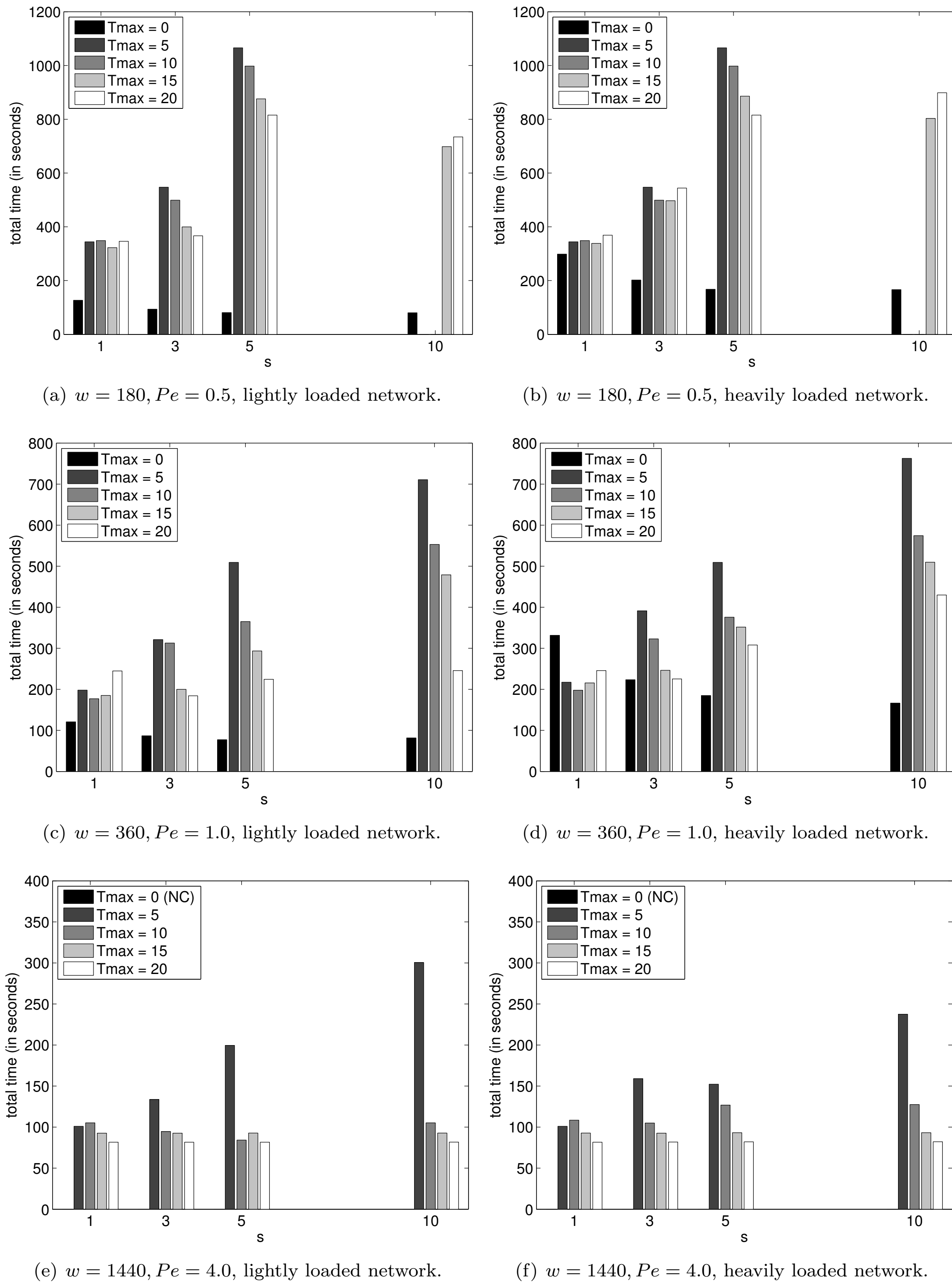
4.7 Conclusions

The recent IDR(s) method is a family of fast algorithms for solving large sparse nonsymmetric linear systems. In cluster computing and in particular in Grid computing, global synchronisation is a critical bottleneck in parallel iterative methods. To alleviate this bottleneck in IDR(s) algorithms, four strategies were used.

Firstly, by reformulating the efficient and numerically stable IDR(s)-biortho method [140], the IDR(s)-minsync method was derived which has a *single* global synchronisation point per iteration step. Experiments on the DAS–3 multi–cluster show that the new IDR(s)-minsync method exhibits increased speedup for increasing values of s. In contrast, the original IDR(s)-biortho variant has no speedup whatsoever on the DAS–3 multi–cluster for our test problem.

In addition, the test matrix in IDR(s)-minsync was chosen in such a way that the work, communication, and storage involving this matrix is minimised on the DAS–3 multi–cluster. Experiments on the DAS–3 show that this approach results in reduced execution times, albeit relatively moderate. However, the experiments also showed that this technique of using a sparse test matrix in IDR(s) methods can result in numerical instabilities, in particular for larger values of s. Given the fact that the reduction in execution times was limited, we do not recommend using sparse test vectors in the context of IDR(s) methods and multi–clusters.

Also, the presented parallel performance models were utilised for *a priori* estimation of the optimal value of s and corresponding number of nodes. This approach can be used to minimise the total execution time of parallel IDR(s) methods on both a single cluster and on a multi–cluster. The estimates were in good agreement with the experimental results. It is interesting to see that in (parallel) IDR(s) algorithms the optimal value of s can be determined in such a



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

manner. Note that it is possible to apply the techniques for estimating the optimal s when a *fixed* preconditioner such as a block incomplete LU factorisation is used. This can be accomplished by adding the communication and computational cost of the preconditioner to the performance models.

Finally, we have discussed the combination of IDR(s) with an asynchronous preconditioner. Experiments on the DAS–3 multi–cluster for a large 3D convection–diffusion problem show that this combination is particularly effective for high Péclet numbers. Moreover, the asynchronous 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). This also means that the technique for computing the optimal s cannot be used in this case, since it depends on the finite convergence behaviour of 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 asynchronous preconditioning step should be performed is at this moment still an open question.

Chapter 5

IDR(s) as a Deflation Method

This chapter has been submitted as:

T. P. Collignon, G. L. G. Sleijpen, and M. B. van Gijzen. Interpreting IDR(s) as a deflation method. *Journal of Computational and Applied Mathematics*, 2011. Special Issue: Proceedings ICCAM–2010 (submitted).

Overview

In this chapter the IDR(s) method is interpreted in the context of deflation methods. It is shown that IDR(s) can be seen as a Richardson iteration preconditioned by a variable deflation–type preconditioner. The main result of this chapter is the *IDR projection theorem*, which relates the spectrum of the deflated system in each IDR(s) cycle to all previous cycles. The theorem shows that this so–called *active spectrum* becomes increasingly more *clustered*. This clustering property may serve as an intuitive explanation for the excellent convergence properties of IDR(s). These remarkable spectral properties exist whilst using a deflation subspace matrix of fixed rank. Variants of explicitly deflated IDR(s) are compared to IDR(s) in which the IDR deflation subspace matrix is augmented with "traditional" deflation vectors. The theoretical results are illustrated by numerical experiments.

5.1 Introduction

In the previous chapter the IDR(s) method was investigated extensively in a parallel context. This included using a parallel asynchronous iterative method as a preconditioner, which means that IDR(s) was treated as a flexible method. In order to gain more understanding into the convergence behaviour of IDR(s), we interpret the IDR(s) method as a deflation method in this chapter. In doing so, it is hoped that we can analyse the effect of a varying preconditioner on the convergence properties of IDR(s).

We again consider nonsymmetric linear systems:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \mathbf{A} \in \mathbb{C}^{n\times n}, \quad \mathbf{x}, \mathbf{b} \in \mathbb{C}^{n}. \tag{5.1}$$

It is shown in Section 5.2 that the IDR(s) method can be viewed as a so–called *adapted deflation method* [127, Section 2.3.3]. The method can be seen as a Richardson iteration preconditioned by a variable deflation–type preconditioner, where the preconditioner is updated in each cycle with new spectral information. This interpretation leads to more insight into the structure of IDR(s) methods. In particular, it leads to the *IDR projection theorem* given in Section 5.3 (Theorem 5.17), which relates the spectrum of the deflated system of a particular cycle to the spectra of the deflated systems of all the previous cycles.

An IDR(s) cycle consists of $s+1$ (preconditioned) matrix–vector multiplications and in the kth cycle a new smoothing parameter ω_k can be chosen. These parameters play the same role as the ω's in the Bi-CGSTAB method: to obtain smoother convergence behaviour. Note that for Bi-CGSTAB we have $s = 1$ and for this method the ω is therefore updated every other matrix–vector multiplication step.

Assume that $\omega_k \neq 0$ and put $\mu_k \equiv \omega_k^{-1}$. The IDR projection theorem states that in the kth cycle, s eigenvalues of $\mathbf{A}$ are shifted to μ_k. Moreover, each $\nu \in \{\mu_0, \mu_1, \ldots, \mu_{k-1}\}$ is an eigenvalue of the deflated system in cycle k each with geometric multiplicity s. This implies that the spectra of the deflated systems becomes increasingly *clustered* for increasing k. Quite remarkably, this is accomplished using a deflation subspace matrix of fixed rank equal to s. The deflation subspace built in cycle k retains spectral information from all previous IDR(s) cycles. It is argued that the effectiveness of IDR(s) methods comes from the clustering of the spectra of the deflated systems. Possible consequences of this interpretation are discussed in this chapter. With the exception of certain extreme cases, these results also suggest that the value of μ_k itself plays a relatively small role in the convergence process, especially for larger s.

It can be shown that in the generic case, IDR(s) methods compute the exact solution in exact arithmetic within n/s cycles [120, Section 3]. In this case the spectrum of the deflated system in the final cycle solely consists of n/s eigenvalues each with geometric multiplicity s.

Deflation for iterative methods has been investigated by many authors, see for example [62, 67, 94, 95, 126]. A typical deflation procedure consists of three steps: identifying some particular subspace that hampers convergence, finding a suitable approximation to this space, and removing the influence of this space on the iteration process. Usually, this subspace is the eigenspace corresponding to eigenvalues of $\mathbf{A}$ that are in some sense "undesirable".

In Section 5.4 two variants of explicitly deflated IDR(s) methods are compared to IDR(s) in which the IDR deflation matrices are augmented with traditional deflation vectors. Spectral comparisons between the deflated systems of these three approaches are performed.

The IDR(s) method adaptively constructs a deflation–type preconditioner. This is not a new concept: for example, methods such as described in [32, 61, 7] *explicitly* construct deflation vectors based on spectral information gathered by the Arnoldi process during iterations of restarted GMRES(m). The goal there is to approximate invariant subspaces associated with a *specific* set of eigenvalues (e.g., eigenvalues that are small in magnitude). These methods differ in how the approximate invariant subspaces are constructed and how they are incorporated in the iteration process. In general, the dimension of the invariant subspace grows during the iterative process. In order to limit memory cost, the dimension of the invariant subspace has to be fixed, which can reduce the effectiveness of the preconditioner.

In contrast, IDR(s) constructs the deflation preconditioner *implicitly*. Furthermore, the deflation subspace in IDR(s) is unrelated to any specific spectral components of $\mathbf{A}$. Nevertheless, new spectral information seems to be continuously injected in the iteration process, all the while keeping the dimension of the deflation subspace fixed. In this sense IDR(s) can be seen as an efficient deflation–type method.

This chapter shows that interpreting IDR(s) as a deflation method has resulted in new insights into the structure of IDR(s) methods and Section 5.5 lists the main conclusions.

Preliminaries

The following notational conventions, terminology, and definitions will be used in this chapter.

If $\mathcal{V}$ is a linear subspace of $\mathbb{C}^n$, then an n-vector $\mathbf{v}$ is *orthogonal* to $\mathcal{V}$, $\mathbf{v} \perp \mathcal{V}$, if $\mathbf{v}$ is orthogonal to all $\mathbf{w} \in \mathcal{V}$. The space of all n-vectors $\mathbf{v}$ that are orthogonal to $\mathcal{V}$ is denoted by $\mathcal{V}^\perp$.

If $\widetilde{\mathbf{R}}, \mathbf{V}_1, \ldots, \mathbf{V}_k$ are matrices with column vectors of size n, then $\text{span}(\mathbf{V}_1, \ldots, \mathbf{V}_k)$ is the subspace of $\mathbb{C}^n$ spanned by all columns of all $\mathbf{V}_j$. We put $\mathbf{V}_1, \ldots, \mathbf{V}_k \perp \widetilde{\mathbf{R}}$ if $\text{span}(\mathbf{V}_1, \ldots, \mathbf{V}_k) \subset \text{span}(\widetilde{\mathbf{R}})^\perp$. Then we say that the $\mathbf{V}_1, \ldots, \mathbf{V}_k$ are *orthogonal to* $\widetilde{\mathbf{R}}$. $[\mathbf{V}_1, \ldots, \mathbf{V}_k]$ is the matrix with column vectors the columns of the matrices $\mathbf{V}_j$. We identify n-vectors $\mathbf{v}$ and $n \times 1$ matrices $[\mathbf{v}]$. We call the columns of $\mathbf{V}$ a basis of $\text{span}(\mathbf{V}) \subset \mathbb{C}^n$.

An MV is a matrix-vector multiplication $\mathbf{A}\mathbf{v}$, where $\mathbf{v}$ is an n-vector. The multiplication $\mathbf{A}\mathbf{V}$ with $\mathbf{V} \in \mathbb{C}^{n \times s}$ requires s MVs. The index k refers to the kth IDR(s) cycle. When not specified otherwise, we assume that the coefficient matrix $\mathbf{A}$ is nonsingular.

For a reminder of other notational conventions used in this chapter, see Table 1.7 on page 29.

5.2 Relation between IDR and deflation

In Section 5.2.1 IDR methods are discussed, while Section 5.2.2 presents the structure of adapted deflation methods. Section 5.2.3 unifies these two concepts by showing how the IDR method can be interpreted as a deflation method. Lastly, Section 5.2.4 contains some remarks related to IDR *algorithms*.

In the following, let $\widetilde{\mathbf{R}}$ be an $n \times s$ matrix; $\widetilde{\mathbf{R}}$ is the s-dimensional *initial shadow residual* or *IDR test matrix*. It is assumed that the value of s is fixed during the iteration process.

5.2.1 IDR methods

Induced dimension reduction (IDR) methods iteratively construct residuals in a sequence $(\mathcal{G}_k)$ of shrinking subspaces: in each cycle k, we start from a residual in $\mathcal{G}_k$ and construct a residual in $\mathcal{G}_{k+1}$. Ultimately, the residual is forced in the zero–dimensional subspace $\mathcal{G}_k = \{\mathbf{0}\}$ for some $k \leq n$. These IDR subspaces are recursively defined as follows:

Definition 5.1. Let $\mathcal{G}_0$ be a linear subspace of $\mathbb{C}^n$ such that $\mathbf{A}\mathcal{G}_0 \subset \mathcal{G}_0$ (for example, the *full* Krylov subspace $\mathcal{K}(\mathbf{A}, \mathbf{v}) \equiv \text{span}\{\mathbf{A}^k\mathbf{v} \mid k = 0, 1, \ldots\}$ for some $\mathbf{v} \in \mathbb{C}^n$). For a sequence (μ_k) in $\mathbb{C}$, let the sequence $(\mathcal{G}_k)$ of *IDR subspaces* be defined by

$$\mathcal{G}_k \equiv (\mathbf{A} - \mu_k\mathbf{I})(\mathcal{G}'_{k-1}), \quad \text{where} \quad \mathcal{G}'_{k-1} \equiv \mathcal{G}_{k-1} \cap \widetilde{\mathbf{R}}^{\perp} \quad (k \in \mathbb{N}). \tag{5.2}$$

The following result states that the sequence $(\mathcal{G}_k)$ of IDR subspaces forms a strict chain of nested linear subspaces. For a proof, see [115, 120].

Theorem 5.2 (IDR)**.** *With $\mathcal{G}_k$ as in Definition 5.1, we have for $k, \ell \in \mathbb{N}_0$*

$$\mathcal{G}_k \subset \mathcal{G}_\ell \quad (\ell \leq k).$$

If $\mathbf{A}$ has no eigenvector in $\widetilde{\mathbf{R}}^{\perp}$, then $\mathcal{G}_k = \mathcal{G}_\ell$ ($\ell < k$) if and only if $\mathcal{G}_k = \{\mathbf{0}\}$. □

The IDR subspaces $\mathcal{G}_k$ are a special case of a wider class of spaces called *Sonneveld subspaces* defined below.

Definition 5.3 (cf. [117])**.** Let $\mathcal{G}_0$ be as in Definition 5.1. For a polynomial P of exact degree k, the *Sonneveld subspace* $\mathcal{S}(P, \mathbf{A}, \widetilde{\mathbf{R}})$ is defined by

$$\mathcal{S}(P, \mathbf{A}, \widetilde{\mathbf{R}}) \equiv \{P(\mathbf{A})\mathbf{v} \mid \mathbf{v} \in \mathcal{G}_0, \mathbf{v} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})\}, \tag{5.3}$$

where

$$\mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}}) \equiv \left\{ \sum_{j=0}^{k-1} (\mathbf{A}^*)^j \widetilde{\mathbf{R}}\gamma_j \mid \gamma_j \in \mathbb{C}^s \right\} \tag{5.4}$$

is the (block) *Krylov subspace* of order k (generated by $\mathbf{A}^*$ and $\widetilde{\mathbf{R}}$). Note that the dimension of the Sonneveld subspace $\mathcal{S}(P, \mathbf{A}, \widetilde{\mathbf{R}})$ is in general (*generic case*) equal to $n - ks$. Also, note that

$$\mathbf{v} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}}) \qquad \Leftrightarrow \qquad \widetilde{\mathbf{R}}^*\mathbf{A}^j\mathbf{v} = 0 \quad \text{for all} \quad j < k. \tag{5.5}$$

The following result explicitly relates the Sonneveld subspaces to the IDR subspaces. For a proof and additional interesting results on Sonneveld subspaces, see [115].

Theorem 5.4. *Let $P_k(\lambda) \equiv \prod_{j=1}^{k}(\lambda - \mu_j)$. With $\mathcal{G}_k$ as in Definition 5.1, we have*

$$\mathcal{G}_k = \mathcal{S}(P_k, \mathbf{A}, \widetilde{\mathbf{R}}). \tag{5.6}$$

The residual that arrives first in an IDR subspace $\mathcal{G}_k$ is called a *primary* residual. This terminology is consistent with the existing literature on IDR(s) methods. In addition, we introduce here the term *secondary* residual, which is also always formed and which lives in the subspace $\mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}$. For observations on the uniqueness of these residuals, see [116, 120]. We call any other residuals that may be constructed during a cycle *auxiliary* residuals. For example, this includes the intermediate residuals generated in Algorithm 4.1 and Algorithm 4.2.

Continuing the discussion of Section 4.2 on page 71, iterative algorithms based on the IDR theorem (i.e., Theorem 5.2) essentially consist of three key steps, which constitute the kth cycle of an IDR method (i.e., $s+1$ MVs). Let $k \in \mathbb{N}_0$. Given a full rank $n \times s$ matrix $\mathbf{V}_k$ with columns in $\mathcal{G}_k$ and a primary residual $\mathbf{r}_k \in \mathcal{G}_k$, we have:

(i) *The projection step*: the secondary residual $\mathbf{r}'_k$ is formed in $\mathcal{G}'_k \equiv \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$ using an oblique projection involving $\mathbf{V}_k$. This step consists of s MVs.

(ii) *The dimension reduction step*: given the secondary residual $\mathbf{r}'_k \in \mathcal{G}'_k$ and after selecting a scalar $\omega_{k+1} \equiv 1/\mu_{k+1}$, the next primary residual $\mathbf{r}_{k+1}$ in the lower dimensional subspace $\mathcal{G}_{k+1} \subset \mathcal{G}_k$ is computed as $\mathbf{r}_{k+1} = (\mathbf{I} - \omega_{k+1}\mathbf{A})\mathbf{r}'_k$. This step consists of 1 MV.

(iii) *The search matrix step*: a full rank $n \times s$ matrix $\mathbf{V}_{k+1}$ with columns in $\mathcal{G}_{k+1}$ of the form $\mathbf{AU}_{k+1}$ with the so–called *search matrix* $\mathbf{U}_{k+1}$ explicitly available is constructed. This matrix is used for the *next* projection step.

To summarise, one full cycle of IDR(s) consists of $s+1$ MVs:

$$\begin{array}{ccccc} \text{primary residual} & \xrightarrow{s \text{ MVs}} & \text{secondary residual} & \xrightarrow{1 \text{ MV}} & \text{next primary residual} \\ \mathbf{r}_k \in \mathcal{G}_k & & \mathbf{r}'_k \in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp & & \mathbf{r}_{k+1} \in \mathcal{G}_{k+1} \end{array} . \quad (5.7)$$

Following the discussion of [120, Section 4], three fundamental choices can be distinguished when deriving practical IDR(s) algorithms. These choices are directly related to the three steps given above. Not only do these choices influence the numerical stability and efficiency of the resulting IDR(s) algorithm, they can also drastically affect parallel scalability. These choices are:

(i) *Choosing the IDR test matrix.* In IDR(s) algorithms the computation of most of the inner products pertains to the columns of $\widetilde{\mathbf{R}}$. To reduce the amount of computational work (and communication) involving the IDR test matrix, sparse column vectors for $\widetilde{\mathbf{R}}$ may be used. A possible disadvantage is that such an approach can have an adverse effect on robustness. For a study on using sparse column vectors for $\widetilde{\mathbf{R}}$ and other methods of minimising communication in parallel IDR(s) algorithms, see Chapter 4 of this thesis.

(ii) *Selecting ω_{k+1} in the dimension reduction step.* In the dimension reduction step the value of ω_{k+1} can be chosen freely. Similar to other short–recurrence methods such as Bi–CGSTAB, smoother convergence may be achieved by choosing ω_{k+1} in such a way that the next residual is minimised in norm. For certain problems the (standard) linear minimal residual step causes break–down of the iteration process and sophisticated repair techniques such as used in Bi–CGSTAB(ℓ) are required [114, 113]. In Bi–CGSTAB(ℓ) stabilisation polynomials of degree ℓ are used and recently this technique has been combined with IDR(s) [117].

(iii) *Constructing vectors for the space $\mathcal{G}_{k+1}$ in the search matrix step.* Vectors for $\mathcal{G}_{k+1}$ can be generated by a number of ways. For example, using GCR–type methods [139]. Also, vectors for $\mathcal{G}_{k+1}$ can be made to satisfy (one–sided) bi–orthogonality relations with the columns of $\widetilde{\mathbf{R}}$ [140]. Furthermore, one can either derive variants that *only* compute primary and secondary residuals (e.g., [117]) or variants that also construct auxiliary residuals (e.g., [139, 140]). For more details on this type of freedom, see [116].

Note that different strategies for the last choice result in mathematically equivalent IDR(s) methods, while different strategies for the first and second choice result in fundamentally different iterative processes.

5.2.2 Deflation methods

In this section the structure of a particular deflation–type method called *adapted* deflation is presented. The discussion follows that of [127, Section 2]. We will first look at general deflation methods.

General deflation

We start with some terminology (cf. Definition 5.5) and a related result (cf. Lemma 5.6).

Definition 5.5 (cf. [62, 94, 116, 127]). Let $\mathbf{A} \in \mathbb{C}^{n\times n}$ be nonsingular and let $\mathbf{U}, \widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$ be suitable *deflation subspace matrices* of full rank. Then the oblique projections (or *deflation matrices*) $\Pi, \widehat{\Pi} \in \mathbb{C}^{n\times n}$, the *correction matrix* (or *coarse grid correction*) $\mathbf{Q} \in \mathbb{C}^{n\times n}$, and the invertible (*Galerkin*) matrix $\mathbf{E} \in \mathbb{C}^{s\times s}$ are defined as

$$\Pi \equiv \mathbf{I} - \mathbf{AQ} \quad \text{and} \quad \widehat{\Pi} \equiv \mathbf{I} - \mathbf{QA}, \quad \text{where} \quad \mathbf{Q} \equiv \mathbf{U}\mathbf{E}^{-1}\widetilde{\mathbf{R}}^* \quad \text{and} \quad \mathbf{E} \equiv \widetilde{\mathbf{R}}^*\mathbf{AU}. \tag{5.8}$$

In the next section we will sometimes relate an operator or matrix with a specific IDR cycle k by adding a subscript k, e.g., $\mathbf{U}_k$ and $\mathbf{Q}_k$. Analogously, if a projection belongs to an IDR cycle k, we write e.g., Π_k.

Lemma 5.6 (cf. [62, 127]). *Note that the following properties hold for arbitrary full–rank matrices* $\mathbf{A}, \mathbf{U}$*, and* $\widetilde{\mathbf{R}}$*. In particular, we do not assume that the columns of* $\mathbf{U}$ *or* $\widetilde{\mathbf{R}}$ *are (approximations of) eigenvectors. Let* $j, k \in \mathbb{N}_0$*. Then*

(i) $\Pi^2 = \Pi, \widehat{\Pi}^2 = \widehat{\Pi}, (\mathbf{AQ})^2 = \mathbf{AQ}, (\mathbf{QA})^2 = \mathbf{QA}$

(ii) $\Pi\mathbf{A} = \mathbf{A}\widehat{\Pi} = \Pi\mathbf{A}\widehat{\Pi}$

(iii) $\widehat{\Pi}\mathbf{U} = \Pi\mathbf{AU} = \mathbf{0}_{n,s}$

(iv) $\widetilde{\mathbf{R}}^*\Pi = \widetilde{\mathbf{R}}^*\mathbf{A}\widehat{\Pi} = \mathbf{0}_{s,n}$

(v) $\mathbf{AQ} = \mathbf{I} - \Pi, \mathbf{QA} = \mathbf{I} - \widehat{\Pi}$

(vi) $\mathbf{QAU} = \mathbf{U}, \mathbf{QAQ} = \mathbf{Q}$

(vii) $\Pi\mathbf{Q} = \mathbf{Q}\Pi = \Pi\mathbf{AQ} = \mathbf{AQ}\Pi = \mathbf{QA}\widehat{\Pi} = \mathbf{Q}\Pi\mathbf{A} = \mathbf{0}_{n,n}$

(viii) $(\mathbf{I} - \widehat{\Pi})\mathbf{x} = \mathbf{Qb}$

(ix) $\widetilde{\mathbf{R}}^*\mathbf{AQ} = \widetilde{\mathbf{R}}^*$

Proof. We only show property (iii). The other properties can be derived similarly. For (iii), note that $\Pi\mathbf{AU} = \mathbf{AU} - \mathbf{AU}(\widetilde{\mathbf{R}}^*\mathbf{AU})^{-1}\widetilde{\mathbf{R}}^*\mathbf{AU} = \mathbf{AU} - \mathbf{AU} = \mathbf{0}_{n,s}$. □

Note 5.7. The following observations can be made.

- The operator Π is an oblique projection along $\text{span}(\mathbf{AU})$ onto the orthogonal complement of $\widetilde{\mathbf{R}}$ (or onto $\text{span}(\widetilde{\mathbf{R}})^\perp = \mathcal{N}(\widetilde{\mathbf{R}}^*)$). The operator $\mathbf{AQ}$ is an oblique projection along the orthogonal complement of $\widetilde{\mathbf{R}}$ onto $\text{span}(\mathbf{AU})$.

- The operator $\widehat{\Pi}$ is an oblique projection along $\text{span}(\mathbf{U})$ onto $\text{span}(\mathbf{A}^*\widetilde{\mathbf{R}})^\perp = \mathcal{N}(\widetilde{\mathbf{R}}^*\mathbf{A})$. The operator $\mathbf{QA}$ is an oblique projection along $\text{span}(\mathbf{A}^*\widetilde{\mathbf{R}})^\perp$ onto $\text{span}(\mathbf{U})$.

- $\Pi\mathbf{v} \perp \widetilde{\mathbf{R}}$ for all $\mathbf{v}$, $\mathbf{A}\widehat{\Pi}\mathbf{v} \perp \widetilde{\mathbf{R}}$ for all $\mathbf{v}$, $\Pi\mathbf{v} \in \mathcal{N}(\widetilde{\mathbf{R}}^*)$ for all $\mathbf{v}$, $\mathbf{U} \in \mathcal{N}(\Pi\mathbf{A})$, $\mathbf{U} \in \mathcal{N}(\widehat{\Pi})$, $\Pi\mathbf{A} \in \mathcal{N}(\widetilde{\mathbf{R}}^*)$.

- Both projections Π and $\widehat{\Pi}$ have s zero and $n - s$ unit eigenvalues, since $\Pi\mathbf{AU} = 0_{n,s}$ and $\Pi^2\mathbf{Y} = \Pi\mathbf{Y}$ for full rank $\mathbf{Y} \in \mathbb{R}^{n\times(n-s)}$ satisfying $\text{span}(\mathbf{Y}) = \text{span}(\widetilde{\mathbf{R}})^\perp$. Vice versa, both the projections $\mathbf{AQ}$ and $\mathbf{QA}$ have $n - s$ zero eigenvalues and s unit eigenvalues.

- Note that $\dim \operatorname{span}(\Pi) = \dim \mathcal{N}(\mathbf{AQ}) = \dim \operatorname{span}(\widetilde{\mathbf{R}})^{\perp} = \dim \mathcal{N}(\widetilde{\mathbf{R}}^*) = n - s$.
- Note that $\dim \mathcal{N}(\Pi) = \dim \operatorname{span}(\mathbf{AQ}) = \dim \operatorname{span}(\mathbf{AU}) = s$.
- Note that $\dim \operatorname{span}(\Pi) + \dim \mathcal{N}(\Pi) = (n - s) + s = n = \operatorname{rank} \Pi$.

Adapted deflation methods

In an adapted deflation method [127, Section 2.3.3], two different preconditioners $\mathbf{C}_1, \mathbf{C}_2 \in \mathbb{C}^{n \times n}$ are combined as follows. Let $k \in \mathbb{N}_0$. Given $\mathbf{x}_0$, consider the two–step stationary iterative method (cf. predictor/corrector–type method, see also Section 3.8 on page 62)

$$\begin{cases} \mathbf{x}'_k &= \mathbf{x}_k + \mathbf{C}_1(\mathbf{b} - \mathbf{A}\mathbf{x}_k); \\ \mathbf{x}_{k+1} &= \mathbf{x}'_k + \mathbf{C}_2(\mathbf{b} - \mathbf{A}\mathbf{x}'_k), \end{cases} \tag{5.9}$$

which can be combined to obtain

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{P}(\mathbf{b} - \mathbf{A}\mathbf{x}_k), \tag{5.10}$$

where

$$\mathbf{P} = \mathbf{C}_1 + \mathbf{C}_2 - \mathbf{C}_2\mathbf{A}\mathbf{C}_1. \tag{5.11}$$

Using Definition 5.5, let $\mathbf{C}_1 \equiv \mathbf{Q}$ and $\mathbf{C}_2 \equiv \mathbf{B}^{-1}$ where $\mathbf{B}$ is an arbitrary (but fixed) matrix. Then $\mathbf{P} = \mathbf{B}^{-1}\Pi + \mathbf{Q}$ and the deflated linear system can be written as

$$\mathbf{PAx} = \mathbf{Pb}, \tag{5.12}$$

which can be solved using an iterative method. This particular deflation variant is called the *adapted deflation variant 1* (A–DEF1) and is described in [127, Section 2.3.3]. A common approach is to take $\mathbf{U}$ equal to $\widetilde{\mathbf{R}}$, consisting of s eigenvectors belonging to the smallest eigenvalues in norm. Additionally, $\mathbf{B}^{-1}$ is a traditional preconditioner such as an Incomplete LU factorisation. As a result, these s eigenvalues of $\mathbf{A}$ are then shifted to one in $\mathbf{PA}$, removing their influence from the iteration process (assuming that an appropriate scaling has taken place).

Note that premultiplying (5.9) with $-\mathbf{A}$ and adding $\mathbf{b}$ gives the corresponding recursions for the residuals

$$\begin{cases} \mathbf{r}'_k &= (\mathbf{I} - \mathbf{A}\mathbf{C}_1)\mathbf{r}_k; \\ \mathbf{r}_{k+1} &= (\mathbf{I} - \mathbf{A}\mathbf{C}_2)\mathbf{r}'_k. \end{cases} \tag{5.13}$$

In the following, we mainly focus on the recursions for the residual. The corresponding recursions for the iterate can normally speaking be constructed easily.

5.2.3 Interpreting IDR(s) as a deflation method

In this section, the three steps of an IDR(s) method given in Section 5.2.1 will be discussed in the context of the adapted deflation variant given in Section 5.2.2. For reference purposes, Algorithm 5.1 lists a complete IDR(s) algorithm for solving a system $\mathbf{Ax} = \mathbf{b}$ by generating primary and secondary residuals with corresponding iterates, illustrating the three distinct phases of IDR(s). In Algorithm 5.1, a sequence of subspaces of $\mathcal{G}_k$ is generated, i.e., $\operatorname{span}(\mathbf{r}_k, \mathbf{V}_k) \subset \mathcal{G}_k$ for $k \in \mathbb{N}_0$ (see also Proposition 5.13). Note that this variant is purely intended for illustrative purposes and that it should not be used to solve linear systems in practice. Also, this variant does not produce auxiliary residuals. In the discussion below we sometimes specifically refer to line numbers in Algorithm 5.1. The projection step and the dimension reduction step are discussed together. After that, the search matrix step is discussed separately.

Algorithm 5.1 IDR(s) as a deflation method: generating a sequence of subspaces of $\mathcal{G}_k$.

INPUT: $\mathbf{A} \in \mathbb{C}^{n\times n}; \mathbf{x}_0, \mathbf{b} \in \mathbb{C}^n; \widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$; preconditioner $\mathbf{B} \in \mathbb{C}^{n\times n}$; tolerance tol
OUTPUT: Approximate solution $\mathbf{x}$ such that $||\mathbf{b} - \mathbf{A}\mathbf{x}|| \leq$ tol

1: // *Initiation*
2: Compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
3: Select an $n \times s$ matrix $\mathbf{V}_0 \equiv \mathbf{A}\mathbf{U}_0$ with $\mathbf{U}_0$ explicitly available such that $[\mathbf{r}_0, \mathbf{V}_0]$ spans the Krylov subspace $\mathcal{K}_{s+1}(\mathbf{A}, \mathbf{r}_0)$
4: // *Loop over nested $\mathcal{G}_k$ spaces*
5: **for** $k = 1, 2, \ldots$ **do**
6: // ***(i) Projection step:***
7: // *Generate unique secondary residual* $\mathbf{r}'_{k-1} \in \mathcal{G}'_{k-1}$ *with corresponding iterate* $\mathbf{x}'_{k-1}$
8: **let** $\Pi_{k-1} \equiv \mathbf{I} - \mathbf{A}\mathbf{Q}_{k-1}$ and $\widehat{\Pi}_{k-1} \equiv \mathbf{I} - \mathbf{Q}_{k-1}\mathbf{A}$ where $\mathbf{Q}_{k-1} \equiv \mathbf{U}_{k-1}(\widetilde{\mathbf{R}}^*\mathbf{V}_{k-1})^{-1}\widetilde{\mathbf{R}}^*$
9: $\mathbf{x}'_{k-1} = \mathbf{x}_{k-1} + \mathbf{Q}_{k-1}\mathbf{r}_{k-1}$
10: $\mathbf{r}'_{k-1} = \Pi_{k-1}\mathbf{r}_{k-1}$
11: // ***(ii) Search matrix step:***
12: // *generate a basis* $\mathbf{U}'_{k-1}$ *of* $\mathcal{K}_s(\widehat{\Pi}_{k-1}\mathbf{B}^{-1}\mathbf{A}, \widehat{\Pi}_{k-1}\mathbf{B}^{-1}\mathbf{r}'_{k-1})$
13: // *generate a basis* $\mathbf{V}'_{k-1}$ *of* $\mathcal{K}_s(\Pi_{k-1}\mathbf{A}\mathbf{B}^{-1}, \Pi_{k-1}\mathbf{A}\mathbf{B}^{-1}\mathbf{r}'_{k-1})$
14: // *for example, a power basis:*
15: **let** $\mathbf{v}'_0 \equiv \mathbf{r}'_{k-1}$
16: **for** $i = 1$ to s **do**
17: $\mathbf{u}'_i = \widehat{\Pi}_{k-1}\mathbf{B}^{-1}\mathbf{v}'_{i-1}$
18: $\mathbf{v}'_i = \mathbf{A}\mathbf{u}'_i = \Pi_{k-1}\mathbf{A}\mathbf{B}^{-1}\mathbf{v}'_{i-1} \in \mathcal{G}'_{k-1} \equiv \mathcal{G}_{k-1} \cap \widetilde{\mathbf{R}}^\perp$
19: **end for**
20: **let** $\mathbf{U}'_{k-1} \equiv [\mathbf{u}'_1, \ldots, \mathbf{u}'_s]$ and $\mathbf{V}'_{k-1} \equiv [\mathbf{v}'_1, \ldots, \mathbf{v}'_s]$
21: // *Entering* $\mathcal{G}_k$
22: // ***(iii) Dimension reduction step:***
23: Select a scalar ω_k, e.g., $\omega_k = \arg\min_\omega ||(\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'_{k-1}||$
24: // *Compute search matrix* $\mathbf{U}_k$ *for next IDR projection step with* $\mathbf{V}_k \in \mathcal{G}_k$
25: $\mathbf{U}_k = \mathbf{U}'_{k-1} - \omega_k\mathbf{B}^{-1}\mathbf{V}'_{k-1}$
26: $\mathbf{V}_k = (\mathbf{I} - \omega_k\mathbf{A}\mathbf{B}^{-1})\mathbf{V}'_{k-1}$
27: // *Compute next primary residual* $\mathbf{r}_k \in \mathcal{G}_k$ *and corresponding iterate* $\mathbf{x}_k$
28: $\mathbf{x}_k = \mathbf{x}'_{k-1} + \omega_k\mathbf{B}^{-1}\mathbf{r}'_{k-1}$
29: $\mathbf{r}_k = (\mathbf{I} - \omega_k\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'_{k-1}$
30: **if** $||\mathbf{r}_k|| \leq$ tol **then** break **end if**
31: **end for**

Projection step and dimension reduction step

Proposition 5.8 below shows that the IDR(s) method can be seen as a combination of two very specific operators $\mathbf{C}_1$ and $\mathbf{C}_2$. It explains how to move from a primary residual $\mathbf{r}_k$ to the next primary residual $\mathbf{r}_{k+1}$.

To form the secondary residual $\mathbf{r}'_k$ in $\mathcal{G}'_k \equiv \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$ from the primary residual in $\mathcal{G}_k$, we need a full rank $n \times s$ matrix with columns also in $\mathcal{G}_k$. In order to be able to update the approximate solution associated with $\mathbf{r}_k$ this $n \times s$ matrix has to be of the form $\mathbf{A}\mathbf{U}_k$ with $\mathbf{U}_k$ explicitly available: the update for the approximate solution is a linear combination of the columns of $\mathbf{U}_k$. This is also expressed in Proposition 5.8 (cf. (5.15) and line 9 of Algorithm 5.1). We call an $n \times s$ matrix $\mathbf{U}_k$ a *search matrix* if it has full rank and $\text{span}(\mathbf{A}\mathbf{U}_k) \subset \mathcal{G}_k$.

Proposition 5.8. *Consider a* $k \in \mathbb{N}_0$*. Assume* $\mathbf{r}_k \in \mathcal{G}_k$ *with* $\mathbf{r}_k$ *a residual,* $\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$*, with approximate solution* $\mathbf{x}_k$*, where* $\mathbf{x}_k$ *is explicitly available. Also, assume* $\mathrm{span}(\mathbf{A}\mathbf{U}_k) \subset \mathcal{G}_k$*, with* $\mathbf{U}_k$ *explicitly available. Select a scalar* $\mu_{k+1} \neq 0$ *and put* $\omega_{k+1} \equiv 1/\mu_{k+1}$*. Let* $\mathbf{B} \in \mathbb{C}^{n\times n}$ *be a "traditional" preconditioning matrix. In the projections* $\Pi, \widehat{\Pi}$ *and in the operator* $\mathbf{Q}_k$ *of Definition 5.5, let* $\widetilde{\mathbf{R}}$ *be the IDR test matrix and let* $\mathbf{U} = \mathbf{U}_k$*. In* (5.13)*, let* $\mathbf{C}_1 \equiv \mathbf{Q}_k$ *(cf. a correction matrix) and* $\mathbf{C}_2 \equiv \omega_{k+1}\mathbf{B}^{-1}$ *(cf. modified preconditioned Richardson, smoothing step). In particular, assume that* $\mathrm{span}(\mathbf{A}\mathbf{Q}_k) \subset \mathcal{G}_k$*. The recursions for computing the primary and secondary residuals of a* single *IDR cycle are (cf.* (5.13)*)*

$$\begin{cases} \mathbf{r}'_k &= (\mathbf{I} - \mathbf{A}\mathbf{Q}_k)\mathbf{r}_k; \qquad \textit{(see line 10 of Algorithm 5.1)} \\ \mathbf{r}_{k+1} &= (\mathbf{I} - \omega_{k+1}\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'_k, \quad \textit{(see line 29 of Algorithm 5.1)} \end{cases} \tag{5.14}$$

where the secondary residual $\mathbf{r}'_k = \Pi_k \mathbf{r}_k$ *in* $\mathcal{G}'_k \equiv \mathcal{G}_k \cap \widetilde{\mathbf{R}}^{\perp}$ *(i.e., step (i), the* projection step*, cf.* (5.8)*) and the next primary residual* $\mathbf{r}_{k+1}$ *in* $\mathcal{G}_{k+1}$ *(i.e., step (ii), the* dimension reduction step*). The corresponding recursions for the iterates are*

$$\begin{cases} \mathbf{x}'_k &= \mathbf{x}_k + \mathbf{Q}_k\mathbf{r}_k; \qquad \textit{(see line 9 of Algorithm 5.1)} \\ \mathbf{x}_{k+1} &= \mathbf{x}'_k + \omega_{k+1}\mathbf{B}^{-1}\mathbf{r}'_k. \quad \textit{(see line 28 of Algorithm 5.1)} \end{cases} \tag{5.15}$$

Note that the update for $\mathbf{x}'_k$ *is a linear combination of the columns of* $\mathbf{U}_k$*. The resulting operator* $\mathbf{P}_k$ *is (cf.* (5.11)*)*

$$\mathbf{P}_k = \mathbf{Q}_k + \omega_{k+1}\mathbf{B}^{-1} - \omega_{k+1}\mathbf{B}^{-1}\mathbf{A}\mathbf{Q}_k = \omega_{k+1}\mathbf{B}^{-1}\Pi_k + \mathbf{Q}_k. \tag{5.16}$$

The complete recursions for the residual and iterate are

$$\begin{cases} \mathbf{r}_{k+1} &= \mathbf{r}_k - \mathbf{A}\mathbf{P}_k\mathbf{r}_k; \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{P}_k(\mathbf{b} - \mathbf{A}\mathbf{x}_k). \end{cases} \tag{5.17}$$

Proof. Use Definition 5.1 and Definition 5.5. □

This proposition also shows that the IDR(s) method can be seen as a Richardson iteration with a varying deflation–type preconditioner. In particular, the operator $\mathbf{P}_k$ is analogous to the A–DEF1 deflation method described above and in [127, Section 2.3.3]. Also, note that $\mathbf{P}_k^2 \neq \mathbf{P}_k$ and that $\mathbf{P}_k\mathbf{A}$ is non–symmetric even if $\mathbf{A}$ is symmetric. The matrix $\Pi_k\mathbf{A}$ is singular, but the matrix $\mathbf{P}_k\mathbf{A}$ is not. It is shown in [120] that the primary residuals $\mathbf{r}_k$ (and therefore also $\mathbf{r}'_k$) are unique. If there is no risk of ambiguity (that is, if it is clear within which IDR subspace k we are operating or if it is irrelevant), we will sometimes drop the index k of for example $\mathbf{P}_k$ and Π_k.

Note 5.9. The error–propagation operator (or iteration matrix) belonging to IDR(s) is

$$\begin{aligned} \mathbf{I} - \mathbf{P}_k\mathbf{A} &= \mathbf{I} - \omega_{k+1}\mathbf{B}^{-1}\Pi_k\mathbf{A} - \mathbf{Q}_k\mathbf{A} && (5.18) \\ &= \widehat{\Pi}_k - \omega_{k+1}\mathbf{B}^{-1}\mathbf{A}\widehat{\Pi}_k && (5.19) \\ &= (\mathbf{I} - \omega_{k+1}\mathbf{B}^{-1}\mathbf{A})\widehat{\Pi}_k, && (5.20) \end{aligned}$$

where $(\mathbf{I} - \omega_{k+1}\mathbf{B}^{-1}\mathbf{A})$ can be seen as a (post–)smoother and $\widehat{\Pi}_k$ a coarse–grid correction operation [127]. Note that the iteration matrix changes in each cycle k.

Note 5.10. In IDR(s), a single Richardson step is applied in each cycle k to the (deflated and nonsingular) system (cf. (5.17))

$$\mathbf{P}_k\mathbf{A}\mathbf{x}_k = \mathbf{P}_k\mathbf{b}. \tag{5.21}$$

Left	Middle	Right

$\mathbf{y}' = \mathbf{y} + \mathbf{Q}\mathbf{r}$
$\mathbf{r}' = \Pi\mathbf{r}$
for $i = 1$ to s **do**
$\quad\mathbf{u}' = \widehat{\Pi}\mathbf{v}'$
$\quad\mathbf{v}' = \mathbf{A}\mathbf{B}^{-1}\mathbf{u}'$
end for
$\mathbf{y}'' = \mathbf{y}' + \omega\mathbf{r}'$
$\mathbf{r}'' = (\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'$
	$\mathbf{B}^{-1}\mathbf{y}' = \mathbf{B}^{-1}\mathbf{y} + \mathbf{B}^{-1}\mathbf{Q}\mathbf{r}$
	$\mathbf{r}' = \Pi\mathbf{r}$
	for $i = 1$ to s **do**
	$\quad\mathbf{B}^{-1}\mathbf{u}' = \mathbf{B}^{-1}\widehat{\Pi}\mathbf{v}'$
	$\quad\mathbf{v}' = \mathbf{A}\mathbf{B}^{-1}\mathbf{u}'$
	end for
	$\mathbf{B}^{-1}\mathbf{y}'' = \mathbf{B}^{-1}\mathbf{y}' + \omega\mathbf{B}^{-1}\mathbf{r}'$
	$\mathbf{r}'' = (\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'$
		$\mathbf{x}' = \mathbf{x} + \underline{\mathbf{Q}}\mathbf{r}$
		$\mathbf{r}' = \Pi\mathbf{r}$
		for $i = 1$ to s **do**
		$\quad\underline{\mathbf{u}}' = \underline{\widehat{\Pi}}\mathbf{B}^{-1}\mathbf{v}'$
		$\quad\mathbf{v}' = \mathbf{A}\underline{\mathbf{u}}'$
		end for
		$\mathbf{x}'' = \mathbf{x}' + \omega\mathbf{B}^{-1}\mathbf{r}'$
		$\mathbf{r}'' = (\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'$

Figure 5.1: Left: solve $\mathbf{A}\mathbf{B}^{-1}\mathbf{y} = \mathbf{x}$. Middle: premultiply $\mathbf{y}$ and $\mathbf{u}$ by $\mathbf{B}^{-1}$. Right: substitute $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$.

The role of the preconditioner $\mathbf{B}^{-1}$ in Algorithm 5.1 may not be entirely apparent so we will discuss this in some detail. In Figure 5.1 the three steps of introducing a (right) preconditioner $\mathbf{B}^{-1}$ into the IDR(s) method from Algorithm 5.1 are shown. In Figure 4.1 on page 83 a similar discussion is performed, but here we use the deflation language of this chapter. Like before, we only show a single IDR(s) cycle and we drop the index k, starting with a primary residual $\mathbf{r}$, generating the secondary residual $\mathbf{r}'$ and finally the next primary residual $\mathbf{r}''$.

In the left part of Figure 5.1, the IDR(s) method from Algorithm 5.1 is applied to the preconditioned system $\mathbf{A}\mathbf{B}^{-1}\mathbf{y} = \mathbf{b}$ with $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$ as the final solution. In the middle part the vectors $\mathbf{y}$ and $\mathbf{u}$ are premultiplied by $\mathbf{B}^{-1}$ and again we have the final solution $\mathbf{x} = \mathbf{B}^{-1}\mathbf{y}$. Finally, in the right part we set $\underline{\mathbf{u}}' = \mathbf{B}^{-1}\mathbf{u}'$ and the transformed iterates $\mathbf{y}$ are scaled back to the iterates $\mathbf{x}$ of the original system $\mathbf{A}\mathbf{x} = \mathbf{b}$. The notation $\underline{\widehat{\Pi}}$ underlines the fact that this projection uses the preconditioned search matrix $\underline{\mathbf{U}} \equiv \mathbf{B}^{-1}\mathbf{U}$. The same holds for the operator $\underline{\mathbf{Q}}$. In the listing of Algorithm 5.1, this distinction in notation is not explicitly made.

Note that in the right part of Figure 5.1, the projection $\underline{\widehat{\Pi}}$ is operating within the original solution space ($\mathbf{x}$), while Π operates within the transformed solution space ($\mathbf{y}$).

For ease of notation, we will often take $\mathbf{B} = \mathbf{I}$ in the remainder of this chapter.

Search matrices

In the search matrix step (i.e., step (iii)), we need to generate a full rank $n \times s$ matrix with columns in $\mathcal{G}_{k+1}$ of the form $\mathbf{A}\mathbf{U}_{k+1}$ with $\mathbf{U}_{k+1}$ explicitly available. We have to construct this matrix from vectors in $\mathcal{G}_k$. In this IDR subspace, we only have the subspace span($\mathbf{r}_k, \mathbf{A}\mathbf{U}_k$) available with $\mathbf{U}_k$ an $n \times s$ matrix. The vector $\mathbf{y}$ for which $\mathbf{A}\mathbf{y} = \mathbf{r}_k$ will not be available (because that would require to solve a linear system with $\mathbf{A}$). Therefore, the vector $\mathbf{r}_k$ is not helpful. However, in the computation of $\mathbf{r}_{k+1} \in \mathcal{G}_{k+1}$, we also computed $\mathbf{A}\mathbf{r}'_k$, which also belongs to $\mathcal{G}_k$ (use Theorem 5.2, now with $\mu_k = 0$). Hence, we can use vectors from span($\mathbf{A}\mathbf{r}'_k, \mathbf{A}\mathbf{U}_k$).

Using the projection $\Pi_k, \widehat{\Pi}_k$ from Proposition 5.8 we have in particular: $\Pi_k(\mathbf{A}\mathbf{v})$ is in $\mathcal{G}'_k$ if $\mathbf{A}\mathbf{v} \in \mathcal{G}_k$ and $\Pi_k(\mathbf{A}\mathbf{v})$ is of the form $\mathbf{A}\mathbf{u}$ with $\mathbf{u}$ explicitly available: $\mathbf{u} = \widehat{\Pi}_k\mathbf{v}$. Moreover, $\Pi_k(\mathbf{A}\mathbf{v}) = \mathbf{A}\mathbf{v} - \mathbf{A}\mathbf{U}\beta$ with $\beta \equiv \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{v}$ and for the same β we have that $\widehat{\Pi}_k\mathbf{v} = \mathbf{v} - \mathbf{U}\beta$: the projection $\widehat{\Pi}_k$ requires vector updates, but no additional inner products.

With $\mathbf{s} \in \mathbb{C}^n$ and $\widetilde{\mathbf{U}}$ a full rank $n \times s$ matrix such that $[\mathbf{s}, \widetilde{\mathbf{U}}]$ spans an $s+1$ dimensional subspace of span($\mathbf{r}'_k, \mathbf{U}_k$), we can use $\widetilde{\mathbf{U}}$ and $\mathbf{A}\widetilde{\mathbf{U}}$ in the projections Π_k and $\widehat{\Pi}_k$ of (5.8) and take $\mathbf{s}$ to start the construction of $\mathbf{U}_{k+1}$. However, for ease of notation, we formulate Proposition 5.12 for $\widetilde{\mathbf{U}} = \mathbf{U}_k$ and we suggest to use $\mathbf{s} = \mathbf{r}'_k$.

Lemma 5.11. *If* $\mathbf{s} \in \mathcal{G}'_k$, *then* $\Pi_k\mathbf{A}\mathbf{s} = \mathbf{A}\widehat{\Pi}_k\mathbf{s} \in \mathcal{G}'_k$.

Proof. From Theorem 5.4 we learn that $\mathbf{As} \in \mathbf{A}(\mathcal{G}'_k) \subset \mathcal{G}_k$, whence $\Pi_k\mathbf{As} \in \mathcal{G}'_k$. $\Pi_k\mathbf{As} = \mathbf{A}\widehat{\Pi}_k\mathbf{s}$ follows from Lemma 5.6:(ii). □

Proposition 5.12. $\mathbf{A}\widehat{\Pi}_k\mathbf{r}'_k \in \mathcal{G}'_k$. *Let* $\mathbf{u}'_1 \in \mathbb{C}^n$ *(for instance,* $\mathbf{u}'_1 = \widehat{\Pi}_k\mathbf{r}'_k$*, see line 15 of Algorithm 5.1).*
If $\mathbf{Au}'_1 \in \mathcal{G}'_k$*, and* $\mathbf{U}'_k$ *is a matrix with columns in* $\mathcal{K}_s(\widehat{\Pi}_k\mathbf{A}, \mathbf{u}'_1)$*, then*

$$\mathrm{span}(\mathbf{AU}'_k) \subset \mathcal{K}_s(\Pi_k\mathbf{A}, \mathbf{Au}'_1) \subset \mathcal{G}'_k \quad \textit{and} \quad \mathrm{span}(\mathbf{A}(\mu_{k+1}\mathbf{I} - \mathbf{A})\mathbf{U}'_k) \subset \mathcal{G}_{k+1}.$$

Proof. The first claim follows by combining Proposition 5.8 and Lemma 5.11. The inclusion $\mathcal{K}_s(\Pi_k\mathbf{A}, \mathbf{Au}'_1) \subset \mathcal{G}'_k$ follows from Lemma 5.11 and an induction argument. Further,

$$\mathbf{A}\mathcal{K}_s(\widehat{\Pi}_k\mathbf{A}, \mathbf{u}'_1) = \mathcal{K}_s(\mathbf{A}\widehat{\Pi}_k, \mathbf{Au}'_1) = \mathcal{K}_s(\Pi_k\mathbf{A}, \mathbf{Au}'_1) \tag{5.22}$$

(use Lemma 5.6:(ii)). □

The proposition tells us that any set of s linearly independent vectors in $\mathcal{K}_s(\widehat{\Pi}_k\mathbf{A}, \mathbf{u}'_1)$ forms a matrix $\mathbf{U}'_k$ for which the columns of $\mathbf{A}(\mathbf{I} - \omega\mathbf{A})\mathbf{U}'_k$ are in $\mathcal{G}_{k+1}$: $\mathbf{U}_{k+1} \equiv (\mathbf{I} - \omega\mathbf{A})\mathbf{U}'_k$ forms an appropriate search matrix for the next IDR step (see line 25 of Algorithm 5.1).

In Algorithm 5.1, a power basis for the Krylov subspaces is generated. For larger values of s (as $s > 4$) a more stable basis might be required. The choice of such a basis and its efficient computation is considered in detail in [116].

5.2.4 IDR algorithms

In this section we will make some general remarks about IDR(s) algorithms. Algorithm 5.1 describes a way to recursively generate $s+1$ dimensional subspaces of $\mathcal{G}_k$:

Proposition 5.13. *If* $[\mathbf{r}_j, \mathbf{V}_j]$ *has rank* $s+1$ *for* $j \le k$*, then* $\mathrm{span}(\mathbf{r}_k, \mathbf{V}_k) \subset \mathcal{G}_k$*.*

Proof. Clearly, $\mathrm{span}(\mathbf{r}_0, \mathbf{V}_0) \subset \mathcal{G}_0$. Assume that $\mathrm{span}(\mathbf{r}_{k-1}, \mathbf{V}_{k-1}) \subset \mathcal{G}_{k-1}$. Then $\Pi_{k-1}\mathbf{r}_{k-1} \in \mathcal{G}'_{k-1}$ and, by Lemma 5.11, $\mathcal{K}_{s+1}(\Pi_{k-1}\mathbf{A}, \Pi_{k-1}\mathbf{r}_{k-1}) \subset \mathcal{G}'_{k-1}$. Hence, $\mathrm{span}(\mathbf{r}_k, \mathbf{V}_k) \subset (\mathbf{A} - \mu_k\mathbf{I})\mathcal{G}'_{k-1} = \mathcal{G}_k$. □

The listing of Algorithm 5.1 is somewhat different from more "common" listings of IDR(s) algorithms, such as the IDR(s) listings in Chapter 4 of this thesis. To give a concrete example, Algorithm 5.2 shows such a traditional listing of an IDR(s) algorithm using the deflation language of this chapter. For comparison purposes, the preconditioner $\mathbf{B}$ is explicitly added. Like Algorithm 5.1, this listing is one of the most basic formulations of an IDR(s) algorithm and it is not intended for practical applications. It was chosen to facilitate comparing with Algorithm 5.1. For efficient and stable variants of IDR(s) algorithms, see Chapter 4 of this thesis and also [140]. We will now discuss the differences and similarities between Algorithm 5.1 and Algorithm 5.2.

In Algorithm 5.2, the vectors $\mathbf{g}_i$ are directly lifted to the (primary) IDR subspace $\mathcal{G}_k$, while Algorithm 5.1 explicitly generates s (secondary) vectors $\mathbf{v}'_i$ in the (secondary) IDR subspace $\mathcal{G}_{k-1} \cap \widetilde{\mathbf{R}}^\perp$. The main goal of traditional listings such as Algorithm 5.2 is to generate linearly independent vectors (often denoted by $\mathbf{g}$ as in Algorithm 5.2) in the IDR subspace $\mathcal{G}_k$ of unknown (possibly large) dimension. These vectors *do not form a basis* of the subspace $\mathcal{G}_k$. In contrast, the vectors $\mathbf{v}'$ are explicitly generated to form a basis of $\mathcal{K}_s(\Pi_{k-1}\mathbf{AB}^{-1}, \Pi_{k-1}\mathbf{AB}^{-1}\mathbf{r}'_{k-1})$. To underline these distinctions, we use different letters for these sets of vectors.

However, it can be shown that in both listings we are in fact generating vectors $\mathbf{u}_i$ for a basis of the same Krylov subspace. In particular, the columns of $\mathbf{U}_a \equiv \mathbf{U}_k$ in line 25 of Algorithm 5.1 form a basis of the Krylov subspace $(\mathbf{I} - \omega_k\mathbf{B}^{-1}\mathbf{A})\mathcal{K}_s(\widehat{\Pi}_{k-1}\mathbf{B}^{-1}\mathbf{A}, \widehat{\Pi}_{k-1}\mathbf{B}^{-1}\Pi_{k-1}\mathbf{r}_{k-1})$, while the columns of $\mathbf{U}_b \equiv \mathbf{U}_k$ in line 14 of Algorithm 5.2 form a basis of the Krylov subspace $\mathcal{K}_s(\mathbf{P}_{k-1}\mathbf{A}, \mathbf{P}_{k-1}\mathbf{r}_k)$. The following proposition shows that these subspaces are identical.

Algorithm 5.2 IDR(s) as a deflation method: a "traditional" listing.

INPUT: $\mathbf{A} \in \mathbb{C}^{n\times n}; \mathbf{x}_0, \mathbf{b} \in \mathbb{C}^n; \widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$; preconditioner $\mathbf{B} \in \mathbb{C}^{n\times n}$; tolerance tol
OUTPUT: Approximate solution $\mathbf{x}$ such that $||\mathbf{b} - \mathbf{Ax}|| \leq \text{tol}$

1: // *Initiation*
2: $\Pi_{-1} = \mathbf{I}; \mathbf{Q}_{-1} = \mathbf{0}; \omega_0 = 1$
3: Compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
4: // *Loop over nested $\mathcal{G}_k$ spaces*
5: **for** $k = 0, 1, \ldots$ **do**
6: // *Compute s independent vectors $\mathbf{g}$ in $\mathcal{G}_k$ using vectors from $\mathcal{G}_{k-1}$*
7: **let** $\Pi_{k-1} = \mathbf{I} - \mathbf{A}\mathbf{Q}_{k-1}$ where $\mathbf{Q}_{k-1} = \mathbf{U}_{k-1}(\widetilde{\mathbf{R}}^*\mathbf{G}_{k-1})^{-1}\widetilde{\mathbf{R}}^*$
8: **let** $\mathbf{P}_{k-1} = \omega_k \mathbf{B}^{-1}\Pi_{k-1} + \mathbf{Q}_{k-1}$
9: **let** $\mathbf{g}_0 = \mathbf{r}_k$ // *(cf. line 15 of Algorithm 5.1)*
10: **for** $i = 1$ to s **do**
11: $\mathbf{u}_i = \mathbf{P}_{k-1}\mathbf{g}_{i-1}$ // *(cf. eq.* (5.16)*)*
12: $\mathbf{g}_i = \mathbf{A}\mathbf{u}_i \in \mathcal{G}_k$ // *(cf. line 18 of Algorithm 5.1)*
13: **end for**
14: **let** $\mathbf{U}_k = [\mathbf{u}_1, \ldots, \mathbf{u}_s]$ and $\mathbf{G}_k = [\mathbf{g}_1, \ldots, \mathbf{g}_s]$
15: **let** $\Pi_k = \mathbf{I} - \mathbf{A}\mathbf{Q}_k$ where $\mathbf{Q}_k = \mathbf{U}_k(\widetilde{\mathbf{R}}^*\mathbf{G}_k)^{-1}\widetilde{\mathbf{R}}^*$
16: $\mathbf{x}'_k = \mathbf{x}_k + \mathbf{Q}_k\mathbf{r}_k$ // *(cf. line 9 of Algorithm 5.1)*
17: $\mathbf{r}'_k = \Pi_k\mathbf{r}_k$ // *(cf. line 10 of Algorithm 5.1)*
18: // *Entering $\mathcal{G}_{k+1}$*
19: $\omega_{k+1} = \arg\min_\omega ||(\mathbf{I} - \omega\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'_k||$ // *(cf. line 23 of Algorithm 5.1)*
20: // *Compute next primary residual $\mathbf{r}_{k+1} \in \mathcal{G}_{k+1}$ and corresponding iterate $\mathbf{x}_{k+1}$*
21: $\mathbf{x}_{k+1} = \mathbf{x}'_k + \omega_{k+1}\mathbf{B}^{-1}\mathbf{r}'_k$ // *(cf. line 28 of Algorithm 5.1)*
22: $\mathbf{r}_{k+1} = (\mathbf{I} - \omega_{k+1}\mathbf{A}\mathbf{B}^{-1})\mathbf{r}'_k$ // *(cf. line 29 of Algorithm 5.1)*
23: **if** $||\mathbf{r}_{k+1}|| \leq \text{tol}$ **then** break **end if** // *(cf. line 30 of Algorithm 5.1)*
24: **end for**

Proposition 5.14. *Let* $\mathbf{P}, \Pi,$ *and* $\widehat{\Pi}$ *be as in Prop 5.8 and let* $\mathbf{B} = \mathbf{I}$. *Then we have*

$$\mathcal{K}_s(\mathbf{P}_{k-1}\mathbf{A}, \mathbf{P}_{k-1}\mathbf{r}_k) = (\mathbf{I} - \omega_k\mathbf{A})\mathcal{K}_s(\widehat{\Pi}_{k-1}\mathbf{A}, \widehat{\Pi}_{k-1}\Pi_{k-1}\mathbf{r}_{k-1}). \tag{5.23}$$

Proof. Using Lemma 5.6 and Proposition 5.8, we have that

$$\begin{aligned}
\mathbf{P}_{k-1}\mathbf{r}_k &= (\omega_k\Pi_{k-1} + \mathbf{Q}_{k-1})(\mathbf{I} - \omega_k\mathbf{A})\Pi_{k-1}\mathbf{r}_{k-1} \\
&= \omega_k\Pi_{k-1}^2\mathbf{r}_{k-1} - \omega_k^2\Pi_{k-1}\mathbf{A}\Pi_{k-1}\mathbf{r}_{k-1} + \mathbf{Q}_{k-1}\Pi_{k-1}\mathbf{r}_{k-1} - \omega_k\mathbf{Q}_{k-1}\mathbf{A}\Pi_{k-1}\mathbf{r}_{k-1} \\
&= \omega_k(\mathbf{I} - \omega_k\Pi_{k-1}\mathbf{A} - \mathbf{Q}_{k-1}\mathbf{A})\Pi_{k-1}\mathbf{r}_{k-1} \\
&= \omega_k(\widehat{\Pi}_{k-1} - \omega_k\mathbf{A}\widehat{\Pi}_{k-1})\Pi_{k-1}\mathbf{r}_{k-1} \\
&= \omega_k(\mathbf{I} - \omega_k\mathbf{A})\widehat{\Pi}_{k-1}\Pi_{k-1}\mathbf{r}_{k-1}.
\end{aligned}$$

Using these arguments, it can be shown that for $\mathbf{U}_b$ of Algorithm 5.2 we have

$$\begin{aligned}
\text{span}(\mathbf{U}_b) &= \mathcal{K}_s(\mathbf{P}_{k-1}\mathbf{A}, \mathbf{P}_{k-1}\mathbf{r}_k) \\
&= \mathcal{K}_s((\omega_k\Pi_{k-1} + \mathbf{Q}_{k-1})\mathbf{A}, \omega_k(\mathbf{I} - \omega_k\mathbf{A})\widehat{\Pi}_{k-1}\Pi_{k-1}\mathbf{r}_{k-1}) \\
&= (\mathbf{I} - \omega_k\mathbf{A})\mathcal{K}_s(\widehat{\Pi}_{k-1}\mathbf{A}, \widehat{\Pi}_{k-1}\Pi_{k-1}\mathbf{r}_{k-1}) \\
&= \text{span}(\mathbf{U}_a),
\end{aligned}$$

Algorithm 5.3 IDR(s) as a deflation method: the IDR(s)-biortho variant from Algorithm 4.1.

INPUT: $\mathbf{A} \in \mathbb{C}^{n\times n}; \mathbf{x}, \mathbf{b} \in \mathbb{C}^n; \widetilde{\mathbf{R}} \in \mathbb{C}^{n\times s}$; preconditioner $\mathbf{B} \in \mathbb{C}^{n\times n}$; tolerance tol
OUTPUT: Approximate solution $\mathbf{x}$ such that $||\mathbf{b} - \mathbf{Ax}|| \leq \varepsilon$

1: $\mathbf{Q}_{-1} = \mathbf{0} \in \mathbb{C}^{n\times n}; \omega_0 = 1$
2: Compute $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$
3: **for** $k = 0, 1, \ldots$ **do**
4:     *// Compute s independent vectors $\mathbf{g}$ in $\mathcal{G}_k$ using vectors from $\mathcal{G}_{k-1}$*
5:     **for** $i = 1$ to s **do**
6:         **let** $\Pi_{k-1} \equiv \mathbf{I} - \mathbf{AQ}_{k-1}$ where $\mathbf{Q}_{k-1} \equiv \mathbf{U}_{k-1}(\widetilde{\mathbf{R}}^*\mathbf{G}_{k-1})^{-1}\widetilde{\mathbf{R}}^*$ and $\widetilde{\mathbf{R}} \equiv [\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_s]$
7:         **let** $\mathbf{P}_{k-1} \equiv \omega_k \mathbf{B}^{-1}\Pi_{k-1} + \mathbf{Q}_{k-1}$
8:         $\widehat{\mathbf{u}}_i = \mathbf{P}_{k-1}\mathbf{r}$
9:         $\widehat{\mathbf{g}}_i = \mathbf{A}\widehat{\mathbf{u}}_i$ $\in \mathcal{G}_k$
10:         **let** $\mathbf{G} \equiv [\mathbf{g}_1, \ldots, \mathbf{g}_{i-1}]$, $\mathbf{U} \equiv [\mathbf{u}_1, \ldots, \mathbf{u}_{i-1}]$, and $\widetilde{\mathbf{R}} \equiv [\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_{i-1}]$ for $i > 1$
11:         **let** $\Pi \equiv \mathbf{I} - \mathbf{AQ}$ and $\widehat{\Pi} \equiv \mathbf{I} - \mathbf{QA}$ where $\mathbf{Q} \equiv \mathbf{U}(\widetilde{\mathbf{R}}^*\mathbf{G})^{-1}\widetilde{\mathbf{R}}^*$
12:         $\mathbf{u}_i = \widehat{\Pi}\widehat{\mathbf{u}}_i$ $\perp_{\mathbf{A}} \widetilde{\mathbf{R}}$
13:         $\mathbf{g}_i = \Pi\widehat{\mathbf{g}}_i$ $\in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$
14:         **let** $\mathbf{G} \equiv [\mathbf{g}_1, \ldots, \mathbf{g}_i]$, $\mathbf{U} \equiv [\mathbf{u}_1, \ldots, \mathbf{u}_i]$, and $\widetilde{\mathbf{R}} \equiv [\widetilde{\mathbf{r}}_1, \ldots, \widetilde{\mathbf{r}}_i]$
15:         **let** $\Pi \equiv \mathbf{I} - \mathbf{AQ}$ and $\widehat{\Pi} \equiv \mathbf{I} - \mathbf{QA}$ where $\mathbf{Q} \equiv \mathbf{U}(\widetilde{\mathbf{R}}^*\mathbf{G})^{-1}\widetilde{\mathbf{R}}^*$
16:         $\mathbf{x} = \mathbf{x} + \mathbf{Qr}$
17:         $\mathbf{r} = \Pi\mathbf{r}$ $\in \mathcal{G}_k \cap \widetilde{\mathbf{R}}^\perp$
18:     **end for**
19:     *// Search matrix $\mathbf{U}_k$ for next IDR projection step and $\mathbf{G}_k$ with columns in $\mathcal{G}_k$:*
20:     **let** $\mathbf{G}_k \equiv \mathbf{G}$ and $\mathbf{U}_k \equiv \mathbf{U}$
21:     *// Entering $\mathcal{G}_{k+1}$*
22:     $\widetilde{\mathbf{v}} = \mathbf{B}^{-1}\mathbf{r}$
23:     $\mathbf{t} = \mathbf{A}\widetilde{\mathbf{v}}$
24:     $\omega_{k+1} = \arg\min_\omega ||(\mathbf{I} - \omega\mathbf{AB}^{-1})\mathbf{r}||$
25:     *// Compute next primary residual $\mathbf{r} \in \mathcal{G}_{k+1}$ and corresponding iterate $\mathbf{x}$*
26:     $\mathbf{x} = \mathbf{x} + \omega_{k+1}\widetilde{\mathbf{v}}$
27:     $\mathbf{r} = \mathbf{r} - \omega_{k+1}\mathbf{t}$
28:     **if** $||\mathbf{r}|| \leq$ tol **then** break **end if**
29: **end for**

for $\mathbf{U}_a$ of Algorithm 5.1. Therefore, both sets of vectors span the same Krylov subspace, which proves the proposition. □

Note that the secondary residual $\mathbf{r}'_k$ is explicitly generated in both listings, since it is needed to perform the dimension reduction step.

Also, the "traditional" listing of the IDR(s)-biortho method from Algorithm 4.1 (and Algorithm 4.2) on page 75 uses the letter $\mathbf{v}$ (line 10 of Algorithm 4.1 and line 11 of Algorithm 4.2) to denote vectors that are orthogonal to $\widetilde{\mathbf{R}}$, which is consistent with the notation of Algorithm 5.1.

For both completeness and illustrative purposes, we reproduce in Algorithm 5.3 the IDR(s)-biortho method from Algorithm 4.1 (and thus also the IDR(s)-minsync method from Algorithm 4.2) using the deflation language of this chapter. Note that this variant also computes auxiliary residuals.

5.3 The IDR projection theorem

In the previous section it has been shown that the IDR(s) method can be seen as a Richardson iteration preconditioned by a variable deflation–type preconditioner. Therefore, it makes sense to investigate the spectra of the sequence of deflated systems, i.e.,

$$\sigma(\mathbf{P}_k\mathbf{A}), \qquad k = 0, 1, \ldots . \tag{5.24}$$

The system $\mathbf{P}_k\mathbf{A}$ is called the kth *active system* of the iteration process.

In Section 5.3.1 the spectrum of the active system is investigated for a *single* IDR(s) cycle. Section 5.3.2 contains the main result of this chapter, which relates the spectra of the active systems of *multiple* IDR(s) cycles. We will show that in IDR(s) the spectrum of $\mathbf{P}_k\mathbf{A}$ become increasingly more *clustered* with increasing k. Also, the spectrum of $\mathbf{P}_k\mathbf{A}$ is related to the active spectra of all the previous cycles $0, \ldots, k-1$. Section. 5.3.3 contains numerical examples to illustrate the IDR projection theorem. In Section 5.3.4 some possible interpretations of the IDR projection theorem are discussed.

5.3.1 Single IDR(s) cycle

In the following, we partly follow [62] and [127]. We will first show some properties of the active system of a *single* IDR(s) cycle k. Therefore, the index k is dropped in this section.

Lemma 5.15 (cf. Lemma 2.5 [62]). *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be nonsingular. For all full ranked rectangular matrices* $\mathbf{U}$ *and* $\widetilde{\mathbf{R}}$, $\mathbf{B}^{-1}\Pi\mathbf{A}$ *and* $\mathbf{PA}$ *have* s *eigenvalues equal to* 0 *and* 1, *respectively.*

Proof. From Lemma 5.6 it follows that $\mathbf{B}^{-1}\Pi\mathbf{AU} = \mathbf{0}_{n,s}$ and $(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{AU} = \mathbf{U}$. Additionally, $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_s]$ are the eigenvectors of $\mathbf{B}^{-1}\Pi\mathbf{A}$ and $(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{A}$ associated with the eigenvalues 0 and 1, respectively. □

Theorem 5.16 (cf. [127], cf. Theorem 2.8 [62]). *For all cycles we have*

$$\sigma(\mathbf{B}^{-1}\Pi\mathbf{A}) = \{0\} \cup \{\lambda'_{s+1}, \ldots, \lambda'_n\} \tag{5.25}$$

$$\Leftrightarrow$$

$$\sigma((\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{A}) = \{1\} \cup \{\lambda'_{s+1}, \ldots, \lambda'_n\}, \tag{5.26}$$

where the eigenvalues 0 *and* 1 *both have multiplicity* s.

Proof. For the implication $\Leftarrow$, see [62, Theorem 2.8]. For the implication $\Rightarrow$, note that for $\mathbf{u}_i$ with $i = 1, \ldots, s$ where $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_s]$ we have $(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{AU} = \mathbf{U}$ and $\mathbf{B}^{-1}\Pi\mathbf{AU} = \mathbf{0}_{n,s}$, so the eigenvectors $\mathbf{U}$ of $(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{A}$ corresponding to the unit eigenvalues are the same as those corresponding to the zero eigenvalues of $\mathbf{B}^{-1}\Pi\mathbf{A}$.

For $i = s+1, \ldots, n$, suppose that the eigenvectors $\{\mathbf{v}_i\}$ satisfy $\mathbf{B}^{-1}\Pi\mathbf{A}\widehat{\Pi}\mathbf{v}_i = \lambda_i\widehat{\Pi}\mathbf{v}_i$ with corresponding eigenvalues $\{\lambda_i\}$. Then (using (vii) from Lemma 5.6)

$$\begin{aligned}(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{A}\widehat{\Pi}\mathbf{v}_i &= \mathbf{B}^{-1}\Pi\mathbf{A}\widehat{\Pi}\mathbf{v}_i + \mathbf{QA}\widehat{\Pi}\mathbf{v}_i &(5.27)\\ &= \lambda_i\widehat{\Pi}\mathbf{v}_i. &(5.28)\end{aligned}$$

So the eigenvalues of $(\mathbf{B}^{-1}\Pi+\mathbf{Q})\mathbf{A}$ are the same as the eigenvalues of $\mathbf{B}^{-1}\Pi\mathbf{A}$, with eigenvectors $\widehat{\Pi}\mathbf{v}_i$. □

In other words, $(\mathbf{B}^{-1}\Pi + \mathbf{Q})\mathbf{A}$ has eigenvectors $\widehat{\Pi}\mathbf{v}_i$, with $\widehat{\Pi}\mathbf{v}_i$ the eigenvectors of $\mathbf{B}^{-1}\Pi\mathbf{A}$.

5.3.2 Main result: IDR projection theorem

Since the spectra of $\mathbf{P}_k\mathbf{A}$ and $\Pi_k\mathbf{A}$ merely differ in the sense that the zero eigenvalues of $\Pi_k\mathbf{A}$ are shifted to one in $\mathbf{P}_k\mathbf{A}$ (cf. Theorem 5.16), we will mainly focus on the spectrum of $\Pi_k\mathbf{A}$ from now on. We first state the main result of this chapter: the IDR projection theorem. It describes the complete spectrum of the active IDR(s) systems $\Pi_k\mathbf{A}$, relating the spectrum of the active system of cycle k to all the previous cycles.

Theorem 5.17 (IDR Projection Theorem)**.** *Let $k \in \mathbb{N}_0$ and let $\mathbf{W}$ be an $n \times s$ matrix such that $\mathbf{W} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})$. The eigenvalues of $\Pi_k\mathbf{A}$ are in*

$$\sigma(\Pi_k\mathbf{A}) = \{0\} \cup \{\lambda \mid P(\lambda) = 0\} \cup \{\lambda \mid \det(\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}) = 0\}, \tag{5.29}$$

where the polynomial $P(\lambda) \equiv \prod_{j=1}^{k}(\lambda - \mu_j)$ as defined in Theorem 5.4. The zero eigenvalue and the zeros μ_j of the polynomial P are all eigenvalues of $\Pi_k\mathbf{A}$ that have geometric multiplicity at least s.

Before we give the proof of the IDR projection theorem, we will give some preliminary results. Let $\mathbf{V}$ be an $n \times s$ matrix such that $\widetilde{\mathbf{R}}^*\mathbf{V}$ is non-singular. Define (cf. Definition 5.5)

$$\Pi_k \equiv \mathbf{I} - \mathbf{V}\mathbf{E}^{-1}\widetilde{\mathbf{R}}^* \quad \text{where} \quad \mathbf{E} \equiv \widetilde{\mathbf{R}}^*\mathbf{V}. \tag{5.30}$$

Note that

$$\widetilde{\mathbf{R}}^*\Pi_k = \mathbf{0}. \tag{5.31}$$

In particular, we have $\mathbf{V} \equiv \mathbf{A}\mathbf{U}_k$ in IDR (cf. Algorithm 5.1 and Prop 5.13).

Proposition 5.18. *Assume* $\text{span}(\mathbf{V}) \subset \mathcal{G}_k$*. Then each $\nu \in \{\mu_1, \ldots, \mu_k\}$ is an eigenvalue of $\Pi_k\mathbf{A}$ with geometric multiplicity at least s.*

Proof. The equality in (5.6) tells us that $\mathbf{V} = (\mathbf{A} - \nu\mathbf{I})\mathbf{W}$ for some $n \times s$ matrix $\mathbf{W} \perp \widetilde{\mathbf{R}}$. Hence

$$(\Pi_k\mathbf{A} - \nu\mathbf{I})\mathbf{W} = (\mathbf{A} - \nu\mathbf{I})\mathbf{W}(\mathbf{I} - \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{W}).$$

Since $\mathbf{E} = \widetilde{\mathbf{R}}^*\mathbf{V} = \widetilde{\mathbf{R}}^*(\mathbf{A} - \nu\mathbf{I})\mathbf{W} = \widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{W}$, we see that $\mathbf{W}$ is in the kernel of $\Pi_k\mathbf{A} - \nu\mathbf{I}$, that is, $\text{span}(\mathbf{W})$ consists of eigenvectors of $\Pi_k\mathbf{A}$ with eigenvalue ν. □

Again using the equality in (5.6), we know that there exists a $\mathbf{W} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})$ such that $\mathbf{V} = P(\mathbf{A})\mathbf{W}$ for some polynomial P of exact degree k. In the following, let $\mathbf{W}$ be such a matrix and assume $\mathbf{E} \equiv \widetilde{\mathbf{R}}^*\mathbf{V}$ is non–singular.

Proposition 5.19. *Assume $\lambda \in \mathbb{C}$ is not an eigenvalue of $\mathbf{A}$ and $\lambda P(\lambda) \neq 0$. Then, a non-trivial vector $\mathbf{x}$ is an eigenvector of $\Pi_k\mathbf{A}$ with eigenvalue λ if and only if $\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha$ for some $\alpha \in \mathbb{C}^s$, $\alpha \neq 0$ such that $\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}\alpha = 0$.*

Proof. Assume $\Pi_k\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$. Since $\Pi_k\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{x} - \mathbf{V}\mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{x}$, we see that $\mathbf{A}\mathbf{x} - \lambda\mathbf{x} = \mathbf{V}\alpha$ for some $\alpha \in \mathbb{C}^s$ (actually, $\alpha = \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{x}$). The scalar λ is not an eigenvalue of $\mathbf{A}$. Therefore, $\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha$. Hence,

$$\alpha = \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{A}(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha = \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\mathbf{V}\alpha + \mathbf{E}^{-1}\widetilde{\mathbf{R}}^*\lambda(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha.$$

Since $\mathbf{E} = \widetilde{\mathbf{R}}^*\mathbf{V}$, this is equivalent to $0 = \lambda\mathbf{E}^{-1}\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha$. Therefore,

$$0 = \widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha. \tag{5.32}$$

Here, we used that, by assumption $\lambda \neq 0$. Conversely, if (5.32) holds, then it is easy to check that $\mathbf{x} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha$ defines an eigenvector of $\Pi_k\mathbf{A}$.

Note that $(\mathbf{A} - \lambda\mathbf{I})^{-1}(\mathbf{A} - \mu\mathbf{I}) = \mathbf{I} + (\lambda - \mu)(\mathbf{A} - \lambda\mathbf{I})^{-1}$. Since, $P(\zeta) = \gamma\Pi_{j=1}^{k}(\zeta - \mu_j)$ ($\zeta \in \mathbb{C}$) for certain scalars $\mu_j \in \mathbb{C}$ (the zeros of P) and a scalar $\gamma \in \mathbb{C}$ (a scaling factor), and $\mathbf{W} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})$, we see that $\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}P(\mathbf{A})\mathbf{W} = P(\lambda)\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}$. Hence, (5.32) holds if and only if $P(\lambda) = 0$ or $\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}\alpha = 0$. □

Note 5.20. Note that the above proof relies on the fact that $\mathbf{W} \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})$.

Note 5.21. Note that the above arguments also prove Proposition 5.18: Because, if $\lambda = \mu_i$ for some zero μ_i of P, then, for any $\alpha \in \mathbb{C}^s$, we have that $P(\lambda)\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}\alpha = 0$ and any $\mathbf{x} \equiv (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{V}\alpha$ is an eigenvector of $\Pi_k\mathbf{A}$.

We can now give a proof of Theorem 5.17:

Proof. [Proof of IDR Projection Theorem] Use (iii) of Lemma 5.6 and use Proposition 5.19. □

The proposition belows shows that in IDR the latter set of (5.29) is independent of P:

Proposition 5.22. *In IDR,* $\mathbf{W}$ *depends on* k *but is independent of* P*, that is, independent of* $\mu_1, \ldots, \mu_k$.

Proof. The projection Π_k applied to some vector forms vectors of the form $P_k(\mathbf{A})\mathbf{s} - P_k(\mathbf{A})\mathbf{W}\alpha = P_k(\mathbf{A})(\mathbf{s} - \mathbf{W}\alpha) \perp \widetilde{\mathbf{R}}$. If $\mathbf{s}, \mathbf{W} \perp \mathcal{K}_{k-1}(\mathbf{A}^*, \widetilde{\mathbf{R}})$, then the orthogonality condition from Π_k is equivalent to requiring $\mathbf{s} - \mathbf{W}\alpha \perp \mathcal{K}_k(\mathbf{A}^*, \widetilde{\mathbf{R}})$. In particular, the vector $\mathbf{s} - \mathbf{W}\alpha$ is independent of P if $\mathbf{s}$ and $\mathbf{W}$ are independent of P.

Following the inductive construction of the $\mathbf{V}$ matrices in IDR (as we saw in Theorem 5.2), the above argument proves the proposition. □

5.3.3 Numerical examples

To illustrate the clustering effect of IDR(s) algorithms, we will inspect the spectra of $\mathbf{A}$ and $\Pi_k\mathbf{A}$ of each cycle k while solving a small test problem using the IDR algorithm from Algorithm 5.1. In other words, we take $\mathbf{V} = \mathbf{A}\mathbf{U}_k$ in the projection Π from (5.30). We will use either a minimum residual strategy for computing ω_k or set $\omega_k = 1$ in each cycle $k > 0$.

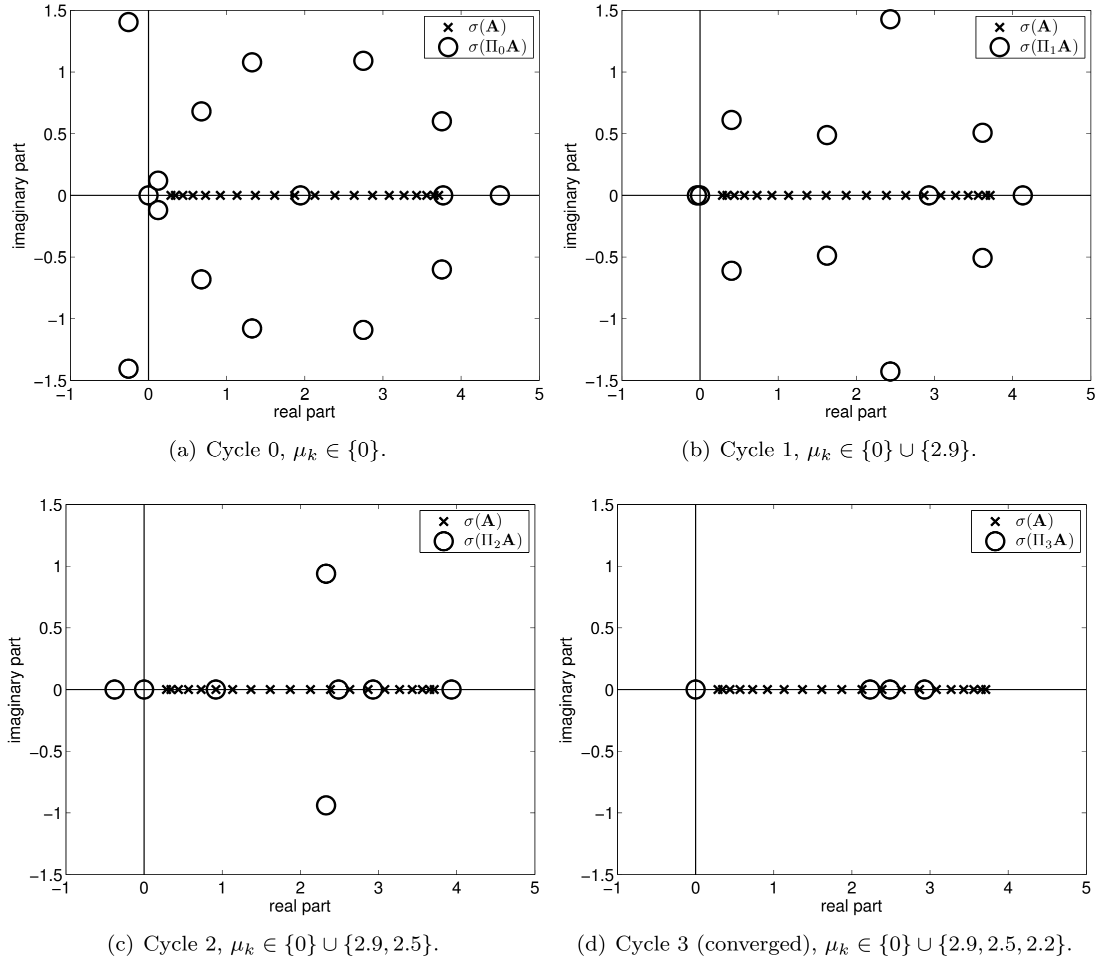
The test problem is a finite difference discretisation of 1D convection–diffusion problem using central differences for the first derivative. The system has size $n = 20$ and the mesh Péclet number p_h is equal to $1/2$. To more precise, the diagonal elements of the coefficient matrix $\mathbf{A}$ are equal to 2, while on the subdiagonal and superdiagonal the values are equal to $-1 - p_h$ and $-1 + p_h$, respectively. The right–hand side vector is equal to $\mathbf{b} = [1 + p_h, 0, \ldots, 0, 1 - p_h]^\top$. Also, we put $s = 5$, so the iteration converges in $n/s = 20/5 = 4$ cycles in exact arithmetic and we set $\mathbf{x}_0 = \mathbf{0}$.

Figure 5.2 shows the spectra of the active systems if a minimum residual strategy for computing ω_k for $k > 0$ is used. Put $\mu_0 = 0$. This gives $\mu_k \in \{0\} \cup \{2.9, 2.5, 2.2\}$ for $k = 0, 1, 2, 3$, resulting in the final clustered spectrum shown in Figure 5.2(d), which solely consists of four eigenvalues each with multiplicity $s = 5$.

Shown in Figure 5.3 are the spectra of the active systems for all four cycles if $\mu_k = 1$ for $k = 1, 2, 3$. As a result, the active system has $10, 15$, and 20 unit eigenvalues in cycle $1, 2$, and 3, respectively.

These results also indicate that the (total) spectrum seems to increasingly converge towards the values of μ_k with each cycle.

Also, it can be observed from these experiments that the “non–clustered” part of the spectrum is independent of the choices for $\mu_1, \ldots, \mu_k$, as indicated by Proposition 5.22.



(a) Cycle 0, $\mu_k \in \{0\}$.

(b) Cycle 1, $\mu_k \in \{0\} \cup \{2.9\}$.

(c) Cycle 2, $\mu_k \in \{0\} \cup \{2.9, 2.5\}$.

(d) Cycle 3 (converged), $\mu_k \in \{0\} \cup \{2.9, 2.5, 2.2\}$.

Figure 5.2: $\mathbf{B} = \mathbf{I}, n = 20, s = 5$, four cycles in total, $\mu_k \in \{0\} \cup \{2.9, 2.5, 2.2\}$ for $k = 0, 1, 2, 3$.

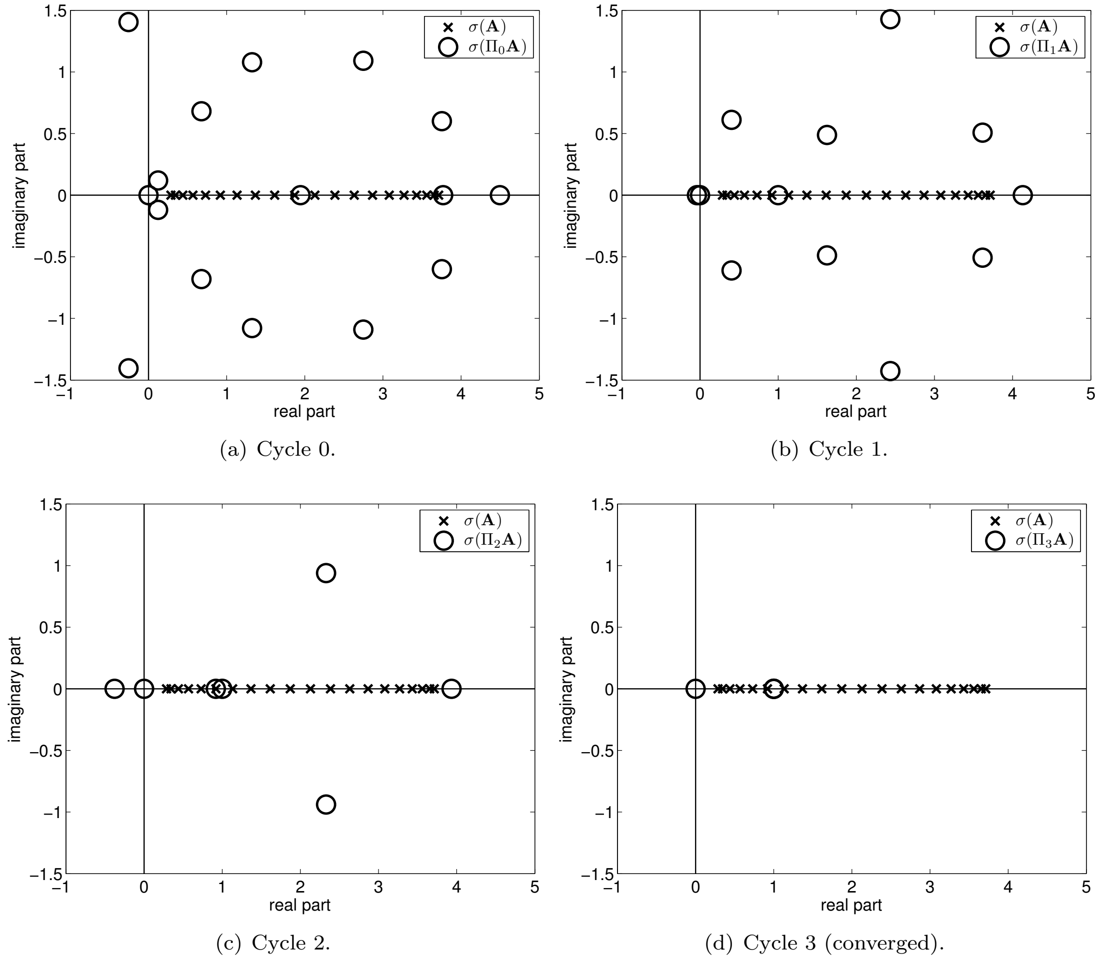
5.3.4 Discussion

In each IDR(s) cycle k, s *additional* eigenvalues of $\mathbf{P}_k\mathbf{A}$ are shifted to μ_k and the matrix $\mathbf{P}_k\mathbf{A}$ (and therefore $\Pi_k\mathbf{A}$) has $k+1$ eigenvalues of geometric multiplicity s. The IDR projection theorem holds independently of the way a basis for $\mathcal{K}_s(\Pi_{k-1}\mathbf{A}\mathbf{B}^{-1}, \Pi_{k-1}\mathbf{A}\mathbf{B}^{-1}\mathbf{r}'_{k-1})$ is computed.

In standard deflation methods, the deflation subspace matrices $\mathbf{U}$ and $\widetilde{\mathbf{R}}$ are often equal to each other and consists of (approximate) eigenvectors belonging to eigenvalues of $\mathbf{A}$ that are small in norm. Also, the matrix $\mathbf{B}$ is a traditional preconditioner that deals with the extremes of the spectrum. In IDR(s), the space span($\mathbf{U}_k$) is not related to any *specific* components of the spectrum of $\mathbf{A}$. What is important is that s new eigenvalues of $\mathbf{P}_k\mathbf{A}$ are deflated in each cycle. In this context, the ideal role of the preconditioner $\mathbf{B}$ is less clear.

In the multigrid context (for elliptic problems), the matrix $\mathbf{B}^{-1}$ should act as a smoothing (or relaxation) step that eliminates the high–frequency errors. Interestingly, in IDR(s) the operator $\mathbf{C}_2 = \omega_{k+1}\mathbf{B}^{-1}$ acts similarly, “smoothing” the new primary residual $\mathbf{r}_{k+1}$ in norm.

During the IDR(s) iteration process, new spectral components of $\mathbf{P}_k\mathbf{A}$ are continuously



(a) Cycle 0.

(b) Cycle 1.

(c) Cycle 2.

(d) Cycle 3 (converged).

Figure 5.3: $\mathbf{B} = \mathbf{I}, n = 20, s = 5$, four cycles in total, $\mu_k \in \{0\} \cup \{1\}$ for all k.

projected out of the residual, while retaining spectral information from all previous cycles.

Note that if we take $s = n$, then $\sigma(\Pi_0) = \sigma(\Pi_0\mathbf{A}) = \{0\}$ and $\sigma(\mathbf{P}_0\mathbf{A}) = \{1\}$, which means that the iteration terminates within a single cycle.

5.4 Explicitly deflated IDR(s)

Similar to other Krylov subspace methods, IDR(s) methods can be explicitly preconditioned with deflation methods. In Section 5.4.1 a comparison is made between two variants of explicitly deflated IDR(s) and IDR(s) where the IDR deflation matrics are augmented with traditional deflation vectors. These results suggest possible “good” choices of $\widetilde{\mathbf{R}}$ and ω_k, which are discussed in Section 5.4.2. Numerical experiments that illustrate the theoretical results are given in Section 5.4.3.

5.4.1 Deflation vs. augmentation

Deflation

In standard deflation methods, the deflation matrices are defined as follows (cf. Definition 5.5 and Section 3.8 on page 62)

$$\Pi^{\text{def}} \equiv \mathbf{I} - \mathbf{A}\mathbf{Q}^{\text{def}} \quad \text{and} \quad \widehat{\Pi}^{\text{def}} \equiv \mathbf{I} - \mathbf{Q}^{\text{def}}\mathbf{A} \quad \text{where} \quad \mathbf{Q}^{\text{def}} \equiv \mathbf{Z}(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}\mathbf{Z}^* \tag{5.33}$$

where $\mathbf{Z} \in \mathbb{C}^{n\times t}$ is a deflation–subspace matrix of full rank and $\mathbf{E}^{\text{def}} \equiv \mathbf{Z}^*\mathbf{A}\mathbf{Z}$ is assumed to be invertible. Note that we have (cf. Lemma 5.6)

$$\Pi^{\text{def}}\mathbf{A}\mathbf{Z} = \mathbf{0}_{n,t} \quad \text{and} \quad \mathbf{Q}^{\text{def}}\mathbf{A}\mathbf{Z} = \mathbf{Z}. \tag{5.34}$$

To distinguish the standard deflation projection from the IDR projection, a superscript is added. Using this notation, the IDR(s) operator is written as (cf. (5.16))

$$\mathbf{P}_k^{\text{idr}} = \omega_{k+1}\Pi_k^{\text{idr}} + \mathbf{Q}_k^{\text{idr}}, \tag{5.35}$$

where Π_k^{idr} and $\mathbf{Q}_k^{\text{idr}}$ are the same as Π_k and $\mathbf{Q}_k$ in Prop 5.8.

Two variants of so–called "explicitly deflated" IDR(s) will be considered. The first one is based on the DEF1 variant [127, Section 2.3.2] where IDR(s) is used to solve the deflated system

$$\Pi^{\text{def}}\mathbf{A}\mathbf{x}' = \Pi^{\text{def}}\mathbf{b}. \tag{5.36}$$

The solution $\mathbf{x}'$ to the system (5.36) is related to the solution $\mathbf{x}$ of the original system $\mathbf{A}\mathbf{x} = \mathbf{b}$ as follows:

$$\mathbf{x} = \mathbf{Q}^{\text{def}}\mathbf{r}_0 + \widehat{\Pi}^{\text{def}}\mathbf{x}', \tag{5.37}$$

where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$. Also, note that the system (5.36) is singular, since $\Pi^{\text{def}}\mathbf{A}\mathbf{Z} = \mathbf{0}_{n,t}$. A singular system can still be solved as long as it is consistent, i.e., $\mathbf{b} \in \text{span}(\mathbf{A})$, see [82]. This is true for our case, since the same projection is applied to both sides of (5.36).

The second deflation method is based on the A–DEF1 variant [127, Section 2.3.3], where IDR(s) is applied to the deflated system

$$\mathbf{P}^{\text{adef1}}\mathbf{A}\mathbf{x} = \mathbf{P}^{\text{adef1}}\mathbf{b}, \tag{5.38}$$

where $\mathbf{P}^{\text{adef1}} = \Pi^{\text{def}} + \mathbf{Q}^{\text{def}}$ (cf. Section 5.2.2 and also Section 3.8 on page 62). The only difference between A–DEF1 and DEF1 is that the zero eigenvalues of $\Pi^{\text{def}}\mathbf{A}$ are shifted to one in $\mathbf{P}^{\text{adef1}}\mathbf{A}$.

The main motivation behind using an adapted deflation method such as A–DEF1 instead of DEF1 is as follows. It is known that perturbations (roundoff errors, perturbed starting vectors, inaccurate preconditioning solves, inaccurate Galerkin solves) can transform the zero eigenvalues of $\Pi^{\text{def}}\mathbf{A}$ into near–zero eigenvalues, making them potentially harmful to the convergence process. In an adapted deflation method, the corresponding near–unit eigenvalues are harmless.

Augmentation

We have shown that IDR(s) itself can be seen as an adapted deflation method. Therefore, a natural way to combine deflation–type preconditioners with IDR(s) is to augment the deflation subspace matrices $\mathbf{U}$ and $\widetilde{\mathbf{R}}$ with the deflation subspace matrix $\mathbf{Z}$ as follows:

$$\underline{\mathbf{U}}_k \equiv [\mathbf{U}_k \;\; \mathbf{Z}] \quad \text{and} \quad \underline{\widetilde{\mathbf{R}}} \equiv [\widetilde{\mathbf{R}} \;\; \mathbf{Z}]. \tag{5.39}$$

The IDR(s) deflation matrices are then

$$\Pi_k^{\mathrm{idr}'} \equiv \mathbf{I} - \mathbf{A}\mathbf{Q}_k^{\mathrm{idr}'}, \quad \text{where} \quad \mathbf{Q}_k^{\mathrm{idr}'} \equiv \underline{\mathbf{U}}_k \underline{\mathbf{E}}_k^{-1} \underline{\widetilde{\mathbf{R}}}^* \quad \text{and} \quad \underline{\mathbf{E}}_k \equiv \underline{\widetilde{\mathbf{R}}}^* \mathbf{A} \underline{\mathbf{U}}_k, \tag{5.40}$$

with dimensions

$$|\underline{\mathbf{U}}_k| = |\underline{\widetilde{\mathbf{R}}}| = n \times (s+t) \quad \text{and} \quad |\underline{\mathbf{E}}_k| = (s+t) \times (s+t). \tag{5.41}$$

The corresponding augmented IDR(s) operator is

$$\mathbf{P}_k^{\mathrm{idr}'} = \omega_{k+1} \Pi_k^{\mathrm{idr}'} + \mathbf{Q}_k^{\mathrm{idr}'}. \tag{5.42}$$

We call this approach "augmented IDR(s)".

Comparisons

Since IDR(s) is analogous to the A–DEF1 method, we will compare augmented IDR(s) to IDR(s) explicitly deflated with A–DEF1. For completeness, comparisons are also made with the DEF1 method.

Unless stated otherwise, no assumptions are made on the columns of $\mathbf{Z}$ or $\widetilde{\mathbf{R}}$. However, in some cases we make one of the following assumptions:

Assumption 5.23. The matrix $\widetilde{\mathbf{R}}$ is orthogonal to the matrix $\mathbf{Z}$.

Assumption 5.24. The matrix $\widetilde{\mathbf{R}}$ is orthogonal to the matrix $\mathbf{AZ}$.

In the following, let (cf. Theorem 5.17)

$$\Lambda_k \equiv \{\lambda \mid P_k(\lambda) = 0\} \cup \{\lambda \mid \det(\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda \mathbf{I})^{-1}\mathbf{W}) = 0\} \tag{5.43}$$

Proposition 5.25. *We distinguish between three deflation–type iterative processes of IDR(s):*

(i) Augmented IDR(s): We have for arbitrary $\mathbf{Z}$

$$\sigma\left(\Pi^{\mathrm{idr}'}\mathbf{A}\right) = \{0\} \cup \Lambda_k \tag{5.44}$$

where the zero eigenvalue has geometric multiplicity $s+t$ *and*

$$\sigma\left(\left[\Pi^{\mathrm{idr}'} + \mathbf{Q}^{\mathrm{idr}'}\right]\mathbf{A}\right) = \{1\} \cup \Lambda_k \tag{5.45}$$

where the unit eigenvalue also has geometric multiplicity $s+t$

(ii) IDR(s)-DEF1: Let $\overline{\mathbf{A}}_1 = \Pi^{\mathrm{def}}\mathbf{A}$*. We have for arbitrary* $\mathbf{Z}$

$$\sigma\left(\overline{\Pi}^{\mathrm{idr}}\overline{\mathbf{A}}_1\right) \equiv \sigma\left((\mathbf{I} - \overline{\mathbf{A}}_1\mathbf{U}(\widetilde{\mathbf{R}}^*\overline{\mathbf{A}}_1\mathbf{U})^{-1}\widetilde{\mathbf{R}}^*)\overline{\mathbf{A}}_1\right) = \{0\} \cup \Lambda_k \tag{5.46}$$

where the zero eigenvalue has geometric multiplicity $s+t$ *and*

$$\sigma\left(\left[\overline{\Pi}^{\mathrm{idr}} + \overline{\mathbf{Q}}^{\mathrm{idr}}\right]\overline{\mathbf{A}}_1\right) = \{0\} \cup \{1\} \cup \Lambda_k \tag{5.47}$$

where the zero eigenvalue has geometric multiplicity t *and the unit eigenvalue has geometric multiplicity* s*. The overlines of the operators signify the fact that they employ vectors based on the projected system.*

(iii) IDR(s)-ADEF1: Let $\overline{\mathbf{A}}_2 = (\Pi^{\mathrm{def}} + \mathbf{Q}^{\mathrm{def}})\mathbf{A}$. *Given Assumption 5.23, we have*

$$\sigma\left(\overline{\Pi}^{\mathrm{idr}}\overline{\mathbf{A}}_2\right) = \{0\} \cup \{1\} \cup \Lambda_k \tag{5.48}$$

where the zero eigenvalue has geometric multiplicity s and the unit eigenvalue has geometric multiplicity t. Also, we have

$$\sigma\left(\left[\overline{\Pi}^{\mathrm{idr}} + \overline{\mathbf{Q}}^{\mathrm{idr}}\right]\overline{\mathbf{A}}_2\right) = \{1\} \cup \Lambda_k \tag{5.49}$$

where the unit eigenvalue has geometric multiplicity $s + t$.

Proof. We have using Theorem 5.16:

(i) Augmented IDR(s): For (5.44) we have

$$\Pi^{\mathrm{idr}'}\mathbf{A}\underline{\mathbf{U}} = \mathbf{0}_{n,s+t} \tag{5.50}$$

and for (5.45) we have

$$\mathbf{Q}^{\mathrm{idr}'}\mathbf{A}\underline{\mathbf{U}} = \underline{\mathbf{U}}_{n,s+t} \tag{5.51}$$

(ii) IDR(s)-DEF1: For (5.46), we have

$$\overline{\mathbf{A}}_1\mathbf{Z} = \mathbf{0}_{n,t} \tag{5.52}$$

$$\overline{\Pi}^{\mathrm{idr}}\overline{\mathbf{A}}_1\mathbf{U} = \mathbf{0}_{n,s} \tag{5.53}$$

and for (5.47) we have

$$\overline{\mathbf{Q}}^{\mathrm{idr}}\overline{\mathbf{A}}_1\mathbf{U} = \mathbf{U}(\widetilde{\mathbf{R}}^*\Pi^{\mathrm{def}}\mathbf{A}\mathbf{U})^{-1}\widetilde{\mathbf{R}}^*)\Pi^{\mathrm{def}}\mathbf{A}\mathbf{U} \tag{5.54}$$

$$= \mathbf{U}_{n,s} \tag{5.55}$$

(iii) IDR(s)-ADEF1: Assumption 5.23 implies that $\widetilde{\mathbf{R}}^*\mathbf{Z} = \mathbf{0}_{s,t}$, so for (5.48) we have

$$\overline{\Pi}^{\mathrm{idr}}\overline{\mathbf{A}}_2\mathbf{U} = \mathbf{0}_{n,s} \tag{5.56}$$

$$\overline{\Pi}^{\mathrm{idr}}\overline{\mathbf{A}}_2\mathbf{Z} = \overline{\Pi}^{\mathrm{idr}}(\Pi^{\mathrm{def}} + \mathbf{Q}^{\mathrm{def}})\mathbf{A}\mathbf{Z} \tag{5.57}$$

$$= (\mathbf{I} - \overline{\mathbf{A}}_2\mathbf{U}(\widetilde{\mathbf{R}}^*\overline{\mathbf{A}}_2\mathbf{U})^{-1}\widetilde{\mathbf{R}}^*)\mathbf{Z} \tag{5.58}$$

$$= \mathbf{Z}_{n,t} \tag{5.59}$$

For (5.49) we have

$$\overline{\mathbf{Q}}^{\mathrm{idr}}\overline{\mathbf{A}}_2\mathbf{U} = \mathbf{U}(\widetilde{\mathbf{R}}^*(\Pi^{\mathrm{def}} + \mathbf{Q}^{\mathrm{def}})\mathbf{A}\mathbf{U})^{-1}\widetilde{\mathbf{R}}^*)(\Pi^{\mathrm{def}} + \mathbf{Q}^{\mathrm{def}})\mathbf{A}\mathbf{U} \tag{5.60}$$

$$= \mathbf{U}_{n,s} \tag{5.61}$$

This concludes the proof. □

Proposition 5.25 implies that in the generic case and in exact arithmetic all three variants compute the exact solution in at most $\frac{n-t}{s}$ IDR cycles.

In iterative process (iii), IDR(s) is applied to a system with $n - t + 1$ distinct eigenvalues. If we do not make Assumption 5.23, the exact solution for this case is computed in at most $\lceil \frac{n-t+1}{s} \rceil$ cycles. Another possibility is to use $t + 1$ deflation vectors for $\mathbf{Z}$.

Ideally, the deflation vectors in $\mathbf{Z}$ approximate the eigenspace corresponding to the unfavourable eigenvalues of $\mathbf{A}$, e.g., eigenvalues small in magnitude. Depending on the deflation technique, these eigenvalues will be shifted to zero or one, removing them from the IDR(s) iteration process. In this way, techniques from domain decomposition and deflation can be easily used in IDR.

Note that augmenting the matrices $\mathbf{U}$ and $\widetilde{\mathbf{R}}$ with deflation vectors is different from increasing s in "standard" IDR(s). Also, if in IDR(s) we would set $\mathbf{U} = \widetilde{\mathbf{R}} = \mathbf{Z}$, we obtain a standard Richardson iteration deflated with A–DEF1.

For most applications we have $t \gg s$, which would make the augmented Galerkin matrix too big to solve directly. To avoid this problem, note that in augmented IDR(s) the augmented Galerkin matrix $\underline{\mathbf{E}}_k$ has the following form:

$$\underline{\mathbf{E}}_k = \begin{bmatrix} \widetilde{\mathbf{R}}^* \\ \mathbf{Z}^* \end{bmatrix} \mathbf{A} \begin{bmatrix} \mathbf{U}_k & \mathbf{Z} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{U}_k & \widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{Z} \\ \mathbf{Z}^*\mathbf{A}\mathbf{U}_k & \mathbf{Z}^*\mathbf{A}\mathbf{Z} \end{bmatrix} = \begin{bmatrix} \mathbf{E}_{11} & \mathbf{E}_{12} \\ \mathbf{E}_{21} & \mathbf{E}_{22} \end{bmatrix} = \left\{ \begin{matrix} |s \times s| & |s \times t| \\ |t \times s| & |t \times t| \end{matrix} \right\}. \tag{5.62}$$

If $\mathbf{Z}$ consists of subdomain deflation vectors, then $\mathbf{Z}^*\mathbf{A}\mathbf{Z}$ is a diagonal band matrix. To compute $\mathbf{x} = \underline{\mathbf{E}}_k^{-1}\mathbf{b}$ for some $\mathbf{b}$, one can make use of the Schur complement as follows

$$(\mathbf{E}_{11} - \mathbf{E}_{12}\mathbf{E}_{22}^{-1}\mathbf{E}_{21})\mathbf{x}_1 = \mathbf{b}_1 - \mathbf{E}_{12}\mathbf{E}_{22}^{-1}\mathbf{b}_2 \qquad |\mathbf{x}_1| = |\mathbf{b}_1| = s \tag{5.63}$$

$$\mathbf{E}_{22}\mathbf{x}_2 = \mathbf{b}_2 - \mathbf{E}_{21}\mathbf{x}_1 \qquad |\mathbf{x}_2| = |\mathbf{b}_2| = t \tag{5.64}$$

Note that there are *three* instances where we have to solve systems involving $\mathbf{E}_{22}$. Also, we have

$$\mathbf{E}_{11} - \mathbf{E}_{12}\mathbf{E}_{22}^{-1}\mathbf{E}_{21} = \widetilde{\mathbf{R}}^*\mathbf{A}(\mathbf{I} - \mathbf{Z}(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}\mathbf{Z}^*\mathbf{A})\mathbf{U}_k \tag{5.65}$$

$$= \widetilde{\mathbf{R}}^*\mathbf{A}\widehat{\Pi}^{\text{def}}\mathbf{U}_k \tag{5.66}$$

$$= \widetilde{\mathbf{R}}^*\Pi^{\text{def}}\mathbf{A}\mathbf{U}_k \tag{5.67}$$

which is exactly the (deflated) Galerkin system of IDR(s)-DEF1.

Let $\mathbf{b}_1 = \widetilde{\mathbf{R}}^*\mathbf{A}$ and $\mathbf{b}_2 = \mathbf{Z}^*\mathbf{A}$. Then we have (omitting the subscript k)

$$\Pi^{\text{idr}'}\mathbf{A} = (\mathbf{I} - \mathbf{A}\underline{\mathbf{U}}\,\underline{\mathbf{E}}^{-1}\underline{\widetilde{\mathbf{R}}}^*)\mathbf{A} \tag{5.68}$$

$$= \mathbf{A} - \mathbf{A}(\mathbf{U}\mathbf{x}_1 + \mathbf{Z}\mathbf{x}_2) \tag{5.69}$$

$$= (\mathbf{I} - \Pi^{\text{def}}\mathbf{A}\mathbf{U}(\widetilde{\mathbf{R}}^*\Pi^{\text{def}}\mathbf{A}\mathbf{U})^{-1}\widetilde{\mathbf{R}}^*)(\mathbf{I} - \mathbf{A}\mathbf{Z}(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}\mathbf{Z}^*)\mathbf{A} \tag{5.70}$$

$$= \overline{\Pi}^{\text{idr}}\overline{\mathbf{A}}_1 \tag{5.71}$$

This implies that the spectrum of $\Pi^{\text{idr}'}\mathbf{A}$ is the same as the spectrum of $\overline{\Pi}^{\text{idr}}\overline{\mathbf{A}}_1$, which is in accordance with Proposition 5.25, i.e., properties (5.44) and (5.46). However, it is expected that in practical applications the IDR(s) processes will behave differently, since in augmented IDR(s) $s + t$ eigenvalues of the active systems are shifted to one, while in IDR(s)-DEF1 t eigenvalues remain zero.

That is, perturbations (roundoff errors, perturbed starting vectors, inaccurate preconditioning solves, inaccurate Galerkin solves) can transform the zero eigenvalues of the active systems in IDR(s)-DEF1 into near–zero eigenvalues, which may result in numerical instabilities. This suggests that augmented IDR(s) will be numerically more stable than IDR(s)-DEF1. A similar argument can be used to show that IDR(s)-ADEF1 is numerically more stable than IDR(s)-DEF1.

Some of the advantages of using augmented IDR(s) as opposed to IDR(s)-DEF1 or IDR(s)-ADEF1 are:

Algorithm 5.4 Computation of $\Pi^{\text{idr}'}\mathbf{y}$

1: $\mathbf{b}_1 = \widetilde{\mathbf{R}}^*\mathbf{y}$
2: $\mathbf{b}_2 = \mathbf{Z}^*\mathbf{y}$
3: **Solve** $(\mathbf{Z}^*\mathbf{A}\mathbf{Z})\mathbf{b}_3 = \mathbf{b}_2$
4: $\mathbf{b}_4 = \widetilde{\mathbf{R}}^*(\mathbf{A}\mathbf{Z})\mathbf{b}_3$
5: $\mathbf{b}_5 = \mathbf{b}_1 - \mathbf{b}_4$
6: **Solve** $(\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{U}_k - \widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{Z}(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}\mathbf{Z}^*\mathbf{A}\mathbf{U}_k)\mathbf{x}_1 = \mathbf{b}_5$
7: $\mathbf{b}_6 = \mathbf{Z}^*(\mathbf{A}\mathbf{U}_k)\mathbf{x}_1$
8: $\mathbf{b}_7 = \mathbf{b}_2 - \mathbf{b}_6$
9: **Solve** $(\mathbf{Z}^*\mathbf{A}\mathbf{Z})\mathbf{x}_2 = \mathbf{b}_7$
10: $\mathbf{y}_1 = \mathbf{A}\mathbf{U}_k\mathbf{x}_1$
11: $\mathbf{y}_2 = \mathbf{A}\mathbf{Z}\mathbf{x}_2$
12: $\Pi^{\text{idr}'}\mathbf{y} = \mathbf{y} - \mathbf{y}_1 - \mathbf{y}_2$

	theory		implementation		
method	$\Pi^{\text{def}}\mathbf{y}$	$\mathbf{Q}^{\text{def}}\mathbf{y}$	IP/MV	AXPY	GSS
IDR(s)-DEF1	1	0	2	1	1
IDR(s)-ADEF1	1	1	3	1	1
augmented IDR(s)	n/a	n/a	3	1	1

Table 5.1: Extra computational cost per MV compared to “standard” IDR(s).

- Possible increased numerical stability, since unfavourable eigenvalues are shifted to one instead of zero.
- It is a natural way of introducing deflation–type preconditioner into IDR(s) algorithms.
- Given a specific choice of $\widetilde{\mathbf{R}}$ (see Section 5.4.2), the two by two block augmented Galerkin system (5.62) can be inverted efficiently.
- Because IDR(s)-ADEF1 is applied to a system with $n - t + 1$ distinct eigenvalues, it converges in more cycles than augmented IDR(s) in exact arithmetic (i.e., $(n - t + 1)/s$ cycles instead of $(n - t)/s$) if no special assumptions are made.

To apply the operator $\Pi^{\text{idr}'}\mathbf{y}$ to some vector $\mathbf{y}$, we need to perform the steps shown in Algorithm 5.4. For efficiency, the following (small) matrices can be computed/factored and stored beforehand:

$$(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}, \quad \mathbf{A}\mathbf{Z}, \quad \text{and} \quad \widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{Z}(\mathbf{Z}^*\mathbf{A}\mathbf{Z})^{-1}. \tag{5.72}$$

Note that the matrices $\mathbf{Z}^*\mathbf{A}\mathbf{U}_k$ and $\widetilde{\mathbf{R}}^*\mathbf{A}\mathbf{U}_k$ have to be recomputed in each cycle k, but they can be reused within a cycle. The matrix $\mathbf{A}\mathbf{U}_k$ is readily available and does not require additional MVs.

In practical algorithms, the operation $\mathbf{Q}^{\text{idr}'}\mathbf{y}$ also has to be computed. This can be done efficiently by reusing quantities from $\Pi^{\text{idr}'}\mathbf{y}$, similar to the A-DEF1 method.

Algorithm 5.5 Computation of $\Pi^{\mathrm{idr}'}\mathbf{y}$ with special choice for $\widetilde{\mathbf{R}}$.

1: $\mathbf{b}_1 = \widetilde{\mathbf{R}}^*\mathbf{y}$
2: $\mathbf{b}_2 = \mathbf{Z}^*\mathbf{y}$ // *IP/MV #1*
3: **Solve** $(\widetilde{\mathbf{R}}^*\mathbf{AU}_k)\mathbf{x}_1 = \mathbf{b}_1$
4: $\mathbf{b}_3 = (\mathbf{Z}^*\mathbf{AU}_k)\mathbf{x}_1$
5: $\mathbf{b}_4 = \mathbf{b}_2 - \mathbf{b}_3$
6: **Solve** $(\mathbf{Z}^*\mathbf{AZ})\mathbf{x}_2 = \mathbf{b}_4$ // *GSS #1*
7: $\mathbf{y}_1 = (\mathbf{AU}_k)\mathbf{x}_1$
8: $\mathbf{y}_2 = (\mathbf{AZ})\mathbf{x}_2$ // *IP/MV #2*
9: $\Pi^{\mathrm{idr}'}\mathbf{y} = \mathbf{y} - \mathbf{y}_1 - \mathbf{y}_2$ // *AXPY #1*

5.4.2 Choosing $\widetilde{\mathrm{R}}$ and ω_k

Choice of $\widetilde{\mathbf{R}}$

Inspection of the Galerkin matrix $\underline{\mathbf{E}}_k$ from (5.62) belonging to augmented IDR(s) shows that the submatrices $\widetilde{\mathbf{R}}^*\mathbf{AU}_k$ and $\mathbf{Z}^*\mathbf{AU}_k$ have to be recomputed each cycle. The non–zero matrix $\mathbf{Z}^*\mathbf{AZ}$ is fixed. However, the matrix $\widetilde{\mathbf{R}}$ can be chosen such that $\widetilde{\mathbf{R}}^*\mathbf{AZ} = \mathbf{0}$ using an orthogonal projection such as (cf. Assumption 5.24):

$$\widetilde{\mathbf{R}}' = (\mathbf{I} - \mathbf{AZ}((\mathbf{AZ})^*\mathbf{AZ})^{-1}(\mathbf{AZ})^*)\widetilde{\mathbf{R}} \quad \Leftrightarrow \quad \widetilde{\mathbf{R}}' \perp \mathbf{AZ}. \tag{5.73}$$

The resulting Galerkin matrix is then lower block diagonal:

$$\underline{\mathbf{E}}_k = \begin{bmatrix} \mathbf{E}_{11} & \varnothing \\ \mathbf{E}_{21} & \mathbf{E}_{22} \end{bmatrix}, \tag{5.74}$$

which simplifies the computation of $\mathbf{x} = \underline{\mathbf{E}}_k^{-1}\mathbf{b}$. Using forward substition, we can compute

$$\mathbf{E}_{11}\mathbf{x}_1 = \mathbf{b}_1 \tag{5.75}$$
$$\mathbf{E}_{22}\mathbf{x}_2 = \mathbf{b}_2 - \mathbf{E}_{21}\mathbf{x}_1. \tag{5.76}$$

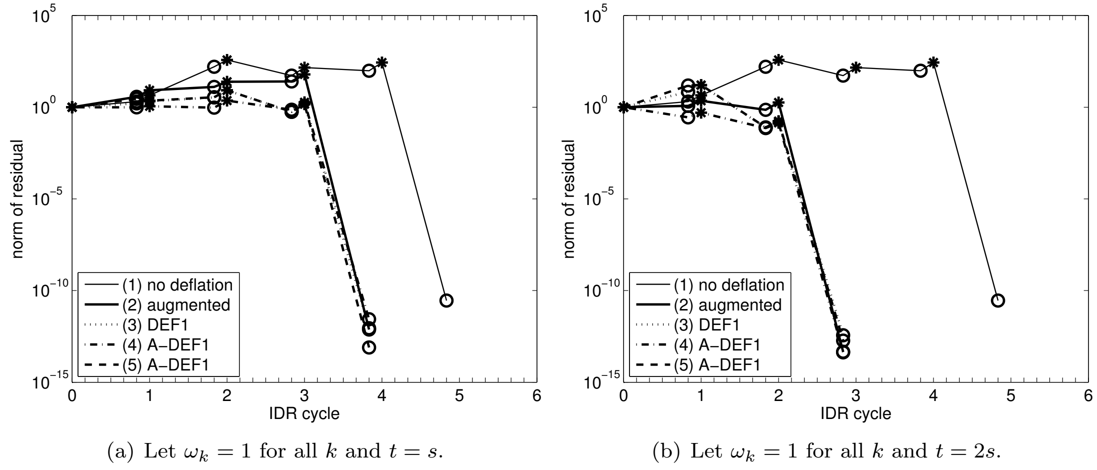
Note that $|\mathbf{x}_1| = s$ and that $|\mathbf{x}_2| = t$. The corresponding computation of $\Pi^{\mathrm{idr}'}\mathbf{y}$ using this choice of $\widetilde{\mathbf{R}}$ is given in Algorithm 5.5.

As before, for the computation of $\mathbf{Q}^{\mathrm{idr}'}\mathbf{y}$ quantities from $\Pi^{\mathrm{idr}'}\mathbf{y}$ can be reused. Also, the matrices $(\mathbf{Z}^*\mathbf{AZ})^{-1}$ and $\mathbf{AZ}$ can be computed and stored beforehand. The matrices $\mathbf{Z}^*\mathbf{AU}_k$ and $(\widetilde{\mathbf{R}}^*\mathbf{AU}_k)^{-1}$ can be computed at the start of cycle k and reused during a cycle. Also, the matrix $\mathbf{AU}_k$ is readily available and does not require additional MVs.

Table 5.1 lists the *additional* computational cost of the three deflation approaches compared to a non–deflated IDR(s) method. The term GSS denotes a "Galerkin System Solve" such as computing the solution to $(\mathbf{Z}^*\mathbf{AZ})\mathbf{x} = \mathbf{y}$ for some $\mathbf{y}$. Also, depending on the type of deflation vectors, an operation involving $\mathbf{Z}$ either counts as a MV or as an inner product (IP). Note that the third IP/MV for augmented IDR(s) is included in the application of $\mathbf{Q}^{\mathrm{idr}'}$.

Choice of μ_k

Generally speaking, the goal of a preconditioner is to cluster the spectrum of the preconditioned system around one (after an appropriate scaling). Choosing $\mu_k = 1$ for all k would be the



(a) Let $\omega_k = 1$ for all k and $t = s$.

(b) Let $\omega_k = 1$ for all k and $t = 2s$.

Figure 5.4: Residual norms for IDR(s), $n = 25, s = 5, \mathbf{B} = \mathbf{I}$, showing primary "$*$" and secondary "$\circ$" residuals.

most effective spectral choice, since in this case the spectrum of the active system will become increasingly more clustered around one. That is, we have (in exact arithmetic) for $k \in \mathbb{N}_0$:

$$\sigma(\mathbf{P}_k\mathbf{A}) = \{1\} \cup \{\lambda \mid \det(\widetilde{\mathbf{R}}^*(\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{W}) = 0\} \tag{5.77}$$

where the eigenvalue 1 has geometric multiplicity $(k+1)s$ and

$$\sigma(\mathbf{P}_m\mathbf{A}) = \{1\} \tag{5.78}$$

where $m = n/s - 1$ and where the eigenvalue 1 has geometric multiplicity n. However, if at the same time the remaining eigenvalues of an active system are located close to zero, convergence of the iteration process within that cycle may be hampered. Numerical experiments seem to indicate that even though this can happen in some particular cycle, this will generally not be the case in the next cycle.

Continuing this line of reasoning, it can be argued that the *value* of μ_k does not significantly affect the convergence process, in particular for large s. The key property of IDR(s) methods is that the spectrum of the active system becomes increasingly more *clustered*. The *location* of the clustered spectrum (i.e., the values of μ_k) is less important. Nevertheless, using a near–zero μ_k should be avoided, since it could result in a badly conditioned active system.

For smaller s, the clustering property is much less pronounced and choosing an appropriate value of μ_k will be more important. This effect is observed in many experiments, see for example [120, Section 6.4] and [119]. For a detailed mathematical analysis of the influence of the factors μ_k on the IDR iteration process, see [119].

5.4.3 Numerical examples

In the following experiments, the matrix $\mathbf{Z}$ consists of random and orthogonalised vectors. These experiments are for illustrative purposes only and the test problem is the 1D convection–diffusion problem from Section 5.3.3 with $n = 25$ and $s = 5$. We use the IDR method from Algorithm 5.1 and set $\omega_k = 1$ for all k. The finite convergence behaviour of the following five iteration processes will be compared:

(1) Non–deflated IDR(s) (i.e., $t = 0$).

(2) Augmented IDR(s) with Assumption 5.24 in order to efficiently invert the two by two block system (5.74), see Section 5.4.2.

(3) IDR(s)-DEF1.

(4) IDR(s)-ADEF1 with an additional deflation vector for $\mathbf{Z}$.

(5) IDR(s)-ADEF1 with Assumption 5.23.

Shown in Figure 5.4(a) and Figure 5.4(b) are the logarithms of the norms of the primary and secondary residuals using $t = s$ and $t = 2s$ deflation vectors for $\mathbf{Z}$, respectively. The ticks on the horizontal axis represent the number of MVs. Note that one IDR cycle consists of $s + 1$ MVs: computing a secondary residual $\mathbf{r}'_{k-1}$ ("$\circ$") involves s MVs, while the computing the next primary residual $\mathbf{r}_k$ ("$*$") uses one MV (see also Section 5.2.1). For iteration process (4), an additional deflation vector is used and the following observations can be made:

- Since the iteration matrices of the four iterative processes are different (and hence the "initiation" matrix $\mathbf{V}_0$), the residual norms are different.
- According to the theory, the four iteration processes (2)–(5) should converge within $(n - t)/s = (25 - t)/5$ cycles, i.e., within four and three cycles for $t = s$ and $t = 2s$, respectively. The non–deflated process converges within five cycles. This is in accordance with the numerical results.
- If we do not make Assumption 5.23, then iteration process (5) converges in at most $(n - t + 1)/s$ IDR cycles.

5.5 Conclusions

By interpreting IDR(s) as a deflation method, interesting properties of the IDR(s) method have been revealed. Firstly, this has lead to the IDR projection theorem, which shows that the spectrum of the deflated systems in IDR(s) become increasingly more clustered. This can be seen as an intuitive explanation for the excellent convergence properties of IDR(s).

Based on this interpretation, one cycle of IDR(s) can be seen as consisting of three key steps: constructing a unique primary residual, constructing the unique secondary residual, and constructing vectors for a basis of a very specific Krylov subspace.

It also shows that the IDR(s) method is an instantiation of a specific deflation method: a so–called adapted deflation method. The deflation subspace matrix in IDR(s) is updated in each cycle with new information while retaining information from all previous IDR(s) cycles.

Although this interpretation did not yet lead to insights into the effect of a varying preconditioner on the convergence of IDR(s), interesting properties on the structure of IDR(s) have been revealed.

Lastly, this interpretation allows for the efficient inclusion of standard deflation–type preconditioners into IDR(s) methods.

Chapter 6

General Conclusions and Outlook

Overview

In this chapter the main results presented in this thesis are summarised and related to the general field of heterogeneous parallel computing. In addition, the issues that were raised but not addressed in this thesis are listed and suggestions for further research are given.

6.1 Aim of research

In order to solve big "Grand Challenge"–type problems [77], it is absolutely necessary that computational resources from many different sources are combined, resulting in highly heterogeneous computing systems as found in Grid computing. The main objective of this research was to design efficient iterative methods for solving large sparse linear systems on such computing platforms. This difficult but important problem was attacked by first identifying the key theoretical properties of an ideal and iterative (possibly wide–area) algorithm for Grid computing, which are: *robust*, *efficient*, *coarse–grained*, *asynchronous*, *minimal synchronisation*, *resource–aware*, *adaptive*, and *fault–tolerant*. All of the algorithms presented in this thesis exhibit these features in one way or another.

To gain practical experience with solving this problem, a real–world case study was performed where we tried to solve sparse linear systems on heterogeneous computing systems using "conventional" techniques. Among other things, this lead to the conclusion that the key obstacle in designing efficient algorithms for heterogeneous computing systems is the high cost of global synchronisation. In order to alleviate this bottleneck, three main techniques were then considered:

- *Exploiting the hierarchical structure of multi–clusters.* Computational grids usually consist of several different clusters with fast intracluster communication and relatively slow intercluster wide-area communication. As an example, the freedom in choosing the IDR test matrix in IDR(s) was exploited in order to efficiently perform operations with this matrix on multi–clusters. In addition, the parallel performance model for computing an optimal s in IDR(s) also exploits the hierarchy in multi–clusters.

- *Using asynchronous communication to make the algorithm latency-tolerant.* With such techniques, computations can be overlapped with wide-area communication. This was achieved by using an asynchronous inner iterative method as a *preconditioner* in a flexible

outer iterative method, where the preconditioner is allowed to change in each outer iteration step.

- *Minimising the inner products that induce global synchronisation points as much as possible.* This was done by reformulating existing iterative methods and by spending most of the computational effort on the preconditioning iteration.

Although these optimisations are well–known in the literature, they had not been applied and combined in the manner as done in this thesis. By using these three techniques it is shown in this thesis that it is possible to iteratively and efficiently solve large sparse linear systems on heterogeneous computing systems. It can therefore be concluded that the main goal of the research has been achieved.

6.2 Research challenges and principal findings

The fact that an asynchronous iterative method was used for the preconditioning operation on heterogeneous computing hardware means that it is an unpredictable and random operation. As a result, *variable* preconditioning in flexible iterative methods became an important research issue. For symmetric systems, variable preconditioning had been investigated extensively for the flexible CG method by many other people so these issues were not too difficult to overcome. For nonsymmetric systems the flexible GCR/GMRES–type methods were used, which had also been investigated in the past.

Since IDR(s) was a relatively new method for solving nonsymmetric systems, it made sense to investigate IDR(s) in the broad context of heterogeneous computing. This turned the direction of the research process to IDR(s) methods and we found that the freedom one has in deriving algorithmic variants of IDR(s) made it extremely suitable for constructing efficient IDR(s) algorithms for parallel and Grid computers. Combining asynchronous preconditioning with IDR(s) was the next logical step, which introduced the problem of using IDR(s) as a flexible method.

The main motivation behind interpreting IDR(s) as a deflation method was as follows. IDR(s) was a new method and its convergence behaviour was not yet completely understood. In particular, it was hoped that this interpretation would result in true flexible variants of IDR(s). Although this interpretation led to key insights into the structure of IDR(s) algorithms, it is not yet clear how these insights can be used to analyse the effect of a varying preconditioner on the convergence behaviour of IDR(s).

Improving the effectiveness of the asynchronous preconditioner was another big challenge. Complementing the asynchronous iterative method with a coarse–grid correction had been less successful than we initially hoped. It is interesting to further investigate the exact causes and to look also at other solutions, such as using overlap in the asynchronous iteration.

Efficient resource–aware partitioning was also a difficult issue and this was briefly touched on at the beginning of the research. In theory, the flexibility of the GridSolve middleware makes dynamic resource–aware repartitioning of the work in each outer iteration step possible. However, the relatively large communication overhead in GridSolve hindered the practical applicability of such an approach. Future versions of GridSolve or other middleware that are similar may have less overhead, which could make dynamic repartitioning more effective.

The original motivation for this research was to solve moving boundary problems using the Immersed Boundary Method, in which structured meshes are employed. That is why the main focus in this thesis is on solving problems on Cartesian meshes, which results in coefficient matrices that are sparse and regular. Using the techniques and experience presented in this thesis as building blocks, problems on unstructured meshes can also be solved.

Generally speaking, the research presented in each chapter of this thesis exhibits a fairly even balance between theory and practice. Towards the end of the research, the focus shifted to being primarily theoretical due to the interpretation of IDR(s) as a deflation method.

This thesis shows that using the partially asynchronous algorithm is more efficient than using either a fully synchronous method or a fully asynchronous method. Considering the less than successful past attempts in using asynchronous iterative method to iterative solve sparse linear system on heterogeneous computing systems, it was somewhat surprising that they could be used in the way as done in this thesis. Also rather surprisingly, we found that the asynchronous preconditioner remains highly effective in a computational environment where network load varies heavily. This once again proves that finding an efficient parallel preconditioner is crucial for solving large computational problems. This thesis shows that asynchronous methods — whose study dates back as far as 1969 — can be used in new and different ways. Also, of all the three techniques mentioned in the beginning for alleviating the synchronisation bottleneck, we found that using an asynchronous iterative method as a preconditioner bore the most fruit.

The experiences obtained in this research re-enforces the general opinion that the future of parallel scientific computing truly lies in *hybrid* algorithmic approaches. That is, the most effective approach for our case was not purely asynchronous or purely synchronous: it was semi–asynchronous. In other words, hybrid/heterogeneous computing requires hybrid/heterogeneous algorithms. Such heterogeneous computing systems are becoming more and more common, whether it is a local cluster of different machines, a multi–cluster, a cluster of GPUs, a multi–core system, a traditional supercomputer, or some combination of all of the above.

Although the algorithms presented in this thesis exhibit the properties of the ideal algorithm for Grid computing listed in the beginning in some way or another, the main goal now is to design algorithms that exhibit *all* of these features. The results presented in this thesis provide a strong foundation to build on in order to achieve that goal.

6.3 Broader implications

The short–term implications of this research are that by using *existing* computational resources more effectively, larger, more accurate, and more efficient numerical simulations can be performed. In the long–term, the techniques in this thesis can be used to construct efficient algorithms for GPU–based computing platforms, resulting in both environmentally friendly and cheaper simulations. As a concrete example of such a platform, the Little GREEN Machine [148] is a Beowulf cluster with a GPU in each node and will be part of the DAS–4 project, the next generation of the DAS–3 multi–cluster. The experiences with designing and implementing iterative algorithms on the DAS–3 can be used with the DAS–4.

According to the TOP500 list [91], the fastest known supercomputer as of October 2010 is the Tianhe–1A machine located at the National Supercomputing Center in Tianjin, China, which combines general purpose GPUs and multi–core CPUs in order to obtain a peak computing rate of 2.5 petaFLOPS. Even with the high–speed interconnect used by such machines, synchronisation cost will always be a bottleneck. In order to reach the claimed peak computing rates for practical applications, techniques such as presented in this thesis can help.

Another issue that is becoming more and more critical in high performance computing is energy consumption. In addition to the "traditional" TOP500 list that measures supercomputers' performance in terms of floating-point operations per second (FLOPS), the Green500 list [63] aims to raise awareness to other equally important performance metrics, such as performance per watt. To quote the description of the Green500 list: "to ensure that supercomputers are only simulating climate change and not creating climate change." From the beginning, GPU computing has been advertised as being eco–friendly and with the top eight places of the Green500

list all being GPU accelerated supercomputers as of June 2010, this claim seems to be valid.

The point is that heterogeneous computing platforms are widespread in high performance computing and occur at many different scales. For example, the presented techniques can be used for wide–area systems and local nondedicated systems as done in this thesis, for smaller architectures such as single node GPU computing and multi–core computing, and for dedicated accelerator–based supercomputers such as the Tianhe–1A.

Curriculum Vitæ

- March 2011: Metrology Software Design Engineer at ASML in Veldhoven
- November 2006 – November 2010: PhD researcher in Numerical Analysis, Department of Applied Mathematical Analysis, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, The Netherlands. Financially supported by the Delft Centre for Computational Science and Engineering. Title of dissertation: *Efficient Iterative Solution of Large Linear Systems on Heterogeneous Computing Systems.* Advisers: Prof.dr.ir. C. Vuik and dr.ir. M. B. van Gijzen
- May 2010: Visiting researcher, department of Mathematical Information Science, Tokyo University of Science, Tokyo, Japan, host: prof. Emiko Ishiwata
- April 2010: Visiting researcher, faculty of Economics and Information, Gifu Shotoku University, Gifu, Japan, host: prof. Kuniyoshi Abe
- September 2006: M.Sc. in Scientific Computing, Utrecht University
- September 2005: B.Sc. in Mathematics, Utrecht University
- September 1991 – September 1997: Stedelijk Gymnasium, Haarlem
- Born on April 1, 1979 in Haarlem, the Netherlands

Scientific Résumé

Publications

Refereed journal papers

- T. P. Collignon, G. L. G. Sleijpen, and M. B. van Gijzen. Interpreting IDR(s) as a deflation method. *Journal of Computational and Applied Mathematics*, 2011. Special Issue: Proceedings ICCAM–2010 (submitted).

- T. P. Collignon and M. B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 2011. (published online: 14 january 2011).

- T. P. Collignon and M. B. van Gijzen. Fast iterative solution of large sparse linear systems on geographically separated clusters. *International Journal of High Performance Computing Applications*, 2011. (to appear).

- T. P. Collignon and M. B. van Gijzen. Two implementations of the preconditioned Conjugate Gradient method on heterogeneous computing grids. *International Journal of Applied Mathematics and Computer Science*, 20(1):109–121, 2010.

Book chapter

- T. P. Collignon and M. B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. In B. Koren and C. Vuik, editors, *Advanced Computational Methods in Science and Engineering. Springer Series Lecture Notes in Computational Science and Engineering*, volume 71, pages 79–106. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

Conference proceedings

- T. P. Collignon and M. B. van Gijzen. Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. In G. Kreiss, P. Lötstedt, A. Målqvist, and M. Neytcheva, editors, *Numerical Mathematics and Advanced Applications 2009: Proceedings of ENUMATH 2009, the 8th European Conference on Numerical Mathematics and Advanced Applications, Uppsala, July 2009*, pages 261–268. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

- M. B. van Gijzen and T. P. Collignon. Exploiting the flexibility of IDR(s) for Grid computing. In *The Proceedings of the 2nd Kyoto–Forum on Krylov Subspace Methods, Kyoto University, Kyoto, Japan*, March 2010.

- T. P. Collignon and M. B. van Gijzen. 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*, pages 527–540, 2007.

Technical reports

- T. P. Collignon, G. L. G. Sleijpen, and M. B. van Gijzen. Interpreting IDR(s) as a deflation method. Technical report, Delft University of Technology, Delft, The Netherlands, 2010. DUT report 10–21.
- M. B. van Gijzen and T. P. Collignon. Exploiting the flexibility of IDR(s) for Grid computing. Technical report, Delft University of Technology, Delft, The Netherlands, 2010. DUT report 10–11.
- T. P. Collignon and M. 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.
- T. P. Collignon and M. 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.
- T. P. Collignon and M. B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. Technical report, Delft University of Technology, Delft, The Netherlands, 2008. DUT report 08–12.
- T. P. Collignon and M. 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.
- T. P. Collignon and M. B. van Gijzen. Implementing the Conjugate Gradient method on a grid computer. Technical report, Delft University of Technology, Delft, The Netherlands, 2007. DUT report 07–11.

Other

- T. P. Collignon. Fast iterative solution of large linear systems on Grid computers. *MaC-Hazine: W.I.S.V. "Christiaan Huygens"*, 14(3):33–34, March 2010

Presentations

Talks

- "Efficient Iterative Solution of Large Linear Systems on Geographically Separated Clusters", *Shell*, the Hague, the Netherlands, August 25, 2010.
- "IDR(s) for Grid Computing", *ICCAM 2010: 15th International Congress on Computational and Applied Mathematics*, Leuven, Belgium, July 8, 2010.
- "IDR(s) for Grid Computing", *Ishiwata group talk*, Tokyo, Japan, April 27, 2010.

- "Fast solution of nonsymmetric linear systems on Grid computers using IDR(s) methods", *NW group talk*, Delft, The Netherlands, February 5, 2010.
- "Solving Large Linear Systems on Grid Computers using an Asynchronous Method as Preconditioner", *ENUMATH 2009: European Conference on Numerical Mathematics and Advanced Applications*, Uppsala, Sweden, June 29, 2009.
- "Solving Large Linear Systems on Grid Computers using an Asynchronous Method as Preconditioner", *VECPAR 2008: 8th International Meeting on High Performance Computing for Computational Science*, Toulouse, France, June 26, 2008.
- "Efficient Iterative Solution of Large Sparse Linear Systems on a Cluster of Geographically Separated Clusters", *PMAA 2008: 5th International Workshop on Parallel Matrix Algorithms and Applications*, Neuchâtel, Switzerland, June 22, 2008.
- "Immersed Boundary Methods on Heterogeneous Networks of Computers", *NW group talk*, Delft, The Netherlands, May 21, 2007.

Posters

- "Interpreting IDR(s) as a deflation method", *The Thirty–fifth Woudschoten Conference*, Zeist, the Netherlands, October 6–8, 2010.
- "Parallelisation of IDR(s) on Cluster and Grid Computers", *The Thirty–fourth Woudschoten Conference*, Zeist, the Netherlands, October 7–9, 2009.
- "Efficient Iterative Solution of Large Sparse Linear Systems on a Cluster of Geographically Separated Clusters", *The Thirty–third Woudschoten Conference*, Zeist, the Netherlands, October 8–10, 2008.
- "Numerical Analysis", *DCSE Customer Day 2008*, Delft, The Netherlands, September 19, 2008.
- "Solving Large Sparse Linear Systems on Grid Computers using an Asynchronous Iterative Method as a Preconditioner", *JMBC Burgersdag 2008*, Delft, The Netherlands, January 10, 2008.
- "Implementing the Conjugate Gradient Method on a Grid computer", *CANA 2007: Computer Aspects of Numerical Algorithms*, Wisla, Poland, October 16, 2007.
- "Implementing the Conjugate Gradient Method on a Grid computer", *The Thirty–second Woudschoten Conference*, Zeist, the Netherlands, October 3–5, 2007.
- "Design and analysis of novel numerical algorithms for heterogeneous computing platforms", *DCSE Symposium 2007*, Delft, The Netherlands, March 30, 2007.

Conferences

- ICCAM 2010: 15th International Congress on Computational and Applied Mathematics, Leuven, Belgium, July 5–9, 2010.
- ENUMATH 2009: European Conference on Numerical Mathematics and Advanced Applications, Uppsala, Sweden, June 29 – July 3, 2009.

- VECPAR 2008: 8th International Meeting on High Performance Computing for Computational Science, Toulouse, France, June 24–27, 2008.
- PMAA 2008: 5th International Workshop on Parallel Matrix Algorithms and Applications, Neuchâtel, Switzerland, June 20–22, 2008.
- CANA 2007: Computer Aspects of Numerical Algorithms, Wisla, Poland, October 15–17, 2007.
- Annual Woudschoten Conferences of the Dutch & Flemish Numerical Analysis Communities, Zeist, The Netherlands, 2007–2010.

Bibliography

[1] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-threading*. Intel Press, Santa Clara, CA, USA, 2006.

[2] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high–performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, pages 230–256. Springer–Verlag, Berlin/Heidelberg, Germany, 2005.

[3] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. V. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The grid application toolkit: Towards generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93(3):534–550, March 2005.

[4] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

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

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

[7] J. Baglama, D. Calvetti, G. H. Golub, and L. Reichel. Adaptively preconditioned GMRES algorithms. *SIAM Journal on Scientific Computing*, 20(1):243–269, 1998.

[8] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms: from sequential to grid computing*. Numerical Analysis & Scientific Computing Series. Chapman & Hall / CRC, Boca Raton, FL, USA, 2007.

[9] J. M. Bahi, R. Couturier, and P. Vuillemin. Asynchronous iterative algorithms for computational science on the grid: three case studies. In M. J. Daydé, J. Dongarra, V. Hernández, and J. M. L. M. Palma, editors, *High Performance Computing for Computational Science - VECPAR 2004, 6th International Conference, Valencia, Spain, June 28-30, 2004, Revised Selected and Invited Papers*, volume 3402 of *Lecture Notes in Computer Science*, pages 302–314. Springer–Verlag, Berlin/Heidelberg, Germany, 2005.

[10] J. M. Bahi, J. C. Miellou, and K. Rhofir. Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, 15(3–4):315–345, January 1997.

[11] J. M. Bahi, S. Contassot-Vivier, and R. 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.

[12] J. M. Bahi, S. Contassot-Vivier, and R. 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*, pages 40–48, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):289–299, 2005.

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

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Performance comparison of parallel programming environments for implementing AIAC algorithms. *The Journal of Supercomputing*, 35(3):227–244, 2006.

[16] H. Bal. DAS–3 opening symposium, 2007. `http://www.cs.vu.nl/das3/symposium07/das3-bal.pdf`. Retrieved 05/02/2009.

[17] H. Bal and K. Verstoep. Large–scale parallel computing on grids. *Electronic Notes in Theoretical Computer Science*, 220(2):3–17, 2008.

[18] B. Barán, E. Kaszkurewicz, and A. Bhaya. Parallel asynchronous team algorithms: Convergence and performance analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):677–688, 1996.

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

[20] D. E. Baz, P. Spiteri, J. C. Miellou, and D. Gazen. Asynchronous iterative algorithms with flexible communication for nonlinear network flow problems. *Journal of Parallel and Distributed Computing*, 38(1):1–15, 1996.

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

[22] M. Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[23] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

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

[25] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, New York, NY, USA, 2004.

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

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

[28] B. Boghosian, P. Coveney, S. Dong, L. Finn, S. Jha, G. Karniadakis, and N. Karonis. NEKTAR, SPICE and Vortonics: using federated grids for large scale scientific applications. *Cluster Computing*, 10:351–364, September 2007.

[29] T. Brady, M. Guidolin, and A. Lastovetsky. Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model. In *GRID '08: Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, pages 49–56, Washington, DC, USA, 2008. IEEE Computer Society.

[30] T. Brady, E. Konstantinov, and A. Lastovetsky. SmartNetSolve: High level programming system for high performance Grid computing. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, 25-29 April 2006 2006. IEEE Computer Society. CD-ROM/Abstracts Proceedings.

[31] E. Brakkee, C. Vuik, and P. Wesseling. Domain decomposition for the incompressible Navier– Stokes equations: solving subdomain problems accurately and inaccurately. In *Domain Decomposition Methods in Sciences and Engineering*, pages 443–451. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[32] K. Burrage and J. Erhel. On the performance of various adaptive preconditioned GMRES strategies. *Numerical linear algebra with applications*, 5(2):101–121, March/April 1998.

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

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

[35] U. V. Çatalyürek and C. Aykanat. *PaToH: A multilevel hypergraph partitioning tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, Turkey. PaToH is available at `http://bmi.osu.edu/~umit/software.htm`, 1999.

[36] D. Chazan and W. L. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[37] A. T. Chronopoulos and C. W. Gear. S–step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[38] T. P. Collignon. Fast iterative solution of large linear systems on Grid computers. *MaC-Hazine: W.I.S.V. "Christiaan Huygens"*, 14(3):33–34, March 2010.

[39] T. P. Collignon, G. L. G. Sleijpen, and M. B. van Gijzen. Interpreting IDR(s) as a deflation method. *Journal of Computational and Applied Mathematics*, 2011. Special Issue: Proceedings ICCAM–2010 (submitted).

[40] T. P. Collignon and M. B. van Gijzen. 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*, pages 527–540, 2007.

[41] T. P. Collignon and M. B. van Gijzen. Parallel scientific computing on loosely coupled networks of computers. In B. Koren and C. Vuik, editors, *Advanced Computational Methods in Science and Engineering. Springer Series Lecture Notes in Computational Science and Engineering*, volume 71, pages 79–106. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

[42] T. P. Collignon and M. B. van Gijzen. Solving large sparse linear systems efficiently on Grid computers using an asynchronous iterative method as a preconditioner. In G. Kreiss, P. Lötstedt, A. Målqvist, and M. Neytcheva, editors, *Numerical Mathematics and Advanced Applications 2009: Proceedings of ENUMATH 2009, the 8th European Conference on Numerical Mathematics and Advanced Applications, Uppsala, July 2009*, pages 261–268. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

[43] T. P. Collignon and M. B. van Gijzen. Two implementations of the preconditioned Conjugate Gradient method on heterogeneous computing grids. *International Journal of Applied Mathematics and Computer Science*, 20(1):109–121, 2010.

[44] T. P. Collignon and M. B. van Gijzen. Fast iterative solution of large sparse linear systems on geographically separated clusters. *International Journal of High Performance Computing Applications*, 2011. (to appear).

[45] T. P. Collignon and M. B. van Gijzen. Minimizing synchronization in IDR(s). *Numerical Linear Algebra with Applications*, 2011. (published online: 14 january 2011).

[46] R. Couturier, C. Denis, and F. Jézéquel. GREMLINS: a large sparse linear solver for grid environment. *Parallel Computing*, 34(6–8):380–391, July 2008. Parallel Matrix Algorithms and Applications.

[47] R. Couturier and S. 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*, pages 289–296, Long Beach, CA, USA, March 2007. IEEE Computer Society.

[48] 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.

[49] E. de Sturler. A performance model for Krylov subspace methods on mesh–based parallel computers. *Parallel Computing*, 22(1):57–74, 1996.

[50] E. de Sturler. Truncation strategies for optimal Krylov subspace methods. *SIAM Journal on Numerical Analysis*, 36(3):864–889, 1999.

[51] F. Desprez and E. Jeannot. 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, University College Cork, Ireland, (ISPDC/HeteroPar'04)*, pages 193–200, Washington, DC, USA, 2004. IEEE Computer Society.

[52] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[53] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[54] S. Dong, G. E. Karniadakis, and N. T. Karonis. Cross–Site Computations on the TeraGrid. *Computing in Science and Engineering*, 7(5):14–23, 2005.

[55] J. Dongarra and A. Lastovetsky. An overview of heterogeneous high performance and Grid computing. In B. DiMartino, J. Dongarra, A. Hoisie, L. Yang, and H. Zima, editors, *Engineering the Grid: Status and Perspective*, Stevenson Ranch, CA, USA, February 2006. American Scientific Publishers.

[56] J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, USA, 1998.

[57] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann, 2003.

[58] J. Dongarra, Y. Li, Z. Shi, D. Fike, K. Seymour, and A. YarKhan. Homepage of NetSolve/GridSolve, 2007. `http://icl.cs.utk.edu/netsolve/`.

[59] C. C. Douglas. A review of numerous parallel multigrid methods. In G. Astfalk, editor, *Applications on Advanced Architecture Computers*, pages 187–202. SIAM, Philadelphia, PA, USA, 1996.

[60] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM Journal on Numerical Analysis*, 20:345–357, 1983.

[61] J. Erhel, K. Burrage, and B. Pohl. Restarted GMRES preconditioned by deflation. *Journal of Computational and Applied Mathematics*, 69(2):303–318, 1996.

[62] Y. A. Erlangga and R. Nabben. Deflation and balancing preconditioners for Krylov subspace methods applied to nonsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 30(2):684–699, 2008.

[63] W.-C. Feng and T. Scogland. The green500 list: Year one. In *5th IEEE Workshop on High-Performance, Power-Aware Computing (in conjunction with the 23rd International Parallel & Distributed Processing Symposium)*, Rome, Italy, May 2009.

[64] Folding. Folding@home distributed computing. `http://folding.stanford.edu/`.

[65] I. Foster and C. Kesselman. *The Grid: Blueprint for a new Computing Infrastructure*. Morgan Kaufman Publishers, San Fransisco, USA, second edition, 2004.

[66] I. Foster. Globus toolkit version 4: Software for service–oriented systems. In H. Jin, D. Reed, and W. Jiang, editors, *Network and parallel computing: IFIP international conference, NPC 2005, Beijing, China, November/December 2005*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer–Verlag, Berlin/Heidelberg, Germany, 2005.

[67] J. Frank and C. Vuik. On the construction of deflation–based preconditioners. *SIAM Journal on Scientific Computing*, 23(2):442–462, 2001.

[68] A. Frommer, H. Schwandt, and D. B. Szyld. Asynchronous weighted additive Schwarz methods. *Electronic Transactions on Numerical Analysis*, 5:48–61, 1997.

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

[70] A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.

[71] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[72] G. H. Golub and C. F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.

[73] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[74] W. Gropp, E. Lusk, and R. Thakur. *Using MPI–2: Advanced Features of the Message–Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[75] T.-X. Gu, X.-Y. Zuo, X.-P. Liu, and P.-L. Li. An improved parallel hybrid bi–conjugate gradient method suitable for distributed parallel computing. *Journal of Computational and Applied Mathematics*, 226(1):55–65, 2009.

[76] T.-X. Gu, X.-Y. Zuo, L.-T. Zhang, W.-Q. Zhang, and Z. Sheng. An improved bi–conjugate residual algorithm suitable for distributed parallel computing. *Applied Mathematics and Computation*, 186(2):1243–1253, 2007.

[77] J. Gustafson. The program of Grand Challenge problems: Expectations and results. In N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms and Architecture Synthesis, March 17–21, 1997, Aizu–Wakamatsu, Fukushima, Japan*, PAS '97, pages 2–7, Washington, DC, USA, 1997. IEEE Computer Society.

[78] M. H. Gutknecht. IDR Explained. *Electronic Transactions on Numerical Analysis*, 36:126–148, 2010.

[79] M. H. Gutknecht and J.-P. M. Zemke. Eigenvalue computations based on IDR. Technical report, Seminar für Angewandte Mathematik, ETH Zürich, SAM, ETH Zürich, Switzerland, 2010. Research Report No. 2010–13.

[80] Y. Hassen and B. Koren. Finite–volume discretization and immersed boundaries. In B. Koren and C. Vuik, editors, *Advanced Computational Methods in Science and Engineering. Springer Series Lecture Notes in Computational Science and Engineering*, volume 71, pages 229–268. Springer–Verlag, Berlin/Heidelberg, Germany, 2010.

[81] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for solving linear systems. *Journal of Research of National Bureau Standards*, 49(6):409–436, 1952.

[82] I. C. F. Ipsen and C. D. Meyer. The idea behind Krylov methods. *American Mathematical Monthly*, 105:889–899, 1998.

[83] E. F. Kaasschieter. Preconditioned Conjugate Gradients for solving singular systems. *Journal of Computational and Applied Mathematics*, 24(1-2):265–275, 1988.

[84] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[85] N. T. Karonis, B. Toonen, and I. Foster. MPICH–G2: A Grid–enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003. Special Issue on Computational Grids.

[86] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing, Cancun, Mexico, May 1-5, 2000*, pages 377–384, Washington, DC, USA, 2000. IEEE Computer Society.

[87] B. Krasnopolsky. The reordered BiCGStab method for distributed memory computer systems. *Procedia Computer Science*, 1(1):213–218, 2010. ICCS 2010.

[88] A. Lastovetsky, X. Zuo, and P. Zhao. A non–intrusive and incremental approach to enabling direct communications in RPC-based grid programming systems. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2006: 6th International Conference, Reading, UK, May 28-31*, volume 3993, pages 1008–1011. Springer–Verlag, Berlin/Heidelberg, Germany, 2006.

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

[90] G. Mercier. MPICH–Madeleine. an MPI implementation for heterogeneous clusters of clusters, 2006. `http://runtime.futurs.inria.fr/mpi/`.

[91] H. W. Meuer. The top500 project: Looking back over 15 years of supercomputing experience. *Informatik–Spektrum*, 31(3):203–222, June 2008.

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

[93] B. Mirghani, M. Tryby, D. Baessler, N. Karonis, R. Ranhthan, and K. Mahinthakumar. Development and performance analysis of a simulation–optimization framework on TeraGrid linux clusters. In *The 6th LCI International Conference on Linux Clusters: The HPC Revolution 2005*. Chapel Hill, NC, USA, April 26-28 2005.

[94] R. Nabben and C. Vuik. A comparison of deflation and coarse grid correction applied to porous media flow. *SIAM Journal on Numerical Analysis*, 42(4):1631–1647, 2004.

[95] R. Nabben and C. Vuik. A comparison of deflation and the balancing preconditioner. *SIAM Journal on Scientific Computing*, 27(5):1742–1759, 2006.

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

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

[98] Y. Onoue, S. Fujino, and N. Nakashima. An overview of a family of new iterative methods based on IDR theorem and its estimation. In S. I. Ao, O. Castillo, C. Douglas, D. D. Feng, and J.-A. Lee, editors, *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.

[99] V. S. Pande. A simple theory of protein folding kinetics. *Physical Review Letters*, Jul 2010.

[100] V. Pereyra. Asynchronous distributed solution of large scale nonlinear inversion problems. In *Selected papers of the second Panamerican workshop on Applied and computational mathematics*, pages 31–40, Amsterdam, The Netherlands, 1999. North-Holland Publishing Co.

[101] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL – a portable implementation of the high–performance Linpack benchmark for distributed–memory computers, 2008. Available at `http://www.netlib.org/benchmark/hpl/`.

[102] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2003.

[103] Y. Saad. A flexible inner–outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

[104] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[105] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison–Wesley Professional, 1st edition, July 2010.

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

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

[108] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. NetSolve: Grid enabling scientific computing environments. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, New York, NY, USA, 2005.

[109] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *GRID '02: Proceedings of the Third International Workshop on Grid Computing, Baltimore, MD, USA, November 18, 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer–Verlag, Berlin/Heidelberg, Germany, 2002.

[110] V. Simoncini and D. B. Szyld. Flexible inner–outer Krylov subspace methods. *SIAM Journal on Numerical Analysis*, 40(6):2219–2239, 2002.

[111] V. Simoncini and D. B. Szyld. Recent computational developments in Krylov subspace methods for linear systems. *Numerical Linear Algebra with Applications*, 14:1–59, 2007.

[112] V. Simoncini and D. B. Szyld. Interpreting IDR as a Petrov–Galerkin method. *SIAM Journal on Scientific Computing*, 32(4):1898–1912, 2010.

[113] G. L. G. Sleijpen and D. Fokkema. BiCGstab(ℓ) for linear equations involving unsymmetric matrices with complex spectrum. *Electronic Transactions on Numerical Analysis*, 1:11–32, 1993.

[114] G. L. G. Sleijpen and H. A. van der Vorst. Maintaining convergence properties of BiCGstab methods in finite precision arithmetic. *Numerical Algorithms*, 10(3–4):203–223, 1995.

[115] G. L. G. Sleijpen, P. Sonneveld, and M. B. van Gijzen. Bi–CGSTAB as an induced dimension reduction method. *Applied Numerical Mathematics*, 60(11):1100–1114, 2010. Special Issue: 9th IMACS International Symposium on Iterative Methods in Scientific Computing (IISIMSC 2008).

[116] G. L. G. Sleijpen and M. B. van Gijzen. The algebra for induced dimension reduction. In *The Proceedings of the 2nd Kyoto–Forum on Krylov Subspace Methods, Kyoto University, Kyoto, Japan*, March 2010.

[117] G. L. G. Sleijpen and M. B. van Gijzen. Exploiting BiCGstab(ℓ) strategies to induce dimension reduction. *SIAM Journal on Scientific Computing*, 32(5):2687–2709, 2010.

[118] B. F. Smith, P. E. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, Cambridge, 1996.

[119] P. Sonneveld. On the convergence behaviour of IDR(s). Technical report, Delft University of Technology, Delft, The Netherlands, 2010. DUT report 10–08.

[120] P. Sonneveld and M. B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2008.

[121] T. Sterling, E. Lusk, and W. Gropp, editors. *Beowulf Cluster Computing with Linux*. MIT Press, Cambridge, MA, USA, 2003.

[122] D. B. Szyld. Different models of parallel asynchronous iterations with overlapping blocks. *Computational and Applied Mathematics*, 17:101–115, 1998.

[123] D. B. Szyld and J. A. Vogel. FQMR: A flexible quasi–minimal residual method with inexact preconditioning. *SIAM Journal on Scientific Computing*, 23(2):363–380, 2001.

[124] D. B. Szyld and J.-J. Xu. Convergence of some asynchronous nonlinear multisplitting methods. *Numerical Algorithms*, 25:347–361, 2000.

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

[126] J. M. Tang, S. P. MacLachlan, R. Nabben, and C. Vuik. A comparison of two–level preconditioners based on multigrid and deflation. *SIAM Journal on Matrix Analysis and Applications*, 31(4):1715–1739, 2010.

[127] J. M. Tang, R. Nabben, C. Vuik, and Y. A. Erlangga. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing*, 39(3):340–370, 2009.

[128] 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.

[129] J. M. Tang and C. Vuik. On deflation and singular symmetric positive semi–definite matrices. *Journal of Computational and Applied Mathematics*, 206(2):603–614, 2007.

[130] J. M. Tang. *Two–Level Preconditioned Conjugate Gradient Methods with Applications to Bubbly Flow Problems*. PhD thesis, Delft University of Technology, 2008.

[131] M. Tanio and M. Sugihara. GBi-CGSTAB(s, L): IDR(s) with higher-order stabilization polynomials. *Journal of Computational and Applied Mathematics*, 235(3):765–784, 2010.

[132] J. D. Teresco, K. D. Devine, and J. E. Flaherty. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations. Springer–Verlag, Berlin/Heidelberg, Germany, 2005.

[133] A. Toselli and O. B. Widlund. *Domain Decomposition: Algorithms and Theory*, volume 34. Springer–Verlag, Berlin/Heidelberg, Germany, 2005.

[134] L. G. Valiant. A bridging model for multi–core computing. In D. Halperin and K. Mehlhorn, editors, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, pages 13–28. Springer–Verlag, Berlin/Heidelberg, Germany, 2008.

[135] S. 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.

[136] 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.

[137] H. A. van der Vorst and C. Vuik. GMRESR: a family of nested GMRES methods. *Numerical Linear Algebra with Applications*, 1(4):369–386, 1994.

[138] M. B. van Gijzen and T. P. Collignon. Exploiting the flexibility of IDR(s) for Grid computing. In *The Proceedings of the 2nd Kyoto–Forum on Krylov Subspace Methods, Kyoto University, Kyoto, Japan*, March 2010.

[139] M. B. van Gijzen and P. Sonneveld. An IDR(s) variant with minimal intermediate residual norms. In *The Proceedings of the International Kyoto–Forum on Krylov Subspace methods, Kyoto University, Kyoto, Japan*, pages 85–92, 2008.

[140] M. B. van Gijzen and P. Sonneveld. An elegant IDR(s) variant that efficiently exploits bi–orthogonality properties. Technical report, Delft University of Technology, Delft, The Netherlands, 2010. DUT report 10–16 (revised version of DUT report 08–21).

[141] J. van Kan. A second–order accurate pressure correction scheme for viscous incompressible flow. *SIAM Journal on Scientific and Statistical Computing*, 7(3):870–891, 1986.

[142] B. Vastenhouw and R. H. Bisseling. A two–dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[143] F. Vernier, J. M. Bahi, S. Contassot-Vivier, and R. Couturier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):4–13, 2005.

[144] K. Verstoep, J. Maassen, H. E. Bal, and J. W. Romein. Experiences with fine–grained distributed supercomputing on a 10G testbed. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 376–383, Washington, DC, USA, 2008. IEEE Computer Society.

[145] P. Wesseling. *An Introduction to Multigrid Methods.* John Wiley & Sons, Ltd., Chichester, UK, 1992.

[146] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[147] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[148] L. Wolters, S. P. Zwart, A. Doelman, B. Koren, J. Batenburg, G. Barkema, R. Bisseling, G. Cats, K. Oosterlee, and K. Vuik. The Little GREEN Machine, 2010. `http://www.littlegreenmachine.org/`.

[149] R. Wyrzykowski, N. Meyer, and M. Stroinski. Concept and implementation of CLUSTERIX: National cluster of linux systems. In *The 6th LCI International Conference on Linux Clusters: The HPC Revolution 2005.* Chapel Hill, NC, April 2005.

[150] R. Wyrzykowski, N. Meyer, T. Olas, L. Kuczynski, B. Ludwiczak, C. Czaplewski, and S. Oldziej. Meta–computations on the CLUSTERIX grid. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing: State of the Art in Scientific Computing, 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers*, volume 4699 of *Lecture Notes in Computer Science*, pages 489–500. Springer–Verlag, Berlin/Heidelberg, Germany, 2006.

[151] L. Yang and R. Brent. The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures. In *5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 324–328, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[152] T. Yang. The improved CGS method for large and sparse linear systems on bulk synchronous parallel architecture. In *ICA3PP '02: Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 232–237, Washington, DC, USA, 2002. IEEE Computer Society.

[153] T. Yang and H.-X. Lin. The improved quasi–minimal residual method on massively distributed memory computers. In L. O. Hertzberger and P. M. A. Sloot, editors, *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1997, Vienna, Austria, April 28-30, 1997, Proceedings*, volume 1225 of *Lecture Notes in Computer Science*, pages 389–399. Springer–Verlag, Berlin/Heidelberg, Germany, 1997.

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

[155] Y. Zheng, A. Bassi, M. Beck, J. S. Plank, and R. Wolski. Internet Backplane Protocol: C API 1.4. Technical report, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, December 2004.

[156] X. Zuo and A. Lastovetsky. Experiments with a software component enabling NetSolve with direct communications in a non–intrusive and incremental way. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–8, Washington, DC, USA, 2007. IEEE Computer Society.

Index

$\mathbf{0}_{n,s}$, 29
$\mathbf{A}^*$, 29
$\mathbf{A}^\perp$, 29
$\mathbf{B}$, 29
$\mathbf{I}$, 29
$\mathbf{u}_x$, 29
$\mathcal{G}$, 29
$\mathcal{K}_r(\mathbf{A}, \mathbf{v})$, 29
$\mathcal{N}(\mathbf{A})$, 29
$\mathcal{S}$, 29
$\mathrm{span}(\mathbf{A})$, 29
$\dim \mathcal{A}$, 29
ω_k, 29
$\phi_{(m:n)}$, 29
$\mathrm{rank}\,\mathbf{A}$, 29
$\sigma(\mathbf{A})$, 29
$\widetilde{\mathbf{R}}$, 29
s, 29
t, 29

ASSIST, 16
asynchronous block Jacobi iteration, 8
  CRAC task, 48
  GridSolve task, 57

bandwidth, 80
  DAS–3, 85
  LU site, 85
block Jacobi iteration, 7
  asynchronous, 8
  synchronous, 8
bubbly flows, 32, 49

Cell computing, 3
classical iteration, 6
Conjugate Gradient method, 10
  Chronopoulos and Gear variant of, 32, 37
  flexible, 47
  preconditioned, 10, 36
  resource–aware, 34
convection–diffusion problem, 58, 85
CRAC, 18
CRS, 37

DAS–3, 21
deflation methods, 62, 101
  A–DEF1, 103, 115
  A–DEF2, 61
  adapted, 103
  DEF1, 115
dimension reduction step, 72, 101
direct methods, 4
domain decomposition, 7
DSI, 18

GAT, 15
GbE, 21
Gigabit Ethernet, 21
Globus Toolkit, 19
GMRESR method, 56
  truncated, 56
GPU computing, 3, 125
Gram–Schmidt method
  classical, 57
  modified, 49
Grand Challenge Problems, 123
Grid computing, 3
GridRPC, 16
GridSolve, 16, 56

Hermann Schwarz, 7
hypergraph partitioning
  Mondriaan, 22
  PaToH, 22

IBP, 18
ICRS, 37
IDR residual

auxiliary, 100
intermediate, 72
primary, 74, 100
secondary, 100
IDR test matrix, 29, 70
IDR(s) cycle, 72, 100
IDR(s) method, 10
bi–orthogonalisation of intermediate residuals variant of, 73
minimal synchronisation points variant of, 75
prototype variant of, 71
IDR(s) theorem, 71, 100
Immersed Boundary Method, 24

Jacobi iteration, 6

latency, 80
DAS–3, 85
LU site, 85

matrix partitioning, 34
message crunching, 18
middleware, 15
CRAC, 18
GridSolve, 16, 56
Open MPI, 19
MPI library, 19
MPICH–G2, 19
MPICH–Madeleine, 19
Open MPI, 19, 85
multi–core, 3, 68, 82
MV, 29
Myri–10G, 21

NetSolve, 16

Open MPI, 19, 85

partitioning, 22
matrix, 22
performance model, 80
preconditioning, 10, 23
asynchronous, 11, 48, 83
IC(0), 50
ILU, 59
variable, 11, 48, 83
projection, 10
oblique, 72

Richardson iteration, 6
RPC, 15

scalability
strong, 5
weak, 5
search matrix step, 101
Sonneveld subspace, 100
subspace methods, 10
flexible, 11
SURFnet, 21