Parallel algorithms and efficient implementation techniques for finite element approximations

THÈSE N^{O} 5980 (2013)

PRÉSENTÉE LE 18 NOVEMBRE 2013
À LA FACULTÉ DES SCIENCES DE BASE
CHAIRE DE MODÉLISATION ET CALCUL SCIENTIFIQUE
PROGRAMME DOCTORAL EN MATHÉMATIQUES

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Radu POPESCU

acceptée sur proposition du jury:

Prof. K. Hess Bellwald, présidente du jury
Prof. A. Quarteroni, Dr S. Deparis, directeurs de thèse
Prof. L. Formaggia, rapporteur
Prof. M. A. Heroux, rapporteur
Prof. M. Picasso, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2013

Put a nice quote here. Plenty of time to decide…

Support

This work has been financially supported by the Swiss High Performance and High Productivity Computing (HP2C)[1] initiative and by the Chair of Modelling and Scientific Computing (CMCS) at École Polytechnique Fédérale de Lausanne (EPFL)[2].

I would like to acknowledge the Center for Advanced Modelling Science (CADMOS)[3] and the Swiss Center for Scientific Computing (CSCS)[4] for providing the invaluable computational resources needed for the work described in this thesis and for the many hours of help and support provided by their staff.

Finally I would like to acknowledge our collaborators Michael Heroux, Siva Rajamanickam and Erik Boman from the Trilinos Project[5] at Sandia National Laboratories in New Mexico, USA. I am truly grateful for our collaboration and for the hospitality they extended during my internship at Sandia National Labs.

Lausanne, October 22, 2013 R. P.

---

[1] HP2C - http://www.hp2c.ch

[2] CMCS - EPFL - http://cmcs.epfl.ch

[3] CADMOS - http://www.cadmos.org

[4] CSCS - http://www.cscs.ch

[5] Trilinos Project - http://trilinos.sandia.gov

Acknowledgements

I have reached the end of this trip and very soon I will start a new one. It's time for me to look back at the last four years of my life and try to understand which were the key points and people who helped me on my way.

Coming to Switzerland to work in the group here at EPFL was definitely a life changing decision for me. It's the best decision I've ever made and it wouldn't have come to pass without Alfio Quarteroni, who agreed to hire me and welcomed me into his very close-knit group. Although, due to the topic of my work, we weren't really able to work together directly, knowing that he was there keeping an eye on me and directing me when needed has always been very reassuring for me.

Then there is Simone Deparis, my other advisor, with whom I've spent countless days (accumulated, not at once) discussing and planning my work. I am grateful for his support, his advice and his knowledge and if I am to look back and try to identify my fondest memory with him, it is definitely the many hours we passed together solving problems related to the LifeV software project. It was really fun putting the tools together and keeping them working. I'm retiring my SSH keys soon, I hope the next one to do this work will enjoy it as much as I did.

I am very lucky to have been part of the CMCS group: really competent and friendly people, past and present, a relaxing and welcoming working environment, one is lucky to find this at his workplace.

I would also like to acknowledge Mike Heroux. I'm really happy to have started working with him and I think he's an example: extremely competent and insightful, but with such a calm and kind demeanor that makes it really confortable to work with him. Also, he has really crazy time management skills! I don't know how he and Alfio can both keep going like this but it's really impressive.

Of course, I wouldn't call this period life changing if work was the only good part of it. I've made many good friends here. Paolo (T), Elena, Claudia, Gosia, Jacopo, Ale: I thank you for your friendship and know that I really cherish the time we spent together so far. It has really changed me and you all know this best. Gilles, I loved our dinners in Bucharest and talking to you in the evening on the computer, usually before you needed to put Ana to bed... I would also like to remember the friends who are no longer with us now... Paolo Crosetto (he's in Germany), Ricardo (he's at UNIL). Then there's Kelly and Danny (they are in Austria). Kelly, I want to thank you last: being friends with you has definitely brought many crazy and weird experiences to my life, but they are some of the brightest memories from these years. Know that I will not be able to look at Morges the same way as before.

Then there is an unexpected source of distraction (the good kind), support and affection that has made its way into my life and made it better. Helene, you made writing this thesis so much easier. You keep me warm and level-headed and I've grown to rely on you.
Last, but not least, there is my family: my parents, Omama and my sisters Anca and Lidia with their own families. I thank you all for making me (where applicable), raising me, loving me and supporting me always. I am truly lucky to have such a wonderful family.

And this is where I'd also thank my cat, if I had one... Hypothetical Mittens, at this point I would have been very grateful for what I assume had been your enduring affection throughout all this.

Lausanne, October 22, 2013 R. P.

Abstract

In this thesis we study the efficient implementation of the finite element method for the numerical solution of partial differential equations (PDE) on modern parallel computer architectures, such as Cray and IBM supercomputers. The domain-decomposition (DD) method represents the basis of parallel finite element software and is generally implemented such that the number of subdomains is equal to the number of MPI processes. We are interested in breaking this paradigm by introducing a second level of parallelism. Each subdomain is assigned to more than one processor and either MPI processes or multiple threads are used to implement the parallelism on the second level. The thesis is devoted to the study of this second level of parallelism and includes the stages described below.

The algebraic additive Schwarz (AAS) domain-decomposition preconditioner is an integral part of the solution process. We seek to understand its performance on the parallel computers which we target and we introduce an improved construction approach for the parallel preconditioner. We examine a novel strategy for solving the AAS subdomain problems, using multiple MPI processes. At the subdomain level, this is represented by the ShyLU preconditioner. We bring improvements to its algorithm in the form of a novel inexact solver based on an incomplete QR (IQR) factorization. The performance of the new preconditioner framework is studied for Laplacian and advection-diffusion-reaction (ADR) problems and for Navier-Stokes problems, as a component within a larger framework of specialized preconditioners.

The partitioning of the computational mesh comes with considerable memory limitations, when done at runtime on parallel computers, due to the low amount of available memory per processor. We describe and implement a solution to this problem, based on offloading the partitioning process to a preliminary offline stage of the simulation process. We also present the efficient implementation, based on parallel MPI collective instructions, of the routines which load the mesh parts during the simulation.

We discuss an alternative parallel implementation of the finite element system assembly based on multi-threading. This new approach is used to supplement the existing one based on MPI parallelism, in situations where MPI alone can not make use of all the available parallel hardware resources.

The work presented in the thesis has been done in the framework of two software projects: the Trilinos project and the LifeV parallel finite element modeling library. All the new developments have been contributed back to the respective projects, to be used freely in subsequent public releases of the software.

Keywords: finite element method, parallel preconditioners, MPI, multi-threading

Résumé

Dans cette thèse, nous proposons une implémentation efficace de la méthode des éléments finis pour la résolution numérique d'équations aux dérivées partielles (EDP) sur des architectures parallèles modernes telles que les superordinateurs Cray et IBM. La méthode de décomposition de domaine (DD) constitue la base des logiciels parallèles d'éléments finis et est généralement implémentée de sorte que le nombre de sous-domaines corresponde à celui des processus MPI. On s'attache à casser ce paradigme en introduisant un second niveau de parallélisme. Chaque sous-domaine est assigné à plus d'un processeur et le second niveau de parallélisme est implémenté à l'aide, soit de processus MPI, soit de threads multiples. La présente thèse est dédiée à l'étude de ce second niveau de parallélisme et inclut les étapes décrites ci-après.

Le préconditionneur décomposition de domaines de type Schwarz additif algébrique (AAS) fait partie intégrante du processus de résolution. On cherche à comprendre sa performance sur les ordinateurs parallèles considérés et on introduit une approche de construction améliorée pour le préconditionneur parallèle. On examine une nouvelle stratégie pour résoudre les problèmes de sous-domaines AAS, en utilisant plusieurs processus MPI. Au niveau du sous-domaine, elle est représentée par le préconditionneur ShyLU. On apporte des améliorations à son algorithme sous la forme d'un nouveau solveur inexact basé sur une factorisation QR incomplète (IQR). Les performances obtenues à l'aide du nouveau préconditionneur sont étudiées sur des problèmes de Laplace et d'advection-diffusion-reaction (ADR) ainsi que sur des problèmes de Navier-Stokes, comme une composante appartenant à un cadre plus large de préconditionneurs spécialisés.

Le partitionnement du maillage computationnel induit des limitations de mémoire considérables lorsqu'il est réalisé durant l'exécution sur des ordinateurs parallèles, du fait de la faible quantité de mémoire disponible sur chaque processeur. On décrit et on implémente une solution pour ce problème, en opérant un transfert du processus de partitionnement à un niveau préliminaire "offline" du processus de simulation. On présente également une implémentation performante - s'appuyant sur des instructions parallèles MPI collectives - des tâches qui chargent les parties du maillage durant la simulation.

On discute une implémentation parallèle alternative du procédure d'assemblage des éléments finis basée sur le multi-threading. Cette nouvelle approche est utilisée pour compléter celle, existante, basée sur le parallélisme MPI, dans des cas où MPI seul est incapable d'exploiter toutes les ressources matérielles parallèles disponibles.

Le travail présenté dans cette thèse a été effectué dans le cadre de deux projets logiciels :

le projet Trilinos et la bibliothèque parallèle de modélisation éléments finis LifeV. Toutes les nouvelles contributions ont participé au développement de leur projet respectif, dans l'optique d'une utilisation libre dans les futures versions publiques du logiciel.

Mots-clés : méthode des éléments finis, préconditionnneurs paralleles, MPI, multi-threading

Contents

Support **v**

Acknowledgements **vii**

Abstract (English/Français) **ix**

List of figures **xv**

List of tables **xviii**

1 Introduction **1**

2 State of the art of finite element software **7**
- 2.1 Introduction 7
- 2.2 The finite element method for the approximation of PDEs 7
  - 2.2.1 Weak formulation 8
  - 2.2.2 Finite dimensional approximation 9
  - 2.2.3 Assembly of the linear system of equations 11
- 2.3 Scientific computing hardware and software tools 13
  - 2.3.1 Vectorization 16
- 2.4 Serial and parallel implementation 17
- 2.5 Parallel libraries for finite element modelling 23
  - 2.5.1 GPU for numerical applications 25
- 2.6 The LifeV parallel finite element modeling library 27
  - 2.6.1 Design and implementation overview 28
  - 2.6.2 Initial limitations of LifeV 29

3 A domain decomposition preconditioner based on two-level MPI parallelism **31**
- 3.1 The algebraic additive Schwarz preconditioner 31
  - 3.1.1 Optimality and scalability 32
  - 3.1.2 Restricted Schwarz algorithm 34
  - 3.1.3 Numerical test 35
- 3.2 Two-level parallelism in AAS 37
  - 3.2.1 Implementation details 38
  - 3.2.2 Numerical assessment 39

3.3 ShyLU - a hybrid subdomain preconditioner 40
3.3.1 Algorithmic description 40
3.3.2 Implementation overview 42
3.4 Benchmarks 44
3.4.1 Node-level measurements 44
3.4.2 Multi-node measurements 52
3.5 Closing remarks 60

4 A preconditioner based on an incomplete QR factorization 61
4.1 Introduction 61
4.2 Rationale 61
4.3 The incomplete QR factorization 63
4.3.1 The Generalized Minimum Residual Method 63
4.3.2 The incomplete QR factorization as a preconditioner 65
4.4 Using IQR within ShyLU 67
4.4.1 Numerical benchmarks and discussion 68
4.5 Conclusions 71

5 Preprocessing requirements 75
5.1 Introduction 75
5.2 Runtime mesh partitioning 75
5.2.1 Building the DOF maps 77
5.2.2 Overlap support for partitioning 78
5.2.3 Limitation of runtime partitioning 78
5.3 Alternate mesh partitioning techniques 79
5.3.1 Offline partitioning 79
5.3.2 Hierarchical partitioning 81
5.3.3 Runtime partitioning with MPI-3.0 shared memory regions 81
5.4 High speed parallel mesh loading 83
5.4.1 Optimization with MPI-IO collectives 84

6 Multi-threaded finite element assembly 87
6.1 Introduction 87
6.2 Requirements of the multi-threaded implementation 87
6.3 Multi-threaded implementation 90
6.3.1 Work sharing 91
6.3.2 Synchronization 91
6.4 Benchmarks 93
6.4.1 Experiment setup 93
6.4.2 Measurements and discussion 94
6.5 Conclusions 95

7 Large scale simulations **97**
7.1 The Navier-Stokes equations . . . 97
7.1.1 Weak form . . . 97
7.1.2 Finite element discretization . . . 100
7.1.3 The solution of the nonlinear system of equations . . . 101
7.1.4 Preconditioners for Navier-Stokes equations . . . 103
7.2 Numerical benchmarks . . . 104
7.2.1 Description of test problems . . . 104
7.2.2 Results and discussion . . . 106
7.3 Closing remarks . . . 111

8 Summary of contributions, recommendations and future work **113**
8.1 Contributions to Trilinos . . . 113
8.2 Contributions to LifeV . . . 115
8.3 Recommendations . . . 116
8.3.1 Preprocessing and finite element assembly . . . 116
8.3.2 Linear solver and preconditioner . . . 117
8.4 Future work . . . 119

Bibliography **129**

Curriculum Vitae **131**

List of Figures

2.1 Simplified diagram of a hybrid compute cluster 15
2.2 Block diagram of a serial finite element method implementation 17
2.3 Ideal parallel speedup according to Amdahl's law 18
2.4 Block diagram of a parallel finite element method implementation 20
2.5 Distribution of the rows of a parallel sparse matrix 22

3.1 Domain decomposition in two overlapping subdomains 32
3.2 Comparison of AAS and 2-level AAS for a 3D Laplacian problem 36
3.3 Domain decomposition for AS with one or two levels of parallelism 38
3.4 Verification of the implementation of parallel subdomain problems for AAS . . 39
3.5 The partitioned and reordered system matrix in ShyLU 40
3.6 Convergence history of GMRES subiterations in ShyLU 44
3.7 ShyLU stand-alone - Time to compute the preconditioner - 3D Laplacian problems 46
3.8 ShyLU stand-alone - Number of GMRES iterations - 3D Laplacian problems . . 47
3.9 ShyLU stand-alone - Time per GMRES iteration - 3D Laplacian problems 47
3.10 ShyLU stand-alone - Total time to solution - 3D Laplacian problems 48
3.11 ShyLU stand-alone - Time to compute the preconditioner - 3D ADR problems . 49
3.12 ShyLU stand-alone - Number of GMRES iterations - 3D ADR problems 50
3.13 ShyLU stand-alone - Time per GMRES iterations - 3D ADR problems 50
3.14 ShyLU stand-alone - Total time to solution - 3D ADR problems 51
3.15 AAS / ShyLU - Time to compute the preconditioner - 3D Laplacian problems . 54
3.16 AAS / ShyLU - Number of GMRES iterations - 3D Laplacian problems 55
3.17 AAS / ShyLU - Time to GMRES convergence - 3D Laplacian problems 55
3.18 AAS / ShyLU - Time per GMRES iterations - 3D Laplacian problems 56
3.19 AAS / ShyLU - Total time to solution - 3D Laplacian problems 56
3.20 AAS / ShyLU - Time to compute the preconditioner - 3D ADR problems 57
3.21 AAS / ShyLU - Number of GMRES iterations - 3D ADR problems 58
3.22 AAS / ShyLU - Time to GMRES convergence - 3D ADr problems 58
3.23 AAS / ShyLU - Time per GMRES iteration - 3D ADR problems 59
3.24 AAS / ShyLU - Total time to solution - 3D ADR problems 59

4.1 ShyLU with IQR - Time to compute the preconditioner - 3D ADR problems . . . 70
4.2 ShyLU with IQR - Number of GMRES iterations - 3D ADR problems 71

4.3 ShyLU with IQR - Time to GMRES convergence - 3D ADR problems 72
4.4 ShyLU with IQR - Time per GMRES iteration - 3D ADR problems 73
4.5 ShyLU with IQR - Total time to solution - 3D ADR problems 73

5.1 The dual graph of a structured triangular mesh of a square domain 76
5.2 The partition of the dual graph of a structured triangular mesh of a square domain 76
5.3 Unique and repeated maps 77
5.4 Mesh partitioning with overlap 78
5.5 Unique, repeated and ghost maps in the case of mesh partitioning with overlap 79
5.6 The two steps of the offline partitioning strategy 80
5.7 Initial implementation of the HDF5 storage for mesh parts 84
5.8 Optimized implementation of the HDF5 storage for mesh parts 85

6.1 FE assembly loop in the MPI single-threaded case 88
6.2 Breakdown of the cost of finite element matrix assembly 90
6.3 FE assembly loop in the multi-threaded case 91
6.4 Strong scalability of the multi-threaded assembly on the IBM BlueGeen/Q . . . 95
6.5 Strong scalability of the multi-threaded assembly on the Cray XE6 96

7.1 The geometry of the aneurysm problems 105
7.2 Navier-Stokes problem - Time to compute the preconditioner - 4.8 million DOFs 107
7.3 Navier-Stokes problem - Number of GMRES iterations - 4.8 million DOFs 107
7.4 Navier-Stokes problem - Time to GMRES convergence - 4.8 million DOFs 108
7.5 Navier-Stokes problem - Time per GMRES iterations - 4.8 million DOFs 108
7.6 Navier-Stokes problem - Total time to solution - 4.8 million DOFs 109
7.7 Navier-Stokes problem - Time to compute the preconditioner - 37 million DOFs 109
7.8 Navier-Stokes problem - Number of GMRES iterations - 37 million DOFs 110
7.9 Navier-Stokes problem - Time to GMRES convergence - 37 million DOFs 110
7.10 Navier-Stokes problem - Time per GMRES iteration - 37 million DOFs 110
7.11 Navier-Stokes problem - Total time to solution - 37 million DOFs 111

List of Tables

4.1 Breakdown of ShyLU runtime with MUMPS as a solver for the Schur complement system . . . 62
4.2 Breakdown of ShyLU runtime with GMRES subiterations on the Schur complement system . . . 62
4.3 Dimensions of the IQR Krylov subspaces . . . 69

6.1 Best case speedup and efficiency of the multi-threaded assembly . . . 96

8.1 Quick reference table for the configuration of finite element simulation with LifeV and Trilinos . . . 118

1 Introduction

In this thesis we study the parallel implementation of the finite element method for the numerical solution of partial differential equations (PDE). More specifically, we focus on algorithms and techniques involving two levels of parallelism, obtained either with MPI only or with a mix of MPI and multi-threading, geared towards modern supercomputer architectures.

The computer hardware used for scientific computing has evolved greatly from the early days of Cray vector computers. In the last two decades we have witnessed many radical changes in design. In the 1990s the first Beowulf cluster was constructed, which was little more than a collection of desktop computers, called nodes, linked together by a fast network. Each node had its own private memory space and there was no global view of the total installed memory of the cluster. This cluster represents a so called *distributed memory architecture*.

One of the distinguishing features of the Beowulf cluster was the fact that is was constructed out of commodity parts, opposed to custom built components, as it was done up to that point. The Beowulf cluster gained popularity and, soon, the majority of the world's most powerful parallel computers were based on the Beowulf architecture. In 1993 the Top500 list was started, tracking the world's 500 fastest supercomputing machines (see [1]).

The advent of the Beowulf cluster had a lasting impact on supercomputer design. Modern supercomputers, from companies like Cray or IBM, have a lot of custom designed features, like the network interconnects, the storage subsystems or the packaging of the computer components. However, they also use standard server or desktop processor architectures, like x86 or PowerPC (although slightly modified in certain cases, like the IBM BlueGene series) and, due to the distributed memory spaces they expose to the user, they represent a basically unchanged usage scenario with respect to the Beowulf.

The message passing interface (MPI) library [2] for the Fortran and C programming languages became the standard for programming distributed memory architectures. With MPI, parallelism is implemented by running multiple processes on a parallel computer. These processes have no view of the global memory space of the computer, each having access only to its own memory. Global communication between processes is achieved through sending and receiving messages between individual or multiple processes. Using MPI to solve scientific problems involves partitioning the problem domain across all the processes used for a simulation. Each process operates on its own partition of the problem data and exchanges information with other processes as needed, with the help of the MPI library.

A further technological change took place in the first decade of the 21st century, when processors with multiple cores became common in personal computers (e.g. dual-core laptops, multi-core desktops etc.). These multicore computers differ from clusters in that all the computing units (cores) are able to access the same memory space on the computer (they are refered to as *shared memory architectures*). Multicore technology made its way into computer clusters and supercomputers to the point that, nowadays, most, if not all, entries into the Top500 are large distributed memory machines featuring multicore nodes.

This new architecture, mixing distributed and shared memory spaces, can be programmed efficiently using MPI, by assigning multiple processes to each shared memory region. By design, MPI programming seeks to reduce the amount of communication between processes, which is more expensive than computation in terms of CPU time. This may come with an increase in memory usage, as some data needs to be duplicated on multiple MPI processes. An alternative to this programming model is to mix MPI with another programming model dedicated to shared memory architectures [3]. Within a node, shared memory parallel programming is based on the use of multithreading. There exist multiple implementations of threads that can be used, such as POSIX threads (see e.g. [4], [5]), OpenMP (see e.g. [6], [7]), Intel Thread Building Blocks (see [8]), just to name a few. Program threads are created by processes and represent independent lines of execution. Each thread has its own private memory space for storing its own state, but, in addition to this, all threads created by a process can also access the memory space of the "parent" process, which leads to lower memory requirements, due to reduced data duplication among threads. The downside, however, is that the memory access operations are performed across the entire shared memory space. This can have a detrimental effect to performance in case of non-uniform memory architectures (NUMA), where the shared memory is divided into areas with a considerable difference in access speed. The effect of NUMA architectures on shared memory algorithms is shown in the experiments contained in Chapter 6. As an additional consequence, shared memory algorithms are affected much more by the so called memory wall, which is the ever increasing gap between processor computation speed and memory access speed [9].

A more recent addition to the array of scientific computing machinery are the general purpose graphical processing units (GPGPU). The idea of using the graphical processing units (GPU) inside personal computers for scientific computations is not new (see e.g. [10], [11], [12]), although initially the effectiveness of GPGPU usage was limited by the fact that neither the hardware, nor the software interface really accommodated the implementation of general computational algorithms. Programs had to be implemented in terms of graphical programming operations and data structures. Additionally, GPUs offer much less support for communication and synchronization between parallel execution elements, making it difficult to implement some algorithms such as parallel direct or iterative solvers with high efficiency. With the introduction of the compute unified driver architecture (CUDA) from NVIDIA (see [13]) and the first graphical processors compatible with Stream Computing from AMD (see [14]), it became possible to write general scientific computing software on GPUs using a dedicated programming interface. Currently, scientific computing is an active topic of research and there exist considerable positive results (see e.g. [15], [16], [17]), although some problem domains and algorithms are still less suitable for GPU implementations than others. In Chapter 2 of this thesis, a more detailed description of scientific computing hardware and software is given, including current achievements in GPGPU software.

In our work we focus on the implementation of the finite element method on supercomputers with multicore nodes. This choice of target architecture is not arbitrary. Although there is evidence that GPUs can accomodate certain variants of the finite element method (see e.g. [18], [19]), there are particular elements of our solution process which make multinode-multicore supercomputers a more suitable target. Some of the elements influencing this decision are: the use of very large unstructured meshes, implicit time discretization schemes, linear and nonlinear solvers and finally the use of direct solvers as components of preconditioners.

All the work done in the scope of this thesis has been implemented in two (related) software projects: the LifeV[1] (pronounced "life five") finite element modeling library and the Trilinos Project [20], on top of which LifeV is constructed. Trilinos is a collection of C++ libraries implementing an object-oriented modular software framework for the solution of large-scale scientific problems. Trilinos provides LifeV with lower level components, such as: parallel data structures for sparse matrices and vectors, linear and nonlinear solvers and some common preconditioners.

LifeV is a C++ library providing state of the art mathematical and numerical methods for finite element simulations. It is the product of a joint collaboration between École Polytechnique Fédérale de Lausanne (CMCS) in Switzerland, Politecnico di Milano (MOX) in Italy and Emory University (Sc. Comp) in the U.S.A. While LifeV is mainly a research tool, it has also served as a production library for medical and industrial simulations. LifeV has been used for diverse problems, for example: fluid-structure interaction [21], [22], simulation of orbitally shaken viscous fluids with free surface [23], [24] or multiscale modeling [25], [26].

Domain-decomposition (DD) methods (see [27]) are used in most, if not all, modern parallel implementations of finite element modelling software. Decomposition usually begins at the geometric stage, with the partitioning of the computational mesh. The mesh of the entire problem domain is cut into a given number of parts which are then individually assigned to the MPI processes which perform the simulation. Once the continuous problem is discretized with finite elements, the degrees of freedom (DOF) are distributed across all the processes and the system of equations is assembled, in parallel, using this distribution.

---

[1] LifeV project website - https://cmcsforge.epfl.ch/projects/lifev

In the solver stage, the algebraic additive Scharz (AAS) domain-decomposition preconditioner represents a fundamental component, its performance and scalability are key to the overall performance of the solution process.

In Trilinos, the established approach to construct the preconditioner (see [28]), in a parallel MPI setting, is with a 1-to-1 correspondence between the number of MPI processes, denoted N_P, and the number of AAS subdomains, denoted N_{DD}. It is known that the condition number of the AAS preconditioner degrades with the number of subdomains (see [29]), although this behaviour can be corrected with the addition of a coarse level to the preconditioner formulation, which results in the 2-level AAS preconditioner.

Originally developed before multicore nodes became popular, the Trilnos implementation of the AAS preconditioner has no knowledge of the topology of the underlying hardware. Our goal is to have an arbitrary number of MPI processes assigned per subdomain, mapping the subdomain problems much better to the multicore nodes of the targetted supercomputers and allowing an independent parallelization algorithm to be used at the subdomain level. After we put in place this improved implementation, we study the scalability and performance of the preconditioner in terms of CPU time, by examining different choices and configurations for the AAS subdomain solvers and bringing our own improvements to this area. We also seek to understand how the new implementation of AAS performs both as a global preconditioner for Laplacian and advection-diffusion-reaction (ADR) problems, and as a component within a larger preconditioner framework for Navier-Stokes problems.

Another limitation of the Trilinos implementation of the 2-level AAS is the fact that the coarse level is built without any information about the domain-decomposition on the fine level. The DOFs of the coarse level problem are selected from all the MPI processes. With the new implementation of AAS, it may be possible to reduce the minimum size of the coarse problem by taking into account that multiple MPI processes are part of the same subdomain and selecting fewer DOFs from each process. This could potentially improve the performance of the preconditioner, although this issue has not yet been investigated in our work.

In Chapter 2 we give an overview of the finite element method and we describe in more detail the current state of the art in scientific computing hardware and software. We then discuss some issues related to the serial and parallel implementations of the finite element method. The chapter continues with a review of the current parallel libraries for finite element approximations, before discussing LifeV from the point of view of design and capabilities, as well as of the limitations that LifeV had at the beginning of the thesis.

Chapter 3 introduces the first problem that we studied: the efficient implementation of the AAS domain-decomposition preconditioner. After a brief theoretical overview of the method, we disscuss possible limitations of its existing implementation in the Trilinos libraries. We describe an alternative approach, similar to an earlier attempt described in [30], with the goal of removing some of the existing limitations and improving performance on our target architecture. The new AAS preconditioner has a second level of parallelism represented by parallel subdomain problems. These problems are constructed by grouping together the DOFs corresponding to a given number of connected mesh parts, which produced in the mesh partitioning stage.

The second part of the chapter is focused on the ShyLU preconditioner [31], developed at Sandia National Laboratories. ShyLU is a parallel preconditioner originally designed for electrical circuit simulations. It is based on a non-overlapping partitioning at the matrix level and a Schur complement algorithm relative to the DOFs on the interface of this partitioning. We implement the needed support for using it as an inexact solver for each of the AAS subdomain problems and we study its performance compared to the pre-existing reference configuration of AAS, for Laplacian and ADR problems.

In Chapter 4 we propose a novel use for a preconditioner based on an incomplete QR factorization (IQR), originally introduced in [32] for fluid-structure interaction segregated algorithms; there it is used as a matrix-free preconditioner for the Jacobian problem. We propose to use it as an inexact solver for the Schur complement system in ShyLU. We run a set of tests, benchmarking IQR against the strategies investigated in the previous chapter and finally we discuss the performance increase which we obtained.

The following two chapters focus on two topics that are not directly related to the AAS preconditioner, but which are nonetheless relevant in the context of large scale parallel computing. Chapter 5 covers the preprocessing stage of finite element simulations in LifeV and describes the solutions that we implemented for two related problems: the partitioning of the computational mesh when using a large number of MPI processes and the fast loading procedure of the mesh parts during simulations performed on large scale parallel computing machines. Chapter 6 focuses on the finite element assembly of the linear system matrix. We discuss some issues related to multi-threaded parallel programming and we describe and evaluate the performance, for different problems and different types of finite elements, of a new multi-threaded implementation of the assembly routine. This multi-threaded approach is used to regain parallel efficiency in the assembly stage in cases when due to memory limitations it is not possible to fully utilize all the hardware resources available on a super computer node using only MPI parallelism.

We conclude the numerical experiments of this thesis with a benchmark which involves the solution of the Navier-Stokes equations in a geometry of physiological interest, more exactly, an arterial aneurysm in a human patient. In the benchmark we examine the use of the new preconditioning strategies introduced in Chapters 3 and 4 as components of preconditioners for Navier-Stokes equations, based on approximate block factorizations.

In the final chapter of the thesis we identify all the contributions that we brought in the course of this thesis to the LifeV and Trilinos software projects. Additionally, we issue a set of recommendations based on the results obtained in the previous chapters. Finally, we identify the future directions in which we would like to continue the research started in the course of this thesis.

2 State of the art of finite element software

2.1 Introduction

This chapter begins with an overview of the finite element method. It continues with a description of the current state of the art in scientific computing hardware, as well as a presentation of the relevant issues concerning serial and parallel finite element method implementations. What follows is a review of parallel libraries for finite element approximations, with a subsection dedicated to the use of general purpose graphical processing units (GPGPU) in scientific computing and their usage for finite element modelling. The final part of the chapter introduces the LifeV parallel finite element library, its design, capabilities and limitations at the time when this thesis began.

2.2 The finite element method for the approximation of PDEs

We are interested in the approximation of partial differential equations (PDEs) by the finite element method. As an explanatory example, we will present the case of the Poisson problem in two dimensions.

Let Ω be an open an bounded domain in $\mathbb{R}^d, d = 2, 3, \dots$. The Poisson problem reads: find u, such that

$$\begin{cases} -\Delta u = f \quad \text{in} \quad \Omega \\ \quad u = 0 \quad \text{on} \quad \partial\Omega_D \\ \dfrac{\partial u}{\partial n} = g \quad \text{on} \quad \partial\Omega_N, \end{cases} \tag{2.1}$$

where f is a given function and Ω_D and Ω_N are distinct subsets of the boundary such that $\partial\Omega = \partial\Omega_D \cup \partial\Omega_n$, on which the boundary conditions of Dirichlet and Neumann type, respectively, are imposed. For simplicity, we have restricted ourselves to the case of homogeneous Dirichlet boundary conditions on $\partial\Omega_D$.

2.2.1 Weak formulation

In many cases, such as problems with non-smooth data or geometry, it is necessary to rewrite the problem (2.1) in the weak form, which accepts solutions which don't necessarily satisfy the original equations in a pointwise manner. The weak form is formally obtained by multiplying the equations with a suitable set of test functions, performing integration on the entire domain and using Green's formula for integration by parts in order to reduce the order of differentiation of the solution:

$$\text{find} \quad u \in V, \quad \text{such that} \quad a(u,v) = F(v) \quad \forall v \in V, \tag{2.2}$$

where F is a linear functional which corresponds to the right hand side f and the Neumann boundary conditions:

$$F : V \to \mathbb{R}, \quad F(v) = \int_\Omega f v + \int_{\partial\Omega_N} g v, \tag{2.3}$$

and a is a bilinear form which corresponds to the Laplacian operator:

$$a : V \times V \to \mathbb{R}, \quad a(u,v) = \int_\Omega \nabla u \nabla v. \tag{2.4}$$

The test space V is chosen as:

$$V = H^1_{\partial\Omega_D}(\Omega) = \left\{ v \in H^1(\Omega) : v|_{\partial\Omega_D} = 0 \right\}, \tag{2.5}$$

where H^1 is the Sobolev space of order 1 over Ω.

Under suitable assumptions on Ω and provided that $f \in L^2(\Omega)$ and $g \in H^{-1/2}(\partial\Omega_N)$, the solution to the weak problem exists and is unique. The proof is based on the Lax-Milgram Lemma [33]. In particular, it assumes that $F(\cdot)$ is a linear continuous functional, the bilinear form $a(\cdot,\cdot)$ is continuous, i.e.

$$\exists \gamma > 0 : |a(w,v)| \le \gamma ||w|| ||v|| \quad \forall w, v \in V, \tag{2.6}$$

and coercive, i.e.

$$\exists \alpha > 0 : a(v,v) \ge \alpha ||v||^2 \quad \forall v \in V. \tag{2.7}$$

Here, $||\cdot||$ denotes the norm of V. Additionally, there exists the following bound on the solution:

$$||u|| \le \frac{1}{\alpha} ||F||_{V'}, \tag{2.8}$$

where V' is the dual space of V.

2.2.2 Finite dimensional approximation

Let us consider that the domain $\Omega \subset \mathbb{R}^3$ has a polyhedral shape. We can define a tetrahedral mesh $\mathcal{T}_h$ of Ω composed of a set of non overlapping tetrahedra $K_i, i = 1, ..., N_k$. We require $\mathcal{T}_h$ to be conforming, i.e., two neighbouring tetrahedra can only have a common vertex, or a common edge, or a common side. The following parameters are useful to describe the mesh. We define

$$h_K = diam(K), \quad \text{with} \quad diam(K) = max_{x,y\in K}|x-y|, \tag{2.9}$$

which represents the diameter of an element K of $\mathcal{T}_h$. We can also define the value: $h = max_{K\in\mathcal{T}_h} h_K$ which is sometimes called mesh size of $\mathcal{T}_h$. Finally, we define:

$$\rho_K = \sup\{diam(S) : S \quad \text{is a ball contained in} \quad K\}. \tag{2.10}$$

Using the parameters h_K and ρ_K it is possible to state the following definition: a family of triangulations $\{\mathcal{T}_h : h > 0\}$ is regular if there exists a constant $\delta > 0$ independent of h such that:

$$\frac{h_K}{\rho_K} \leq \delta, \quad \forall K \in \mathcal{T}_h. \tag{2.11}$$

Condition (2.11) ensures that the tetrahedra are not too streched in any direction.

Considering $\mathbb{P}_r$ the space of polynomials of degree r, we introduce the finite element space:

$$X_h^r = \left\{v_h \in C^0(\overline{\Omega}) : v_h|_K \in \mathbb{P}_r, \forall K \in \mathcal{T}_h\right\}, \quad r = 1, 2, ..., \tag{2.12}$$

which is the space of globally continuous functions that are polynomials of degree r on each element of $\mathcal{T}_h$. Additionally, we can define the space:

$$\overset{\circ}{X}{}_h^r = \left\{v_h \in X_h^r : v_h|_{\partial\Omega} = 0\right\}. \tag{2.13}$$

In general, when $r = 1, 2, ...$ we speak of P1, P2,... finite elements, respectively.

It is possible to obtain a finite dimensional approximation of the weak problem (2.2) by considering a subspace V_h of V:

$$V_h = \overset{\circ}{X}{}_h^r \subset V, \quad \dim V_h = N_h < \infty \quad \forall h > 0. \tag{2.14}$$

The Galerkin problem that approximates the weak problem is written as:

$$\text{find} \quad u_h \in V_h, \quad \text{such that} \quad a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h. \tag{2.15}$$

Let $\{\Phi_j, j = 1,2,...N_h\}$ be a basis for the space V_h, such that:

$$v_h = \sum_{j=1}^{N_h} v_j \Phi_j, \quad \forall v_h \in V_h. \tag{2.16}$$

Thanks to the linearity of a and F, (2.15) is equivalent to

$$a(u_h, \Phi_h) = F(\Phi_h), \quad i = 1,2,...N. \tag{2.17}$$

In the Galerkin approximation, the trial space (the solution u_h belongs to) is the same as the test space V_h. Consequently, we can write u_h as:

$$u_h(x) = \sum_{j=1}^{N_h} u_j \Phi_j(x), \tag{2.18}$$

where $u_j, j = 1,2,...N_h$ are unknown coefficients. Using (2.18) we can rewrite (2.17) as:

$$\sum_{j=1}^{N_h} u_j a(\Phi_j \Phi_i) = F(\Phi_i), \quad \forall i = 1,2,...,N_h, \tag{2.19}$$

that is:

$$\sum_{j=1}^{n} u_j \int_\Omega \nabla\Phi_j \cdot \Phi_i = \int_\Omega \Phi_i f + \int_{\partial\Omega_N} \Phi_i g_N \quad \forall \quad i = 1,...,N_h. \tag{2.20}$$

It is possible to write (2.19) as a linear system:

$$A\mathbf{u} = \mathbf{f}, \tag{2.21}$$

where the matrix A, called the stiffness matrix has the entries $a_{ij} = a(\Phi_j, \Phi_i)$, the vector $\mathbf{f}$ has the components $f_i = F(\Phi_i)$ and the solution vector $\mathbf{u}$ is composed of the unknown coefficients u_j.

We refer to [33] for properties of the Galerkin approximation. The stiffness matrix A associated with an elliptic problem is positive definite. In addition, for symmetric bilinear form $a(\cdot,\cdot)$, the stiffness matrix is also symmetric. Consequently, the linear system that results from the Galerkin approximation of an elliptic problem can be solved using either direct methods like Cholesky factorization [34], or iterative methods like the conjugate gradient method [35]. For more details on the solution of the linear system of equations we refer to section 2.4 of this chapter.

The Lax-Milgram Lemma gives the conditions under which the solution to the weak problem exists and is unique. This conditions also hold for the Galerkin approximation, due to the fact that the space V_h is a closed subspace of the Hilbert space V and the bilinear form $a(\cdot,\cdot)$ and the functional $F(\cdot)$ are unchanged with respect to the weak formulation.

The method is stable with respect to h and the solution has the following upper bound:

$$||u_h||_V \leq \frac{1}{\alpha}||F||_{V'}. \tag{2.22}$$

The Galerkin approximation is strongly consistent, it's solution u_h is the orthogonal projection on V_h of the exact solution u:

$$a(u - u_h, v_h) = 0 \quad \forall v_h \in V_h \tag{2.23}$$

Regarding convergence, the following inequality holds:

$$||u - u_h||_V \leq \frac{\gamma}{\alpha}\text{inf}_{w_h \in V_h}||u - w_h||_V \leq C\frac{\gamma}{\alpha}h^r|u|_{H^{r+1}(\Omega)}, \tag{2.24}$$

for a suitable constant $C > 0$, with $|u|_{H^p(\Omega)}, p \geq 1$ denoting the semi-norm of u of order p.

2.2.3 Assembly of the linear system of equations

In this section we describe the process to generate the linear system of equations associated with the Galerkin approximation.

For simplicity, we describe the choice $r = 1$, i.e. piecewise linear finite elements: $V_h = \overset{\circ}{X}{}_h^1$. Starting from the union of vertices of the tetrahedra of the mesh $\mathscr{T}_h$, it is possible to define a basis for V_h. Each vertex, say $\vec{x}_i$, in this case also called a node, is assigned a function Φ_i which is linear in all the tetrahedra, is equal to one on that vertex, and is equal to zero on all other vertices. The support of the defined basis function is equal to the union of tetrahedra which have that vertex in common.

It is possible to rewrite the terms of (2.20) as elementwise sums. For example:

$$\sum_{j=1}^{n} u_j \int_{\Omega} \nabla\Phi_j \cdot \Phi_i = \sum_{j=1}^{n} u_j \left\{ \sum_{\Delta_k \in T_k} \int_{\Delta_k} \nabla\Phi_j \cdot \nabla\Phi_i \right\}. \tag{2.25}$$

Thanks to the chosen basis, the indices i, j and k for which the integral over Δ_k is different from zero are limited to the patch of elements where Φ_i and Φ_j share their support. The two major consequences are that the matrix A is sparse and that the computation of the entries of A can be done on a restricted number of nodes. In the case of the $P1$ basis functions, we only have three functions that are non-zero on the element. Restricting these global basis functions to the element Δ_k gives us the local element basis set:

$$\Theta_j := \left\{\psi_{k,1}, \psi_{k,2}, \psi_{k,3}\right\}. \tag{2.26}$$

```
Data: $A$
for $k \leftarrow 1$ to $nElements$ do
    Data: $Jac \leftarrow$ ComputeJacobian($k$)
    Data: $elementalA \leftarrow$ CreateElementalMatrix()
    for $i \leftarrow 1$ to $nTestDof$ do
        for $j \leftarrow 1$ to $nTrialDof$ do
            for $q \leftarrow 1$ to $nQuadPoints$ do
                $elementalA(i, j)$ = value($q$, $i$, $j$) * $Jac$ * weight($q$)
    Data: currentIndices $\leftarrow$ GetGlobalIndices($k$)
    InsertElementalToGlobal($elementalA$, $A$, currentIndices)
```

Algorithm 1: Pseudo-code describing the assembly process of the stiffness matrix.

The next step is to perform a mapping from a reference element Δ_* to a current element Δ_k, which allows us to rewrite the expression for the coefficients of the elemental stiffness matrix in terms of this mapping:

$$\begin{aligned} a_{ij}^{(k)} &= \int_{\Delta_k} \frac{\partial \psi_{k,i}}{\partial x}\frac{\partial \psi_{k,j}}{\partial x} + \frac{\partial \psi_{k,i}}{\partial y}\frac{\partial \psi_{k,j}}{\partial y} \mathrm{d}x\mathrm{d}y \\ &= \int_{\Delta_*} \left\{ \frac{\partial \psi_{*,i}}{\partial x}\frac{\partial \psi_{*,j}}{\partial x} + \frac{\partial \psi_{*,i}}{\partial y}\frac{\partial \psi_{*,i}}{\partial y}\frac{\partial \psi_{*,j}}{\partial y} \right\} |J_k| \mathrm{d}\xi \mathrm{d}\eta, \quad i, j = 1, \dots, n, \end{aligned} \tag{2.27}$$

where (x, y) are the global coordinates, (ξ, η) are the local coordinates in the reference element, $\psi_{k,j}$ are local basis functions in the current element Δ_k, $\psi_{*,j}$ are local basis functions in the reference element Δ_*, while $|J_k|$ is the determinant of the Jacobian of the mapping from the reference element to the current one.

The numerical computation of the integrals in (2.27) is performed using quadrature formulae:

$$\int_{\mathscr{D}} f(\mathbf{x}) d\mathbf{x} \approx \sum_{i_q=1}^{N_{qn}} f(x_{i_q}) w_{iq}, \tag{2.28}$$

where $\mathscr{D}$ represents the region over which the integration is performed, N_{qn} represents the number of quadrature nodes, x_{iq} represent the coordinates of the quadrature nodes, while w_{iq} represent the weights associated with the corresponding quadrature nodes. The accuracy of quadrature formulae increases with the number of quadrature nodes used, although this increases the computational cost as well. For an overview of numerical integration using quadrature rules we refer to [36]. Guassian integration formulae, which, for a given number of quadrature nodes, achieve maximal accuracy are described in [33].

A straight-forward approach to implement the assembly of the stiffness matrix is by using nested loops, over the elements, then the degrees of freedom of the test space, then the degrees of freedom of the trial space (the solution u_h belongs to) and finally over the quadrature points. This algorithm is presented in pseudo-code in Algorithm 1.

```
Data: rhs
for $k \leftarrow 1$ to $nElements$ do
    Data: $Jac \leftarrow$ ComputeJacobian($k$)
    Data: elementalRhs ← CreateElementalVector()
    for $i \leftarrow 1$ to $nTestDof$ do
        for $q \leftarrow 1$ to $nQuadPoints$ do
            elementalRhs(i) = value($q$, $i$) * $Jac$ * weight($q$)
    Data: currentIndices ← GetGlobalIndices($k$)
    InsertElementalToGlobal(elementalRhs, rhs, currentIndices)
```

Algorithm 2: Pseudo-code describing the assembly process of the right hand side vector.

The process to compute the right hand side (Algorithm 2) is analogous to the stiffness matrix computation, although it involves one fewer nested loops, as there is only one loop over the local degrees of freedom.

For the case of different mesh types, like quadrilateral, tetrahedral or hexahedral, as well as for higher order basis functions like piecewise quadratic functions, a similar aproach can be used to derive the expression of the coefficients of the elemental stiffness matrices. For a more in-depth treatment, we refer to [37].

Neumann boundary conditions are included in the weak form and are taken into account when doing the computation of the right hand side vector. For Dirichlet boundary conditions, there are multiple approaches. One can remove the degrees of freedom associated with Dirichlet boundary conditions and include these boundary conditions in the right hand side, or one can retain these degrees of freedom, set the diagonal value of the matrix to one, offdiagonal values to zero and set the corresponding element of the right hand side vector to the value of the Dirichlet boundary condition.

This elementwise approach is appropriate to describe, more formally, the assembly of the linear system of equations, although it is also usable in practice. However, other strategies can be encountered that seek to optimize the performance of this basic approach, involving clustering elements together or looping over degrees of freedom instead of looping over elements, or different parallel approaches.

2.3 Scientific computing hardware and software tools

In order to describe the implementation issues concerning the finite element method, an overview of scientific computing hardware is required. This section presents the architectures which are currently most common and the associated software tools used to program these machines.

The supercomputers that are in use today are very different to the early designs from Cray Supercomputers. These computers were vector machines where parallelism was implemented by performing the same operation simultaneously on large data sets. This approach is commonly refered to as *single-instruction multiple-data* (SIMD). A shift in the design strategy of supercomputing machines occured in the late 1990s, with the appearance of the Beowulf cluster. Contrary to previous machines constructed with custom made components, the Beowulf cluster is composed of a number of computers (nodes), each with their own central processing unit (CPU) and memory, which are all connected together in a network. Since the memory of the machine is not presented to the user or programmer as a single unified space, this type of machine is called a *distributed memory architecture*, and programs use some form of message passing to exchange data between the nodes of the cluster. The message passing interface (MPI) [2] has become a standard and it is the most widely used software toolkit for programming supercomputers, with computer vendors each providing implementations of the standard optimized for their own machines.

The shift from custom design to off-the-shelf parts was hugely successful, as Beowulf clusters (now more tightly integrated through the use of specialized motherboards or network hardware) have grown to represent the majority of the Top500; this list, which is compiled yearly, started in 1993 and represents a global ranking of the world's most powerful supercomputer machines based on the High-Performance LINPACK benchmark for distributed memory machines [38]. Even supercomputers from companies like Cray and IBM, which contain custom network interconnects, still share a lot with the Beowulf design: distributed memory spaces and standard CPU architectures like x86 or PowerPC, which are used in workstation or server computers.

Early distributed memory cluster nodes were single processor computers. At this stage MPI was perfectly mapped to the architecture. Each node would run one process and the MPI library would handle communication between these processes. A second stage of the evolution of supercomputing machines arrived with the popularization of multi-core processors. Today, most personal computers have processors with at least 2 cores and it is not uncommon in workstations to have up to 48 cores and the trend is to increase the number of cores. Since all the cores can use the global memory installed in the computer this is refered to as *shared memory architecture*. This type of architecture has made its way also into supercomputers. A typical hybrid configuration for a modern supercomputer is a few number of interconnected nodes, each with multiple cores available. It is important to note the distinction between core and process. A core represents a hardware unit of a CPU, capable of executing one or more processes simultaneously. Process is a purely software concept: a program that is executing on a computer represents a process.

Such an architecture with two levels of parallelism can still be programmed using MPI, with multiple processes running on each node. There exist alternative approaches to programming for this architecture. Since the cores on a node can all use the same memory space, there is no need to have MPI communication inside a node. A single MPI process is executed per node, and the available node level parallelism is exploited with multiple threads using toolkits and libraries like POSIX Threads and OpenMP. A thread, also called lightweight process, is

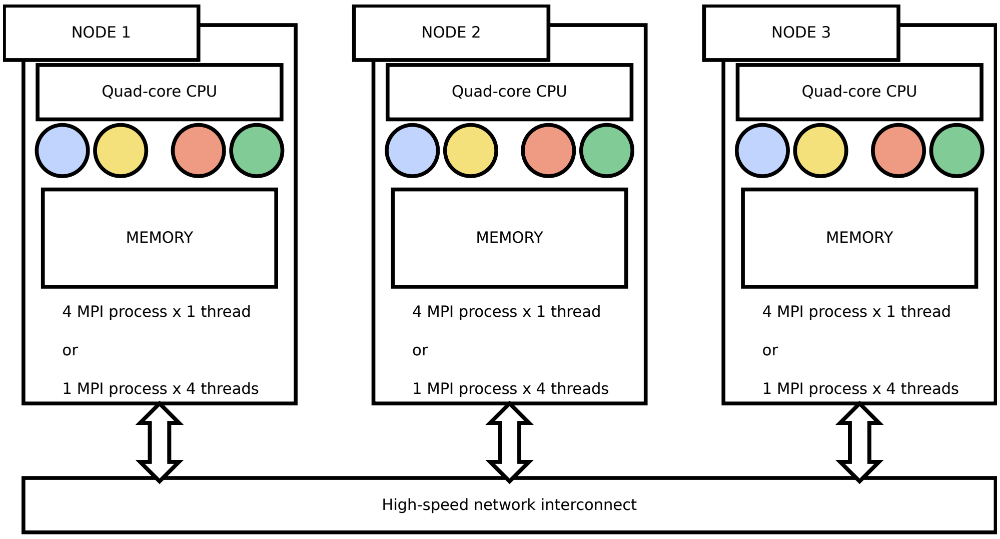


Figure 2.1: Simplified diagram depicting the structure of a hybrid cluster composed of 3 nodes. Each node has a processor with 4 cores which is capable of hosting either 4 single-threaded MPI processes or a single MPI process that spawns 4 threads.

an independent line of execution created by a process. The process can create multiple threads but the threads can not exist beyond the lifetime of the owner process. Although threads can each have private data, all the threads of a process can access the process' global memory space and use it to exchange information. One advantage of this hybrid approach to programming is the fact that it may be easier to express certain algorithms in terms of parallel operations to a shared global memory. As this is only possible at node level, MPI must still be used between nodes, so the presence of two programming models in the same code base could make the software more difficult to maintain or to understand. A graphical description of the two possibilities is shown in Figure 2.1.

A final class of machines, which is more recent are graphical processing units (GPU). The GPU is the specialized hardware of a computer that handles the tasks related to drawing 2D graphics, accelerating the rendering of 3D scenes and performing video output. Unlike a CPU, which has a small number of powerful cores, each optimized for single threaded performance, a GPU is composed of many simple processor cores and is designed for parallel threaded performance. This comes with restrictions, however, as the hardware is very limited when it comes to the synchronization of threads or the exchange of information between them. It is well suited for performing many simple calculations in parallel, in a SIMD fashion, but this has to be done with little or no communication between the parallel threads.

Initial attempts to perform general scientific computations on GPUs using graphical programming tools showed that these devices are attractive platforms for certain types of applications. The programming languages have improved and now make it easier to implement scientific applications on GPUs. There even exist clusters of computers equipped with GPUs, where the computational power of GPUs is complemented by the large amount of memory available on distributed memory machines. A more detailed description of GPUs for scientific computing and finite element modeling is provided in section 2.5.1.

2.3.1 Vectorization

Distributed and shared memory architectures are effectively programmed using MPI and multithreading, respectively. However, all modern computer hardware offers an additional level of parallelism, in the form of vector arithmetic units, i.e. the capability of executing the same instruction simultaneously on multiple data elements (SIMD).

The use of vector hardware in scientific computing can be traced back to early supercomputer designs, but it was with the Cray 1 supercomputer that first introduced a processor with good scalar performance as well as vector processing capabilities. The Cray 1 was followed by the Cray-XMP, which expanded the design of the Cray 1 with multiple vector processors and is considered to be the first *parallel vector processor*.

Vector processing would make its way, in the second half of the 1990s, into mainstream desktop processors in the form of the MMX instructions, which provided vector operations for integer data. The MMX instructions could process either two 32bit integers, four 16bit integers or eight 8bit integers at the same time. The 3DNow! and SSE instruction sets, from AMD and, respectively, Intel, extended the vector operations to floating point data. In the most recent version of the SSE instruction set, SSE4.2, either two double precision or four single precision floating point variable can be processed at the same time. The follow-up to SSE was the AVX instruction set from Intel, which extended the vector width to four double precision or eight single precision floating point variables. The QPX vector instruction set supported by the BlueGene/Q architecture from IBM also supports four double precision floating point variables [39].

There is a continuing trend to increase the width of vector processing units. Recent developments such as the Intel Many Integrated Core (MIC) architecture (see [40]) brings the vector width to 16 double precision variables. The presence of wide vector units together with a very large number of processor cores makes the Intel MIC very similar to GPUs, from a hardware and programming perspective.

The vector processing hardware can be used with two different approaches. The first is to explicitly implement the vector parallelism through the use of special compiler instructions[1]. These instructions offer the most control over which part of the code is vectorized, but their use requires a considerable amount of programming effort.

The alternative is to rely on the compiler to automatically generate vectorized code. Historically, the automatic vectorization capabilities of compilers have been less reliable than coding vectorization by hand, using compiler intrinsics. However, as automatic vectorization greatly

---

[1] Intel Intrinsics Guide - http://software.intel.com/en-us/articles/intel-intrinsics-guide

decreases the effort on the part of the programmer, it represents a very important focus for compiler developers. For an overview of the current performance of automatic vectorization techniques, see [41], [42].

The LifeV library does not make use of compiler intrinsics for vectorization. The vector processing hardware is exploited through optimized third-party libraries, in areas where performance is critical, such as dense linear algebra routines. Although there exists potential for some performance increase by applying vectorization techniques directly into certain parts of the LifeV code, this does not represent the focus of this thesis.

2.4 Serial and parallel implementation

The serial implementation of finite element software is summarized in Figure 2.2. The usual workflow begins with loading a polyhedral mesh from disk into memory, as an appropriate data structure. The next step is the finite element loop: given a certain choice of finite element discretization, the list of all the elements in the mesh (and associated degrees of freedom) is parsed and the coefficients a_{ij} and f_i of the stiffness matrix and right hand side vector, respectively, are computed (see Algorithms 1 and 2). At the end of this stage, the stiffness matrix A and the right hand side vector f are given to the linear system solver. In a serial setting, this solver can use a sparse implementation of direct method like the Cholesky or LU factorization. There exist many variants of these methods, such as the multi-frontal method used in the very robust and efficient solver packages UMFPACK [43] and MUMPS[44]. Due to the large memory requirements of direct methods, when attempting to solve larger two-dimensional and three-dimensional problems, it is common to use iterative solvers like preconditioned Conjugate Gradient or GMRES [35]. When the problem is nonlinear, a nonlinear method such as Newton iterations needs to be used [45]. In this case, at each nonlinear iteration the linear Jacobian system, $J_F(x_n)(x_{n+1}-x_n) = -F(x_n)$, has to be solved. The final step, after the solution of the linear system of equations is composed of any computations that need to be performed on the solution, such as error computation, or simply comprises the export of the solution

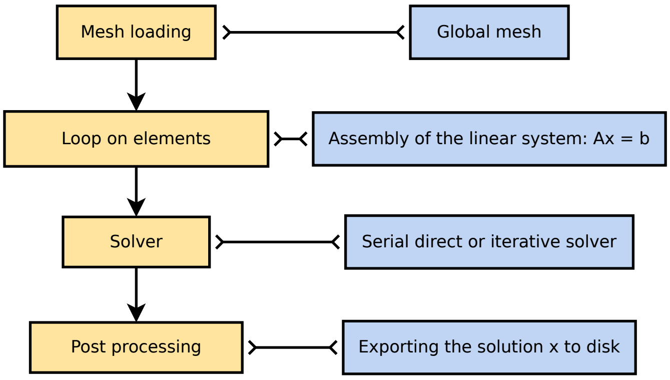

Mesh loading
Global mesh
Loop on elements
Assembly of the linear system: Ax = b
Solver
Serial direct or iterative solver
Post processing
Exporting the solution x to disk

Figure 2.2: The block diagram describing the steps of a serial finite element method implementation.

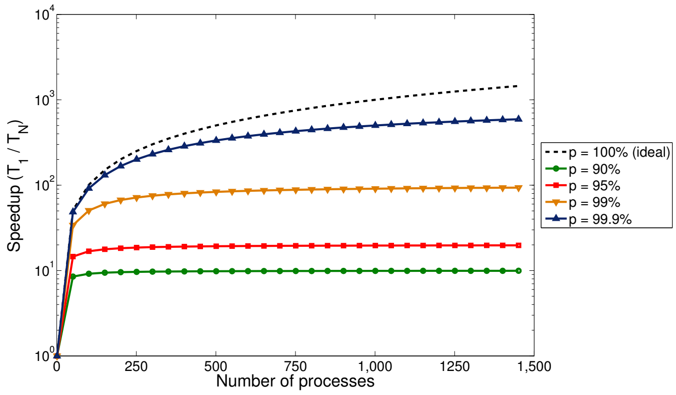


Figure 2.3: Ideal parallel speedup, according to Amdahl's law, with respect to the number of processes used, for different values of the parallel fraction of execution time p.

back to disk.

Parallelism and scalability

There exist two possibly complementary goals when developing the parallel implementation for a numerical simulation. The first is to be able to solve the problem considered in a shorter amount of time, by making use of additional parallel hardware resources. The measure of the efficiency of this approach is represented by the speedup, the ratio between the total run-time in the serial case (T_1) and the run-time in the parallel case (T_N):

$$S_N = \frac{T_1}{T_N}, \tag{2.29}$$

where the index N represents the number of processors used in the parallel case. In the ideal case, the speedup would always be equal to the number of processors N (for example: a program should finish four times faster, when using four processors instead of a single one), but in practice this is never the case. Amdahl's law [46] tries to quantify the speedup obtained in practice:

$$S_N = \frac{1}{(1-P)+\frac{P}{N}}. \tag{2.30}$$

Here S is the speedup, N is the number of processors used and P is a fraction which represents the parallel portion of a program. Computing a precise value of the parameter P is not possible, but the law is nonetheless useful to show that even for algorithms that are almost 100% parallelizable there exists a maximum achievable speedup which is independent of the number of processors that is used. The measure of a program's speedup for a given problem size using an increasing number of processors is called *strong scalability*.

In Figure 2.3 we see that even with a parallel portion of 99.9%, strong scalability is limited when increasing the number of processors beyond 1000, unless the parallel portion of the program also increases with the number of processors. At $p = 95\%$ the maximum speedup obtained is less than 100, using 200 processes or more, while at $p = 99.9\%$ the speedup is approximately 800, using 200 processes more. If the work is not constant, Amdahl's law does not apply any longer.

It should be noted that there also exist cases when superlinear strong scalability can be observed. This can happen when dividing the work load into smaller and smaller parts allows each part to fit completely into the smaller, but faster, cache memory.

A different concept, called *weak scalability*, has been introduced by Gustafson in [47]. Instead of keeping the workload constant, it is assumed that the work is proportional to the number of processors. Parallel hardware resources are not used to solve the same problem in a shorter time, but they are used on a larger problem with the aim of solving it in the same amount of time. This approach is described by Gustafson's law which redefines speedup as:

$$S(N) = p(N-1)+1, \tag{2.31}$$

where N is the number of processors and p is the parallel fraction of the program. This new definition of speedup works under the assumption that the problem size is scaled *linearly* with the available compute resources available and is computed with respect to the hypothetical time to solve the scaled problem on a single processor. In this context the ideal speedup $S(N) = N$ is never reached, but there is a clear asymptote pN.

In the context of large scale parallel computing machinery the metric of weak scalability has been essential in the development of new algorithms and approaches to computational science. Although achieving weak scalability at large scale is possible, there are certain limitations that prevent perfect scalability, like the communication cost associated with a certain parallel configuration. Depending on the algorithm, the time spent transfering information between processors could actually grow to dominate, or at least represent a considerable fraction of the total run-time. This would mean that p is not constant any longer, but depends on N.

Parallel implementations

Algorithms based on domain or mesh partitioning are essential to all highly parallel finite element implementations. They are useful in that the full resources of a parallel computing machine are used, ensuring that the programs have weak scalability.

A parallel implementation follows the same general steps as the serial implementation, although it involves additional concepts and issues which need to be managed to ensure proper results. Figure 2.4 contains a block diagram of the parallel finite element process.

The parallel strategy which is described here is based on MPI, since it is known to allow very

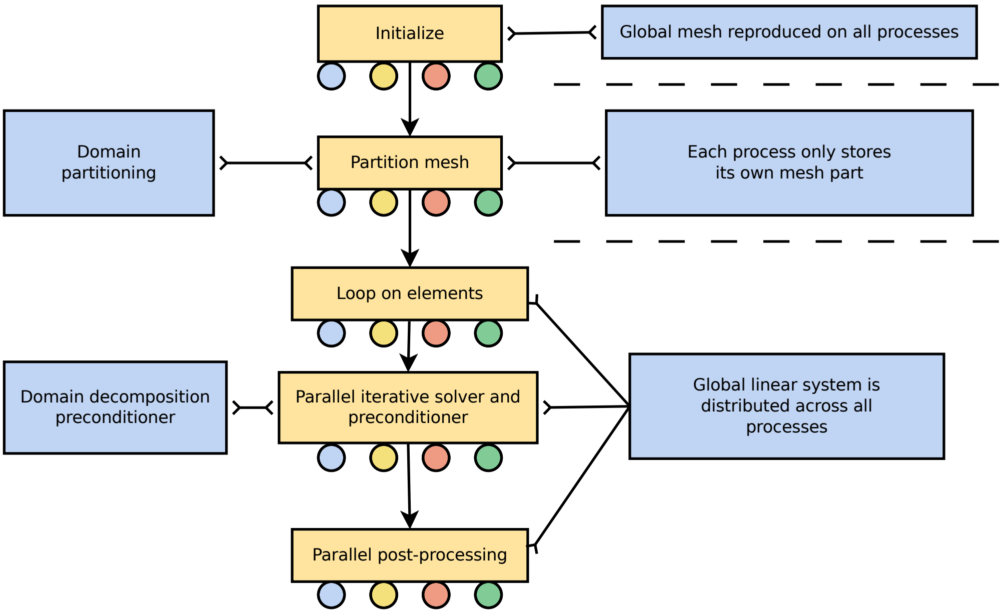

Initialize
Global mesh reproduced on all processes
Domain partitioning
Partition mesh
Each process only stores its own mesh part
Loop on elements
Domain decomposition preconditioner
Parallel iterative solver and preconditioner
Global linear system is distributed across all processes
Parallel post-processing

Figure 2.4: The block diagram describing the typical steps of a parallel finite element method implementation. The coloured circles represent the four parallel processes in this example.

efficient implementations on large distributed memory machines, such as the benchmarks used for the Top500. Due to the fact that the expensive communication between processes needs to be explicitly performed by the programmer, it leads to a development process focused on minimizing the amount of communication that is performed. Additionally, MPI is a very mature standard and at this point represents by far the most common and efficient approach used in parallel finite element software.

A typical simulation begins with creating the desired number of MPI processes. Since we are mostly interested in situations like cardiovascular flows, where the geometry is not described by a computer aided design (CAD) program, the mesh can not be created online, during the simulation. Generally, the mesh is unstructured and needs to be partitioned among the processes. If the mesh partitioning is performed online, during the parallel simulation, there is the need to read the global unpartitioned mesh on each MPI process. This represents a serious memory bottleneck. As we see in Chapter 5, moving the mesh partitioning to an offline stage, performed before the online stage can remove this bottleneck.

Mesh partitioning is performed by dividing the elements of the mesh among all the processes. Partitioning is usually done by first building the dual graph of the mesh which is then cut into parts using a graph partitioner package like ParMETIS [48] or SCOTCH [49]. The algorithms employed by the graph partitioners aim to reduce the total interface between mesh parts, in order to reduce the amount of communication between processes. The original uncut mesh can be deleted from memory, since from this moment on, each MPI process will only operate on it's own set of elements.

```
Data: A
for $k \leftarrow 1$ to $nLocalElements$ do
    Data: $kG \leftarrow$ LocalToGlobal($k$)
    Data: $Jac \leftarrow$ ComputeJacobian($kG$)
    Data: $elementalA \leftarrow$ CreateElementalMatrix()
    for $i \leftarrow 1$ to $nTestDof$ do
        for $j \leftarrow 1$ to $nTrialDof$ do
            for $q \leftarrow 1$ to $nQuadPoints$ do
                $elementalA(i, j)$ = value($q$, $i$, $j$) * $Jac$ * weight($q$)
    Data: currentIndices $\leftarrow$ GetGlobalIndices($kG$)
    InsertElementalToGlobal($elementalA$, $A$, currentIndices)
GlobalAssemble()
```

Algorithm 3: Pseudo-code describing the assembly process of the stiffness matrix in the parallel case. The parallel setting requires that the local element index k is translated to a global element index kG, unique across all processes. The final step, GlobalAssemble, performs the communication of partial coefficients associated with degrees of freedom on the interfaces between mesh parts.

Having divided the list of elements among them, each process proceeds to loop over its own elements and computes the coefficients of the stiffness matrix and right hand side vector that are associated with the elements that they own (Algorithms 3 and 4).

This part is highly scalable, since the majority of the degrees of freedom owned by each process are located in the interior of the mesh parts in which case the owner process computes the associated coefficients in the linear system. Coefficients associated with degrees of freedom located on the interfaces between mesh parts are computed as the sum of partial values coming from all the processes which own elements that contain the respective degree of freedom. This sum involves MPI communication and is represented in Algorithms 3 and 4 by the call to the GlobalAssemble routine.

Each process maintains a set of associative arrays, or maps, which record the list of degrees of freedom that are owned by the process. Figure 2.5 shows the distribution of degrees of freedom across multiple processes for a simplified case of a parallel sparse matrix. The parallel implementations of these data structures provide additional functionality, such as querying which process owns a given set of degrees of freedom. The maps are also used to construct the parallel distributed matrix and vector data structures that are used in the linear solver. The matrix data structure usually employs a compressed sparse row storage strategy, with the rows distributed across processes.

In the parallel case, the solution of the linear system of equations is handled by parallel implementations of linear solvers. While it is technically possible to use parallel direct solvers like MUMPS [44] or SuperLU_Dist [50], by far the most efficient solvers are the iterative ones with parallel preconditioners based on domain-decomposition (DD) [51] or multigrid (MG)

```
Data: rhs
for $k \leftarrow 1$ to $nLocalElements$ do
    Data: $kG \leftarrow$ LocalToGlobal($k$)
    Data: $Jac \leftarrow$ ComputeJacobian($kG$)
    Data: elementalRhs ← CreateElementalVector()
    for $i \leftarrow 1$ to $nTestDof$ do
        for $q \leftarrow 1$ to $nQuadPoints$ do
            elementalRhs($i$) = value($q$, $i$) * $Jac$ * weight($q$)
    Data: currentIndices ← GetGlobalIndices($kG$)
    InsertElementalToGlobal(elementalRhs, rhs, currentIndices)
GlobalAssemble()
```

Algorithm 4: Pseudo-code describing the assembly process of the right hand side vector in the parallel case. The parallel setting requires that the local element index k is translated to a global element index kG, unique across all processes. The final step, GlobalAssemble, performs the communication of partial coefficients associated with degrees of freedom on the interfaces between mesh parts.

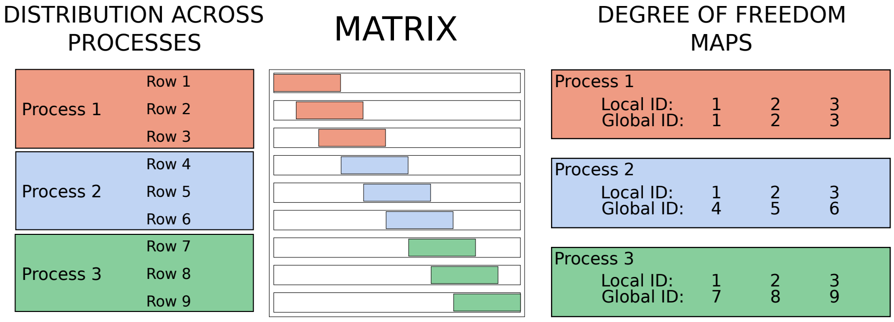


Figure 2.5: The distribution of the rows of a parallel sparse matrix across three processes. According to this distribution, each process maintains a degree of freedom map which maintains the association between local and global DOF numbering.

[52]. The next section gives an overview of the different methodologies and libraries that offer such components for use in finite element software.

Typically, the parallel solvers provide a unified interface for serial and parallel use. The user or programmer just connects the data structures that represent the linear system with the parallel solvers and all the parallel implementation details, like communication between processes, are hidded in the implementation. In most cases, application code that is built on third party libraries that hide the MPI interface is nearly identical in the serial and parallel cases.

For parallel input and output (I/O) of data, most applications use a high performance I/O library, like HDF5 [53]. Again, such libraries seek to hide the complexity of parallel disk read and write operations, while achieving a high level of performance.

In parallel finite element software, one can encounter different performance bottlenecks in different stages of the simulation pipeline. Different ways to implement geometric domain partitioning can lead to excessive memory usage, while the speed of parallel I/O greatly affects the performance of this stage and of the post-processing stage. Many institutions in the field of high performance computing dedicate a large amount of resources to the maintainance and the development of highly efficient parallel file systems and storage hardware.

Scalable and robust solver and preconditioning strategies represent a very active and challenging field of research. Preconditioners for more complex problems, such as saddle-point problems, remain very much an open research direction.

The final section of this chapter will continue the ideas from this section and will give an overview of the design and implementation of the LifeV finite element modelling library. The next chapter of the thesis will provide a discussion on parallel preconditioners based on domain decomposition.

2.5 Parallel libraries for finite element modelling

Finite element modelling software is an established tool in engineering. Typically such software is closed source, commercial, such as COMSOL [54], ANSYS [55] or Abaqus [56]. These packages are very robust and are able to perform simulations in multiple problem domains, such as mechanics, fluid dynamics, electromagnetics or coupled multiphysics problems. While limited support for parallelism is present in recent version of these applications, the focus is more on powerful workstations and small compute clusters, rather than large supercomputers.

In the research domain, open source project offerings are the norm and in some cases they are designed to make use of large high performance computing architectures. There is also a greater variety in the interface that these packages offer, ranging from lower level library collections, to integrated environments for partial differential equation (PDE) solution and simulations, finally to applications with a graphical user interface (GUI) that emulate the usage experience of the commercial software. We will next give a short overview of the most important open source projects in each one of these cathegories.

Frameworks for FEM applications

Software frameworks do not represent complete integrated applications for FEM. Instead, they are collections of component libraries that are designed to be used together in order to build such an application. They include: tools and structures for mesh handling, load balancing and transformations, for high-performance linear algebra, for the assembly of the system of linear equations, solvers and preconditioners and finally tools for post-processing.

In this category, there are two mature and robust major players which include parallelism through MPI, the Trilinos Project [20], a collection of C++ libraries and PETSc [57], written in

C. They are both developed at national research laboratories in the United States of America and are used by a large number of research projects around the world. They support large-scale distributed memory parallelism through MPI and, in more recent versions, also support shared memory parallelism. While Trilinos is more modular in design, one can say that both solutions offer equivalent features in different programming languages. Both projects offer strong support for FE modelling, however the interfaces that they expose are general enough to allow the use of other methodologies, such as finite volume or finite differences.

Integrated simulation environments

Integrated simulation environments represent an intermediate step between frameworks and applications. They offer a more restrictive and higher level programming interface than framework. The interface is tailored to focus more on the modelling parameters of the simulations, hiding low level details from the user. This user interface is often provided in a higher level programming language, such as Python [58]. The definition of a domain specific language (DSL) for describing the weak formulation of the problem is a very convenient feature offered by these environments and allow the user to define his problem in a language that is closer to the underlying mathematics than regular C, C++ or Python would permit. From the point of view of implementation, these DSL can be provided through template meta-programming in the C++ language or through code generation at run-time, as is the case of Python interfaces.

Some projects, such as FEniCS [59] or deal.II [60] use Trilinos and PETSc to handle the lower level and performance sensitive aspects of the simulation. As such, large scale parallel support is inherited from these frameworks. In addition, FEniCS offers a comprehensive set of tools for finite element modelling in the Python language.

Other projects, such as DUNE [61] or OpenFOAM [62] are more self-contained and do no rely on Trilinos or PETSc, implementing their own versions of the tools offered by these frameworks. They all offer a different interface for constructing a simulation and all have good support for MPI parallelism and more limited support for shared memory parallelism.

The LifeV C++ library, which represents the context in which the new developments presented in this work have taken place, falls into this category. A description of LifeV is given in section 2.6. The Feel++ library [63], also written in C++ should also be mentioned. It originated as a fork of LifeV, but has since grown into an independent project with its own goals and approaches.

Graphical user interface applications

Compared to the number of projects listed in the previous two categories, the number of open source GUI applications for finite element modelling is much smaller. Projects that focus on larger scale parallelism instead of workstation level performance will have limited use for a GUI. However, the Elmer project [64] provides a mature open-source application for FE modelling, with preprocessing, solving and postprocessing capabilities and with support for distributed and shared memory architectures.

2.5.1 GPU for numerical applications

Although in this thesis we do not focus on graphical processing units (GPU), we briefly describe here their use for numerical modelling software in high-performance computations.

Hardware overview

Using the GPU to perform general computations is not a new idea. However, before the development of the current programmable hardware and software application programming interfaces designed specifically for general purpose computations, scientists needed to implement their algorithms using graphical processing APIs like OpenGL and GLSL. This was a difficult process, since scientific applications use algorithms, for instance direct and iterative solvers, for which this hardware and software had not been intended.

The first release of GPUs built on the compute unified device architecture (CUDA) platform from NVIDIA and the Stream Computing platform from AMD represent the first generation of graphical hardware that directly support the development of scientific computing applications. CUDA GPUs are programmed using an extension to the C++ programming language (C for CUDA) and at this point represent the mainstream architecture for GPU computing.

GPUs are architecturally very different to modern CPUs, as they are more similar to vector computers from the earlier days of supercomputers, such as the Cray-2. They consist of a large number of processor cores, each core able to operate on multiple data elements in a vector fashion. The processor cores are not very sophisticated, lacking much of the logic hardware that allows modern CPUs to have very good scalar performance. The result is that GPUs achieve high performance by maintaining a large number of threads across all these processors cores, which hides the latency coming from the memory operations between the CPU memory and GPU memory. Additionally the hardware offers very little support for synchronization and communication between threads, with respect to a CPU. Algorithms that are most suitable for implementation on the GPU have a large computational intensity (the ratio between floating-point operations and memory operations) and can express parallelism with few or no points of global synchronization between parallel elements. An in depth description of CUDA hardware and software will not be given here, but is available in [13].

Finite element modelling on GPUs

One of the first numerical methods successfully ported to GPUs is the finite difference method (FDM). In [10] a single-precision implementation of FDM on the GPU is presented. It uses OpenGL for solving 2D electromagnetic scattering problems and is reported to be 7 times faster than a CPU implementation. The FDM, using a regular grid as spatial discretization, which ensures very regular memory access patterns, combined with an explicit time discretization scheme, is suited for implementation on GPUs.

The FE method, although more complicated from the implementation point of view, has also been adapted to use GPUs, even prior to the appearance of CUDA hardware and software. Turek et al. [65] attempted to use GPUs through the FEAST finite-element library which they develop. Initially, a single-precision iterative solver is implemented on the GPU to serve as a preconditioner for an outer iterative solver running in double precision on the CPU. A 2D Laplacian problem is solved on a regular cartesian grid. This approach, using OpenGL, is approximately 3.5 times faster than a CPU implementation. A later development by the same group is described in [66]. The FEAST library is used to solve a non-linear steady-state Navier-Stokes problem. The linearized subproblems of the non-linear solver are solved with a global BiCGStab preconditioned with a Schur complement matrix. Solving the advection-diffusion problem is done with a global multigrid solver that uses as smoother multi-grid solvers on the local domains running on the GPU. To ensure the regular access patterns suitable for the GPU, this strategy uses a 2D unstructured mesh composed of a small number of quadrilateral domains, while the domains themselves, on which the local multi-grid GPU solvers are operating, are discretized with regular generalized tensor product grids. The components that are ported to the GPU are up to an order of magnitude faster than the original CPU version. These components represent only a fraction of the total solver code, so the total simulation time is only decreased by a factor of two, as can be expected due to Amdahl's law.

There have been also other attempts to port multi-level preconditioners to GPUs, such as the one described in [67]. It is a multi GPU implementation of a preconditioned conjugate gradient (PCG) solver with an algebraic multi-grid preconditioner for 3D problems on unstructured grids. It is based on a GPU implementation of a sparse matrix vector product and uses an efficient interleaved compressed row storage format for the sparse matrices. This implementation is 13 times faster than one on a high end CPU. A PCG solver implemented on the GPU with a multi-level preconditioner for computational electromagnetics problems discretized with high order FE is presented in [68], [69]. The coarse level problem in the multi-level preconditioner is solved on the CPU using a direct solver. In this case, the GPU was 4 times faster than the CPU.

An earthquake modelling application has been ported to NVIDIA CUDA hardware, operating in single precision floating point arithmetic. Initially it ran on a single node equipped with NVIDIA GPUs [70], it was later ported to a large GPU cluster [71]. The application uses a high order spectral element method on a 3D unstructured hexahedral mesh with and explicit time discretization scheme. The entire finite element loop runs on the GPU and mesh colouring is used to avoid synchronization between thread blocks on the same GPU. The single node version of the code is up to 25 times faster than the CPU version of the code, while the MPI version is up to 20 times faster than the equivalent CPU implementation. The MPI version of the code also shows good weak scalability up to 192 GPUs.

A GPU implementation for fast simulations of the brain position shift during open cranium surgery is found in [72]. The software uses finite element discretization on an unstructured 3D mesh composed of linear tetrahedra and hexahedra and an explicit time advance scheme. The GPU components are written in CUDA, using single precision floating point arithmetic. This implementation is about 20 times faster than a CPU version, running in double precision.

A solver for Maxwell's equations running in single precision floating point arithmetic on NVIDIA CUDA GPUs is described in [17]. A nodal discontinuous Galerkin (DG)finite element discretization on an unstructured tetrahedral mesh is used along with an explicit Runge-Kutta time discretization. In the finite element loop the DG operator is split into several GPU kernels, according to memory access pattern, which permits a more efficient use of the hardware. A multi-GPU implementation using 8 GPUs is 18 times faster than a CPU version using 28 cores [19]. A more efficient multi-rate time discretization scheme is introduced in [73], in order to deal with the multi-scale nature in problem geometry. This version using 4 GPUs is up to 33 times faster compared to a CPU implementation running on a quad-code processor.

The key aspects of these efforts to implement the FE method on the GPUs are the prevalence of regular grids, which facilitate regular memory access patterns, essential for high performance achievement on the GPU. Additionally, FE discretizations such as the Discontinuous Galerkin approach are preferred, which allow splitting the assembly operators into independent and communications parts, in order to reduce the need for syncronization on the global GPU memory space. Finally, for strategies where the entire simulation is running on the GPU, the use of explicit time discretization schemes does away with the need to perform the solution of the global linear system of equations, either using a direct or an iterative solver. The best performance on a GPU is achieved for algorithms which can be implemented using single precision floating point operations. In the case of algorithms which require higher precision, there have been attempts to make use of linear solvers implemented in single precision arithmetic used in conjuction with iterative refinement performed in high precision [74], [75].

2.6 The LifeV parallel finite element modeling library

This final section of the chapter introduces the software framework in which the work presented in this thesis has been implemented, namely the LifeV parallel finite element library. An overview of the design and implementation of the library is given first, followed by a summary of the main limitations and performance bottlenecks of the library at the beginning of the author's doctoral work.

2.6.1 Design and implementation overview

LifeV (pronounced "life five") is a library for finite element modeling, developed as a joint effort between Ecole Polytechnique Federale de Lausanne (CMCS) in Switzerland, Politecnico di Milano (MOX) in Italy and Emory University in the U.S.A. The main application domain is modeling the cardiovascular system (fluid dynamics, structure dynamics, fluid-structure interaction, electromechanics of the heart).

The library has been designed for parallel operation. Although it can also be used on single processor computers, the main target platforms are parallel systems, like IBM Blue Gene and Cray supercomputers, that make it possible to perform large scale simulations, much beyond the memory capacity and computational power of a single computer.

LifeV is a C++ library built on top of Trilinos, which provides distributed sparse matrices and vectors, and parallel MPI numerical algorithms, like parallel direct and iterative solvers, preconditioners, load balancing and graph manipulation procedures etc. As a result, there are very few explicit calls to MPI functions, Trilinos acting as an abstraction layer above MPI. The extensive use of the Trilinos framework allows a consistent parallel operation in the main steps of a simulation: preprocessing, solving the linear system of equations, postprocessing.

LifeV is designed with modules dedicated to specific tasks. There is a core module, which implements algorithms for the finite element method independently of the problem domain, such as parallel matrix and vector interfaces, interfaces for linear algebra, matrix and vector assembly and solvers for linear and nonlinear problems. The other modules of the library depend on this core module and can be activated or deactivated at will. They represent extensions to the core features needed to perform specialized simulations, such as level set solvers, solvers and preconditioners for Navier-Stokes equations, electromechanics, fluid-structure interaction problems and multi-scale simulations.

The assembly of the linear system of equations is implemented using expression-templates, a template metaprogramming technique which provides a syntax closer to mathematics for defining the weak formulation of the chosen problem and ensures enhanced performance during this step of the simulation.

In addition to well known domain decomposition preconditioners, provided by the Trilinos libraries, LifeV contains for example a set of preconditioners designed for Navier-Stokes problems [76], [77], with an approach based on an approximate block factorization of the Navier-Stokes system matrix, and others for fluid-structure interaction problems [78].

Thanks to the domain-decomposition method the parallel implementation of a finite element solver is well suited for MPI. During simulations, LifeV performs the following steps:

1. **Initialization** - a given number of parallel MPI processes are started, each reading simulation parameters from disk.

2. **Mesh loading** - a polyhedral mesh describing the entire computational domain is read from disk by each MPI process.

3. **Mesh partitioning** - the global mesh is partitioned into a series of subdomains, one for each MPI process, using the ParMETIS parallel graph partitioning library. At the end of this process, each MPI process keeps in memory only the mesh partition that belongs to him, all further operations being performed on this local mesh.

4. **Assembly of the global linear system** - The linear systems associated with the Galerkin approximation of the chosen problem is stored in memory using Trilinos data structures distributed across all MPI processes. The assembly algorithm on each MPI process has been described in section 2.4.

5. **Solving the linear system of equations** - Once assembled, the linear system of equations is solved using a parallel implementation of an iterative solver, also provided by Trilinos, usually preconditioned conjugate gradient (PCG) or preconditioned GMRES.

6. **Postprocessing and output** - The solution of the problem is saved to disk using the high performance parallel file format, HDF5.

2.6.2 Initial limitations of LifeV

LifeV currently only uses MPI for parallel operation. While this was a viable approach at the time when LifeV was originally designed, the recent advances in computing hardware indicate that a pure MPI approach to parallelism may not be the path of least resistance towards achieving efficiency at scale[79]. A more flexible and hybrid parallelism could eliminate a series of performance limitations of LifeV:

- **Domain decomposition preconditioners** - the Algebraic Additive-Schwarz preconditioner is used, as a component, in all the preconditioner strategies that are available in LifeV: multi-level preconditioners, approximate block factorization preconditioners for Navier-Stokes. The available implementation is designed with a 1 to 1 correspondence between the number of MPI processes and DD subdomains. Uncoupling these two aspects in the implementation could increase the parallelism of the code, while maintaining the same numerical behaviour of the preconditioner.

- **Inefficient memory usage** - in the preprocessing phase of a LifeV simulation, the global mesh of the domain has to be partitioned between all the MPI processes. Due to the partitioning library used, it is necessary that each process holds at one point in time a complete image of the global mesh. This limits the maximum size of the simulations that can be attempted, a limitation which is becoming greater with the decreasing size of memory per processor that can be observed in newer supercomputers. Additionally, an MPI-only domain decomposition approach, due to the need to duplicate and exchange information in the halo regions of each subdomain requires more and more memory when increasing the number of MPI processes.

- **Unexplored sources of parallelism** - using multiple threads, in addition to MPI parallelism, in certain stages of the simulation, such as assembly and the solver and preconditioner could allow a better exploitation of modern hybrid supercomputing architectures.

3 A domain decomposition preconditioner based on two-level MPI parallelism

This chapter focuses on a novel implementation of a domain decomposition preconditioner, based on two levels of MPI parallelism, in order to better match the topology of the underlying hardware and cope with the serial bottleneck bound to the coarse solver. We begin with an overview of the algebraic additive Schwarz (AAS) preconditioner, and the two-level Schwarz variant, which represent the setting in which the new approach is implemented. The new strategy with two levels of parallelism is then described, from the algorithmic and implementation viewpoint. The final part of the chapter contains a discussion of different ways to treat the subdomain problems that are associated with the AAS preconditioner, including a new one based on a second level of parallelism.

3.1 The algebraic additive Schwarz preconditioner

The Schwarz method is one of the earliest domain decomposition methods [80], which has seen a resurgence with the advent of parallel computing and is now in wide use in finite element software packages. It provides an ideal framework for parallel execution, thanks to its reinterpretation as an efficient preconditioner.

We describe the algebraic additive Schwarz (AAS) preconditioner, by reconsidering the Poisson problem from the previous chapter (2.1). The domain Ω is partitioned in several overlapping subdomains (see Figure 3.1). Suppose we want to solve the linear system of equations deriving from the finite element discretization of this problem,

$$A\mathbf{u} = \mathbf{f}. \tag{3.1}$$

Because of the problem size or for efficiency reasons we adopt a preconditioned iterative method like GMRES or Conjugate Gradient. The idea of the additive Schwarz preconditioner is breaking the global problem down into a series of local problems of Dirichlet type on overlapping subdomains and defining the global preconditioner P_{AS} of A as a sum of transformations

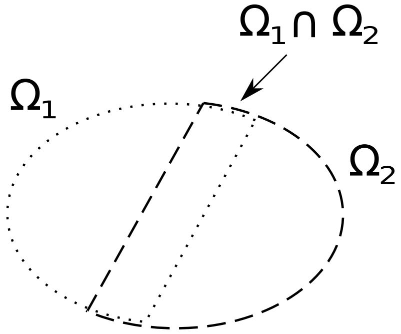


Figure 3.1: The case when domain Ω is divided into two subdomains Ω_1 and Ω_2, with overlap $\Omega_1 \cap \Omega_2$.

of local inverse matrices, as follows:

$$P_{AS}^{-1} = \sum_{i=1}^{N} R_i^T \hat{A}_i^{-1} R_i, \tag{3.2}$$

where N is the number of subdomains, R_i is the restriction operator from the global problem to the subdomain problem, R_i^T is the prolongation operator from the subdomain problem to the global one and $\hat{A}_i^{-1}$ is an exact or inexact inverse of the local stiffness matrix $A_i = R_i A R_i^T$. When solving (3.1) with an iterative method, it is necessary to repeatedly solve the problem

$$P_{AS}\mathbf{z} = \mathbf{r}, \tag{3.3}$$

which is composed of N independent problems

$$\hat{A}_i \mathbf{z}_i = R_i \mathbf{r}. \tag{3.4}$$

In a parallel setting, where the rows of the global matrix A are already distributed among the available processes, the restriction operation involves selecting the locally stored rows from the global matrix and any additional rows which correspond to the degrees of freedom in the imposed overlap region. This local matrix $\hat{A}_i$ has a much smaller size than the global problem, it is therefore possible to compute an exact or inexact LU or Cholesky factorization if it. The prolongation operator represents applying $\hat{A}_i^{-1}$ to the set of coefficients in the residual vector, which correspond to the degrees of freedom contained in a subdomain, in the course of a Krylov iteration, like conjugate gradient (CG) or the generalized minimum residual (GMRES) methods.

3.1.1 Optimality and scalability

One important property of preconditioners in general is optimality. A preconditioner is considered optimal if its condition number is independent of the finite element gridsize which is used. This means that using an optimal preconditioner there is an upper bound on the number of iterations that an iterative solver will require to converge (up to a prescribed tolerance), independently of the mesh refinement used.

In the context of parallel computing and parallel preconditioners, we would like to define an additional metric, namely preconditioner scalability. It has a slightly different meaning than strong and weak scalability, described in the previous chapter. We consider that a preconditioner is scalable if its effectiveness is independent of the number of processors used, which, in the case of the pre-existing implementation of AAS, is the number of subdomains used in the domain decomposition scheme. As in the case of optimality, the upper bound on the number of iterations that an iterative solver, using a scalable preconditioner, requires to converge should be independent of the number of processors used.

There exists in the literature a detailed analysis of the additive Schwarz method [81] [82] [27] [51]. In this section we would like to recall some results which describe the optimality and scalability of the AAS preconditioner in the case of Poisson problems. The convergence rate of the preconditioned Conjugate Gradient solver when both A and the preconditioner are symmetric positive definite is (see, [83]):

$$||\mathbf{U}^k - \mathbf{U}||_A \leq 2\left(\frac{\sqrt{\kappa(P^{-1}A)}-1}{\sqrt{\kappa(P^{-1}A)}+1}\right)^k ||\mathbf{U}^0 - \mathbf{U}||_A, \tag{3.5}$$

where $||\mathbf{v}||_A = \sqrt{(\mathbf{v},\mathbf{v})_A}$ is the norm associated with the scalar product $(\mathbf{v},\mathbf{w})_A = (A\mathbf{w},\mathbf{v})$. This inequality relates $\kappa(P^{-1}A)$ with the number of iterations necessary to achieve a prescribed tolerance.

The following estimate is given in [29] for the condition number of the AAS preconditioner:

$$\kappa(P_{AS}^{-1}A) \leq C\frac{1}{\delta^2 H^2}, \tag{3.6}$$

where H is the maximum diameter of the subdomains, $\delta \in (0,1)$ is a measure of the overlap between subdomains and the constant C does not depend on the mesh refinement h or the subdomain size H (which gives the number of subdomains), but depends on the coefficients of the operator of the problem. If we consider a three dimensional case, the number of subdomains $N \approx C\frac{1}{H^3}$ and the estimate (3.6) can be written in terms of N:

$$\kappa(P_{AS}^{-1}A) \leq C\frac{1}{\delta^2}N^{2/3}. \tag{3.7}$$

The AAS preconditioner has the optimality property, but is not scalable.

One way to improve the scalability of the AAS preconditioner is to add to the subdomain problems an additional coarse problem, using a coarse mesh where each element represents one of the subdomains. In the new, two-level form, the AAS preconditioner can be written:

$$P_{CAS}^{-1} = R_H^T \hat{A}_H^{-1} R_H + \sum_{i=1}^{N} R_i^T \hat{A}_i^{-1} R_i, \tag{3.8}$$

where R_H and P_H are the restriction and prolongation operators associated with the coarse problem and $\hat{A}_H^{-1}$ is the exact or inexact solve of the coarse problem.

For the two-level formulation of AAS, [84] gives an improved estimate for the condition number, independent of the number of subdomains:

$$\kappa(P_{CAS}^{-1} A) \leq C\frac{1}{\delta}. \tag{3.9}$$

This estimate shows that in the presence of a coarse problem the preconditioner is scalable.

We note that in both cases the overlap δ has to be chosen by the user. In many cases, $\delta = O(h)$, which of course leads to a suboptimal preconditioner. However, having δ independent of h means that the overlap includes more and more mesh layers as h decreases. This implies more communication. Later on, we choose the minimal overlap $\delta = h$, i.e. minimal communication at the price of sub-optimality.

A qualitative interpretation of the two estimates presented here is that in the one-level formulation, the AAS preconditioner degrades with the increase of subdomains, as the exchange of information is only done through the overlap, and only between neighbouring subdomains. The presence of a coarse level removes this restriction and allows an exchange of information, even if only coarsely, between any two given subdomains.

From the point of view of the implementation, when using regular grids, the fine and coarse grids are easy to obtain. First the coarse grid is generated and then the fine grid is obtained by refining the coarse grid until the desired grid size is attained. In the case of unstructured meshes, on a domain with irregular geometry, it is not possible to obtain the fine mesh from the coarse one, as the fine mesh is needed to properly approximate the boundary of the domain. Second, the coarsening procedure for the fine mesh is not trivial. In practice, the easiest way to generate the coarse level problem for a 2-level Schwarz scheme is by using an algebraic multi-grid (AMG) preconditioner [85], such as the ML package in Trilinos. The AMG preconditioner handles building the coarse problem and the AAS preconditioner is set as a smoother on the fine level.

3.1.2 Restricted Schwarz algorithm

A variation of the AAS preconditioner is the restricted Schwarz algorithm [51]. The prolongation operators R_i^T are simplified by discarding the overlap information. The restricted variant of (the one-level) AAS can be written as:

$$P_{AS}^{-1} = \sum_{i=1}^{N} \tilde{R}_i^T \hat{A}_i^{-1} R_i, \tag{3.10}$$

where $\tilde{R}_i^T$ is a trivial prolongation operator which discards non local (i.e. not owned) degrees of freedom. When computing A_i^{-1} the data on the halo is gathered from the neighbour processes. The same happens when applying the restriction R_i. Both steps require communication, however R_i^T does not, since the non local data is discarded at this point.

Although a complete analysis of this approach is to our knowledge still missing, the restricted variant has been shown to lead to improved convergence used within GMRES with respect to the standard form of AAS [86], while in the case of some symmetrical problems, GMRES preconditioned with the restricted version of AAS performed better than CG with the standard AAS, from the point of view of the number of iterations and CPU time [51]. It should be noted that the restricted variant of AAS, due to its improved performance, is currently the default implementation of AAS in both Trilinos and PETSc numerical frameworks. For the remainder of this work, the restricted variant should be assumed whenever AAS is mentioned.

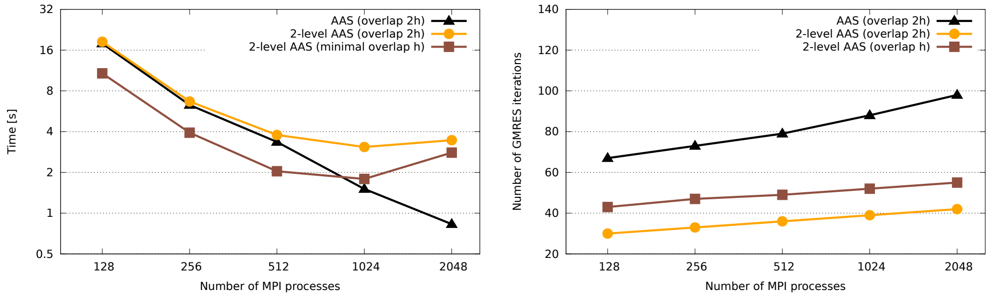
3.1.3 Numerical test

What follows is a simple numerical experiment, meant to visualize the differences in preconditioner scalability between single level AAS and 2-level AAS. We setup, using LifeV, a 3D Laplacian boundary value problem, on a regular cubic grid. P2 finite elements are used which results in a linear system with approximately 5 million degrees of freedom (DOF). To study strong scalability, the number of MPI processes is varied between 128 and 2048. The linear system is solved using GMRES with a tolerance of 10^{-10}. Given that the condition numbers of AAS and 2-level AAS are influenced by the level of overlap (see Equations (3.6) and (3.9)), three preconditioners are considered:

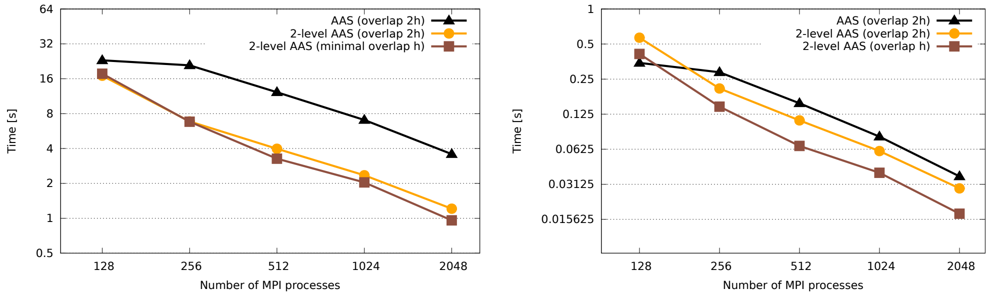
1. AAS - the single level variant implemented in the IFPACK package of Trilinos, with overlap $\delta = 2h$. Exact LU factorization is used to solve the subdomain problems, provided by the PARDISO [87] linear solver package.

2. 2-level AAS, with overlap - identical to the first case, but for the addition of a coarse problem (through the ML package). Unfortunately, because of the number of MPI processes used, it isn't possible to solve the coarse problem exactly. Instead we use 5 iterations of Gauss-Seidel. It should be noted that the implementation of the coarse solve is serial, independently of the number of MPI processes that are used. The coarse problem is constructed and solved on the first MPI process.

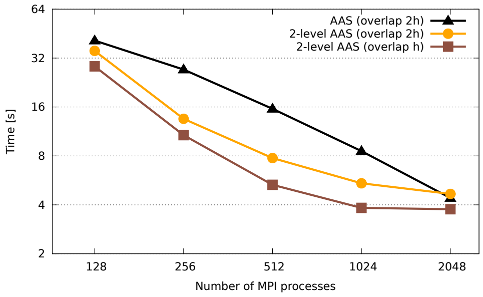
3. 2-level AAS, with minimal overlap - the same configuration as in the previous case, with the exception of the overlap between subdomains, which is set at $\delta = h$.



(a) Time to compute the preconditioner (b) Number of GMRES iterations



(c) Time to GMRES convergence (d) Time per GMRES iteration



(e) Total time to solution

Figure 3.2: Comparison of AAS and 2-level AAS for a 3D Laplacian problem on a regular cubic grid with P2 finite elements, 5 million degrees of freedom. Strong scalability is examined for different levels of overlap.

In the implementation of AAS from the IFPACK package the minimal overlap $\delta = O(h)$ is represented as zero overlap, while overlap of $\delta = O(2h)$ is represented as overlap of level 1. The reason for this is that the boundary degrees of freedom in the overlapping subdomains represent the homogeneous Dirichlet boundary conditions of the subdomain problems and are eliminated.

Figure 3.2 displays the results of this numerical test. Measured quantities are the time to compute the preconditioner, the number of GMRES iterations to convergence and the time to GMRES convergence. Using the three measurements, we also compute the time to perform one GMRES iteration and the total time, which represents the sum of the time to compute the preconditioner and the time to GMRES convergence. The total time reflects the overall effectiveness of each approach.

Figure 3.2b shows the number of iterations to GMRES convergence in the three cases. The presense of the coarse problem is effective at limiting the iteration creep that is very pronounced in the case of single level AAS. The number of iterations is still increasing with the number of subdomains even in the case of 2-level AAS, due to the fact that the coarse problem is not solved exactly. While the impact of the overlap level in the 2-level preconditioners is visible, the iteration count increases with the number of processes at the same rate, regardless of the level of overlap.

The cost to compute the preconditioner is higher in the two level case (Figure 3.2a), as is expected due to the added coarse problem, although the reduced number of iterations that are needed leads to an overall lower time to GMRES convergence for 2-level preconditioners (Figure 3.2c). The serial implementation of the coarse level greatly impacts strong scalability beyond a certain number of subdomains and MPI processes.

Finally, considering the time to compute the preconditioner together with the time to GMRES convergence (Figure 3.2e), we observe that the 2-level case with minimal overlap is overall the fastest approach, due to the slightly smaller subdomain problems that are used and the reduced communication costs involved. In the parts that follow, this is the configuration that will be used for numerical tests using 2-level AAS.

3.2 Two-level parallelism in AAS

In Trilinos, the AAS preconditioner is implemented in the IFPACK package, which provides the algebraic domain decomposition framework, as well as support for a collection of different exact or inexact LU solvers for use with the local subdomain problems. The main limitation of the implementation is the strict 1:1 relationship between the number of subdomains for the domain decomposition scheme and the number of MPI processes in use. As shown in the initial benchmark from the previous section, due to this dependence the parallel AAS preconditioner is not scalable with the number of processes. We also observe a loss of strong scalability in the preconditioner computation phase, since the coarse problem is solved serially, on the first MPI process. This is a lesser limitation for stationary simulations, where the preconditioner needs to be computed only once, but in the case of time-dependent simulations this becomes a greater problem. In the non-linear case, although many times it is possible to reuse the preconditioner, if the preconditioner needs to be updated multiple times per time step, this loss of scalability becomes even more costly to the overall performance.

In this section we describe the implementation, in the AAS framework, of the support for parallel subdomain problems. This implementation is similar to an earlier attempt, described in [30] and it allows computing the AAS preconditioner using a number of processors that is a multiple of the number of subdomains.

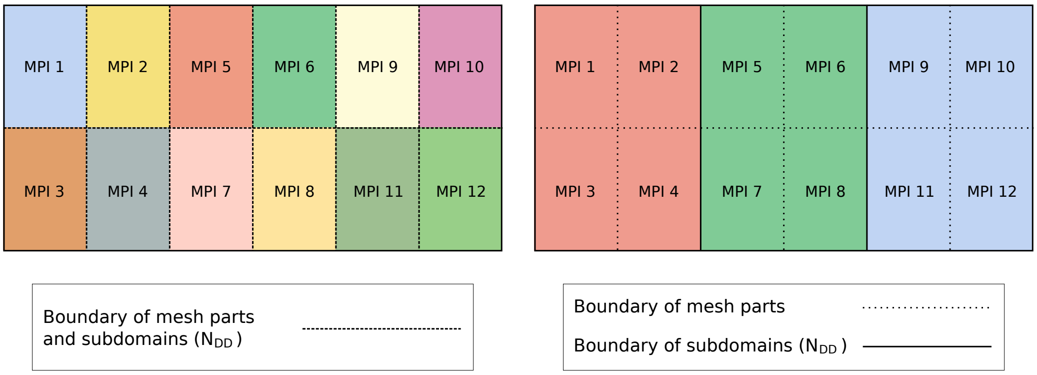
3.2.1 Implementation details

In the first section of this chapter, it was already mentioned that the AAS preconditioner construction maps well on top the MPI parallel decomposition used for the linear system matrix, since the rows of the matrix that are stored on each processor represent the non-overlapping parts of each subdomain problem. Introducing two levels of MPI parallelism to the AAS preconditioner does not affect the mesh partitioning and the assembly of the linear system of equations. Both of these steps are performed using the total number of MPI processes available, which we denote by N_P. The mesh is partitioned into N_P parts and the linear system of equations is constructed with the rows of the system matrix distributed across all the processes. At this point, we can either impose a number of subdomains N_{DD} for AAS or impose how many MPI processes should be used per subdomain N_S. The following relationship applies:

$$N_P = N_{DD} \times N_S. \tag{3.11}$$

The processes which belong to each subdomain are grouped together and construct the subdomain matrix using their locally stored rows. The coefficients associated with the coupling between the subdomain's degrees of freedom and those of the other subdomains are discarded. Figure 3.3 illustrates the process described here. This approach results in an AAS configuration with minimal overlap which, as shown in the previous section, is preferred.

The parallel subdomain problems, once built, can be solved with an MPI parallel solver which is able to use Trilinos parallel sparse matrices. Trilinos includes an interface to MUMPS, a mature and efficient parallel direct solver. By default, it was intended to be used for solving the global problem, but we have made modifications in Trilinos that restrict its operation to the MPI subcommunicators associated with each parallel subdomain. As an alternative to using parallel direct solvers for subdomain problems, Trilinos also provides the ShyLU package



(a) Serial subdomain problems: $N_P = 12$; $N_{DD} = 12$; $N_S = 1$

(b) Parallel subdomain problems: $N_P = 12$; $N_{DD} = 3$; $N_S = 4$

Figure 3.3: Domain decomposition for AS with one or two levels of parallelism. Each AS subdomain has a different color.

which can solve these problems inexactly. A discussion of ShyLU is given in Section 3.3, while Section 3.4 contains a set of benchmarks where MUMPS and ShyLU are compared.

Some serial direct solvers which are available in Trilinos, such as KLU [88] and UMFPACK, are able to solve the parallel subdomain problems, serially, by collecting the problems on one process. The same applies to incomplete Cholesky and LU factorizations implemented in Trilinos. The use of these serial exact or inexact solvers is only useful during development or debugging, due to the serial bottleneck that is introduced.

3.2.2 Numerical assessment

To verify the correctness of the implementation of the parallel subdomain problems for AAS, a simple numerical test is set up. A 3D Laplacian problem is solved with GMRES and preconditioned with AAS. A varying number of MPI processes is used, ranging from 8 to 512. First, the subdomain problems are serial ($N_S = 1$, variable N_P); then the subdomains problems are solved in parallel, keeping N_{DD} constant, while varying the total number of processes N_P and the number of processes per subdomain N_S (constant N_{DD}, $N_S = N_P / N_{DD}$).

The results of this test are shown in Figure 3.4 and it can be seen that the implementation is working as expected. This test involves no performance measurements, as the only concern is the correctness of the implementation. We expect a constant number of GMRES iterations, for a constant number of AAS subdomains. For validation, in both the serial ($N_S = 1$) and the parallel ($N_S \geq 1$) case, a serial LU solver is used, which gathers the subdomain problem on the first MPI processes of each subdomain.

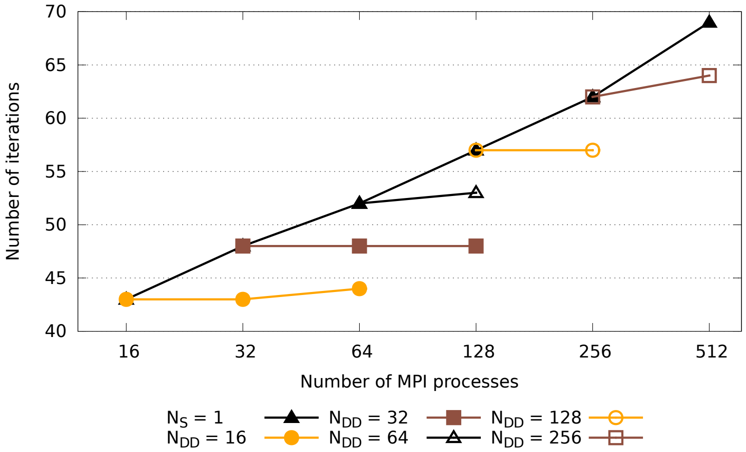


Figure 3.4: Verification of the implementation of parallel subdomain problems for AAS. Keeping the number of subdomains constant, while varying the total number of MPI processes, results in a constant number of GMRES iterations.

3.3 ShyLU - a hybrid subdomain preconditioner

This section describes an alternative approach to treating subdomain problems in AAS with exact LU factorizations. ShyLU is a hybrid preconditioner for multicore platforms, available as a package in Trilinos [31]. The term hybrid refers to the mixed direct and iterative approach used, as well as to the fact that while the main parallelisation scheme of ShyLU is MPI, it can also make use of software subcomponents that are multi-threaded. There exist similar efforts for developing such hybrid solvers or preconditioners [89], [90], [91]. While ShyLU was originally developed as a preconditioner for circuit simulation problems, the author of this thesis collaborated with the original developers of ShyLU with the goal of modifying and adapting the preconditioner to the 2-level MPI framework described in the previous section, i.e. to use it as an inexact solver for AAS. ShyLU has been developed with circuit applications in mind. We are interested in extending it as a parallel inexact solver for the subdomain problems in AAS in the finite element context.

The end goal is determining the effectiveness of ShyLU as a subdomain preconditioner in AAS for PDE problems and introduce new components adapted to matrices that are less sparse than the ones encountered in circuit simulations.

3.3.1 Algorithmic description

The ShyLU preconditioner is based on a Schur complement at the level of the partitioned subdomain. Starting from a non-overlapping partition of the subdomain, the degrees of freedom are grouped into internal ones and interface ones. Since the partitions are created with a minimal interface, the corresponding matrix looks like in Figure 3.5 (possibly after reordering). We suppose to be able to compute an approximated Schur complement relative to the interface degrees of freedom. Indeed, ShyLU performs a partitioning and permutation of the matrix, in order to obtain a matrix with a block diagonal upper left block. The D and G blocks are square, while D is non-singular.

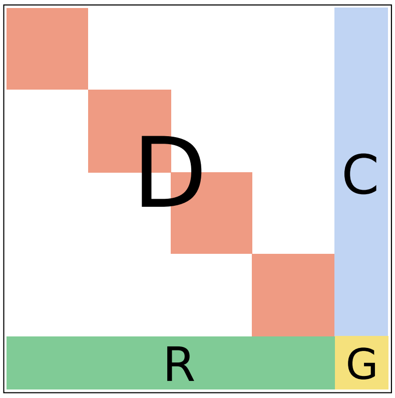


Figure 3.5: The matrix ShyLU produces for a decomposition into 4 non-overlapping parts

In the case of LifeV, where ShyLU is applied to the parallel subdomain problems which are built with the approach described earlier, there is no need for a second partitioning of the matrix, as the subdomain problem matrices already have the needed structure.

To describe the steps of the Schur complement algorithm, let us consider the linear system:

$$A_i \mathbf{u}_i = \mathbf{f}_i, \tag{3.12}$$

which represents our problem for subdomain i of the AAS preconditioner. Consequently, A_i, $\mathbf{u}_i$ and $\mathbf{f}_i$ are the restrictions to the subdomain i of the system matrix, solution and right-hand side. This problem can be written in terms of blocks, after permutation:

$$A_i = \begin{bmatrix} D & C \\ R & G \end{bmatrix}, \quad \mathbf{u}_i = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \quad \mathbf{f}_i = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}. \tag{3.13}$$

The blocks R, C and G are all stored as parallel matrices, with rows distributed across all the processes associated with the problem. Each process will only store its own subblock of D, as a serial matrix. ShyLU solves the problem by performing Algorithm 5.

Algorithm 5 does not solve (3.13) but provides an approximation. We intend to use ShyLU as part of a preconditioner. The exact Schur complement is dense, while an approximation can reduce the memory requirements of the algorithm. Steps 1, 2 and 6 of the algorithm are embarassingly parallel, due to the block diagonal structure of D, while Steps 3, 4 and 5 involve some communication between processes.

While it is beyond the scope of this work to cover the analysis of Schur complement preconditioning methods, we refer the reader to the relevant literature [51], [27]. In the following sections we will focus on implementation and performance issues.

```
Factorize: $D$                                   (NO COMM)      1
Solve: $D\mathbf{z} = \mathbf{f}_1$                               (NO COMM)      2
/* $\bar{S}$ is an approximation of Schur complement                    */
Compute: $\bar{S} \approx G - RD^{-1}C$                                          3
Solve: $\bar{S}\mathbf{u}_2 = \mathbf{f}_2 - R\mathbf{z}$                                            4
Compute: $\mathbf{t} = \mathbf{f}_1 - C\mathbf{u}_2$                                          5
Solve: $D\mathbf{u}_1 = \mathbf{t}$                              (NO COMM)      6
                                                                7
```

Algorithm 5: The solution algorithm used by ShyLU. All the steps are performed in parallel. Lines marked with **NO COMM** denote operations that are embarassingly parallel, i.e. no communications is needed between processes.

3.3.2 Implementation overview

ShyLU is designed to be configurable and modular and can use other components from Trilinos in various stages of its algorithm. This subsection seeks to describe the main configuration options and parameters that ShyLU currently exposes to the user.

Solvers for diagonal blocks

Using efficient solvers for the diagonal blocks is of great importance, since they represent the largest part of the subdomain matrix. The matrix partitioning, either done during the ShyLU algorithm or resulting from the geometric mesh partitioning, aims at minimizing the interfaces between partitions. In consequence, the diagonal blocks, which contain the interior DOFs of each part, are maximized with respect to the entire matrix. Another consequence of the block diagonal structure is that the linear systems

$$D_i \mathbf{z}_i = \mathbf{f}_i, \quad i = 1,..,N_S \tag{3.14}$$

and

$$D_i \mathbf{u}_{1i} = (\mathbf{b}_i - C\mathbf{u}_2)_i, \quad i = 1,...,N_S, \tag{3.15}$$

associated with the blocks can be solved independently by the processes involved.

A sparse direct solver such as UMFPACK is used here. This solver is usually serial, although an MPI parallel solver such as MUMPS can also be used, in serial mode. As stated earlier, some modifications were needed in Trilinos to allow this mode of operation. MUMPS was initially operating only in parallel, using the MPI processes available for the entire simulation, without the possibility to restrict it to a subset of processes. If there are extra cores available which cannot be used for MPI, a multi-threaded direct solver such as PARDISO can be used. As with MPI solvers, a multi-threaded solver can of course also be used in serial.

For the purpose of the benchmarks performed in this chapter, no multi-threaded strategy is used in ShyLU or outside, focusing instead on the 2-level MPI parallelism. ShyLU also allows that inexact factorizations be used for the diagonal blocks, although this configuration is not examined in this work.

Approximation method for the Schur complement

The Schur complement matrix:

$$S = G - RD^{-1}C \tag{3.16}$$

is dense. ShyLU uses a sparse approximation, denoted $\bar{S}$, to lower memory usage. ShyLU provides two strategies to compute such an approximation. The first one involves computing

$\bar{S}$ as per (3.16), a small number of columns at a time, and dropping values from these columns based on an imposed threshold. The second strategy is based on the interface probing technique [92], i.e. imposing a sparsity pattern of the approximate Schur complement. The chosen sparsity pattern is that of a banded matrix, with a bandwidth that is imposed by the user. The approximation $\bar{S}$ is improved by adding the structure of block G to the banded structure [31]. Neither strategy involves explicitly computing the exact Schur complement matrix, as only the sparse approximation $\bar{S}$ is computed explicitly and stored as a parallel sparse matrix.

As we see later on, choosing between the two methods and tuning the parameters of the dropping and probing directly impacts the time to compute the preconditioner and time to GMRES convergence.

Solvers for the approximate Schur complement system

Once the approximate Schur complement $\bar{S}$ is available, the solution of the approximate Schur complement system:

$$\bar{S}\mathbf{u}_2 = \mathbf{f}_2 - R\mathbf{z} \tag{3.17}$$

can be performed in different ways. It can be solved directly and an MPI parallel direct solver such as MUMPS needs to be used, since the sparse approximation of the Schur complement $\bar{S}$ is stored in parallel regardless of approximation strategy. The size of this system is minimized by the partitioning. Using a serial direct solver is possible, but not advised, as it can become a considerable bottleneck, due to the added communication cost introduced by gathering all the rows of the system on a single process.

The second way to solve the system is with a parallel iterative solver, typically GMRES. Two options are possible for stopping the iterative solver. A stopping tolerance can be imposed, which could lead to a prohibitively large CPU time if the tolerance is too small. Alternatively, the solver can be stopped after a fixed number of iterations. Setting this number too low leads to an increase in the number of outer iterations. Figure 3.6 shows the convergence history of the inner solver for different problem sizes, using 8 MPI processes. Due to the approximation methods of the Schur complement, we can not rely on any convergence analysis results available.

The benchmarks described in the last section of this chapter show that even a relatively low number of inner iterations (less than 10) leads to only a small increase in the number of outer iterations.

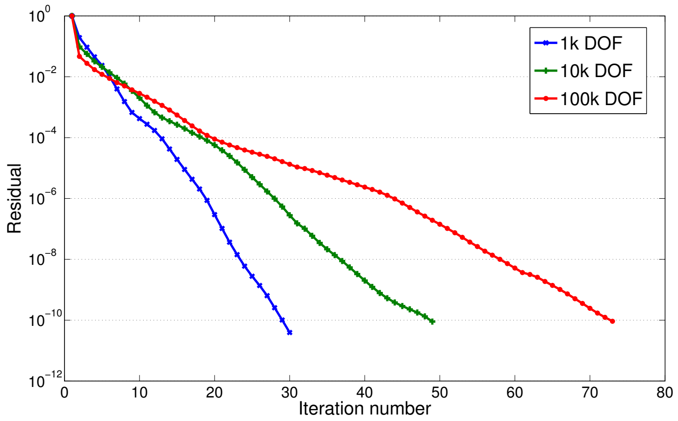


Figure 3.6: Convergence history of inner iterations in ShyLU for three problem sizes, using 8 MPI processes

3.4 Benchmarks

This final section of the chapter contains a set of numerical tests which allow to evaluate the effectiveness of the 2-level MPI parallelism for AAS and the ShyLU subdomain preconditioner. ShyLU is first studied without AAS, as a global preconditioner for GMRES and it is compared with the MUMPS MPI parallel direct solver, both of which can be used for the subdomain problems. The last set of benchmarks compares ShyLU and MUMPS, when used in a 2-level AAS preconditioner for different problems.

3.4.1 Node-level measurements

An initial set of tests is presented with the goal of determining a good configuration for ShyLU and also comparing it in terms of performance to the exact parallel LU factorization provided by MUMPS. For this set of tests, AAS is not used.

We discretize using P1 finite elements, on regular grids, a 3D Laplacian problem

$$\begin{cases} -\Delta u_1 = f_1 \quad \text{in} \quad \Omega \subset \mathbb{R}^3 \\ \quad u_1 = \psi_1 \quad \text{on} \quad \partial\Omega \end{cases} \tag{3.18}$$

and a 3D advection-diffusion-reaction (ADR) problem

$$\begin{cases} -\Delta u_2 + \beta \nabla u_2 + u_2 = f_2 \quad \text{in} \quad \Omega \subset \mathbb{R}^3 \\ \qquad\qquad\qquad u_2 = \psi_2 \quad \text{on} \quad \partial\Omega, \end{cases} \tag{3.19}$$

where $\beta : \mathbb{R}^3 \to \mathbb{R}^3$ is a constant advection field and the functions ψ_1 and ψ_2 used when imposing the essential boundary conditions represent the exact solutions to the two problems. We examine the results for different magnitudes of the advection field in the ADR problem. In terms of scalability and performance, the benchmarks produced the same results in all cases and we only present one set of results, corresponding to $\beta = (1,0,0)$. Three problem sizes (50000, 100000 and 200000 DOFs) were considered for each problem type.

The tests are run on one node of the "Aries" cluster at EPFL, which contains 4 AMD Opteron "Magny-Cours" processors, each with 12 CPU cores. The total amount of RAM available on the node is 24 GB. We use up to 48 MPI processes, on a single node of a computer cluster, starting from 1 in the case of MUMPS and starting from 2 for ShyLU, since running ShyLU on 1 processor does not invoke the Schur complement framework and represents just a serial LU factorization.

In the case of ShyLU, GMRES is used as an iterative solver. The two strategies for approximating the Schur complement are examined for ShyLU, and in the case of probing, two diagonal bandwidths are compared. MUMPS is used inside ShyLU both for the diagonal blocks and for the approximate Schur complement system.

First the factorization phase is measured, which in the case of MUMPS includes the symbolic and numeric factorizations. In the case of ShyLU it involves factorizing the diagonal blocks and computing the approximation of the Schur complement.

We also record the number of GMRES iterations required to converge, in the case of ShyLU. In the case of MUMPS, due to an implementation detail, we are forced to used GMRES as the global solver. MUMPS is applied as a preconditioner for GMRES and the iteration count in this case is meaningless, as it is always equal to one.

The time per GMRES iteration is also recorded and finally a total time to solution, which includes the time to compute the preconditioner and the time to GMRES convergence, is recorded to give a general view of the effectiveness of both approaches.

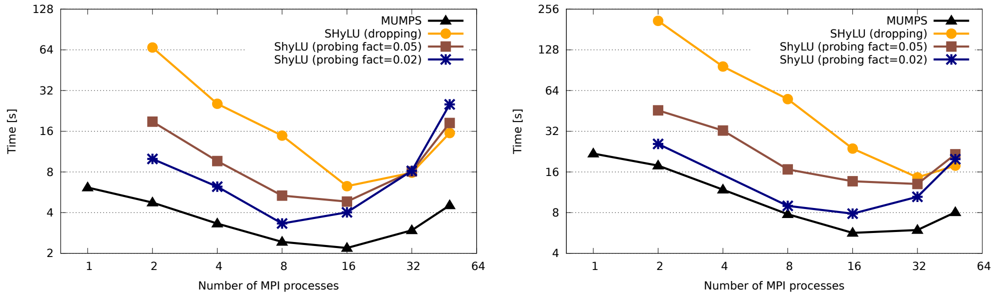
Laplacian problems

In terms of computing the preconditioner (Figure 3.7), MUMPS comes ahead of ShyLU consistently, although it is important to note that it demonstrates poorer scalability. Additionally we see that the first two problems are too small to maintain scalability when using 48 processes. The threshold dropping strategy is considerably slower than probing and lowering the diagonal bandwidth in the case of probing also has a visible impact, making ShyLU approach MUMPS in terms of CPU time.

The slower threshold dropping method has an advantage over the probing method in terms of the number of GMRES iterations needed to converge (Figure 3.8), as the dropping produces a better approximation of the exact Schur complement.

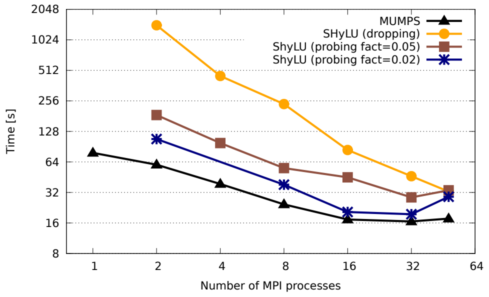
From the point of view of time to perform one GMRES iteration (Figure 3.9), which involves a forward and a backward triangular solve for MUMPS and two solutions of the diagonal blocks and the solution of the approximate Schur complement system for ShyLU, MUMPS is slower and less scalable than ShyLU. There is also no considerable difference between dropping and probing. Given that in the case of multi AAS subdomains both for MUMPS and ShyLU GMRES will require multiple iterations to converge, this could prove a signification benefit for ShyLU.

In terms of total time to solution (Figure 3.10) we see that ShyLU comes close to MUMPS, but only when using probing to approximate the Schur complement and with an appropriately small diagonal bandwidth factor.



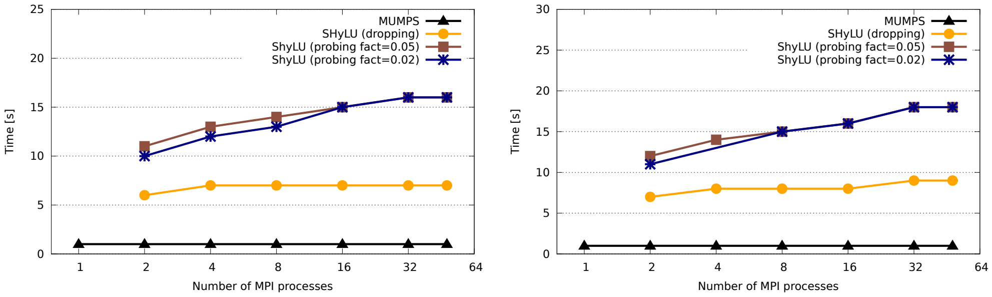
(a) 50k DOF

(b) 100k DOF



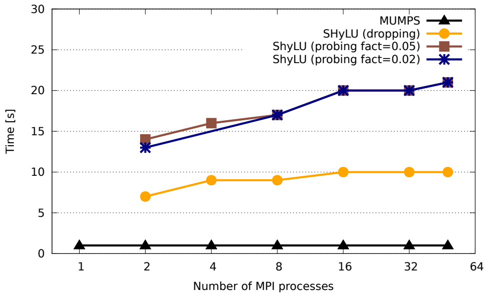
(c) 200k DOF

Figure 3.7: Time to compute the preconditioner - 3D Laplacian problems



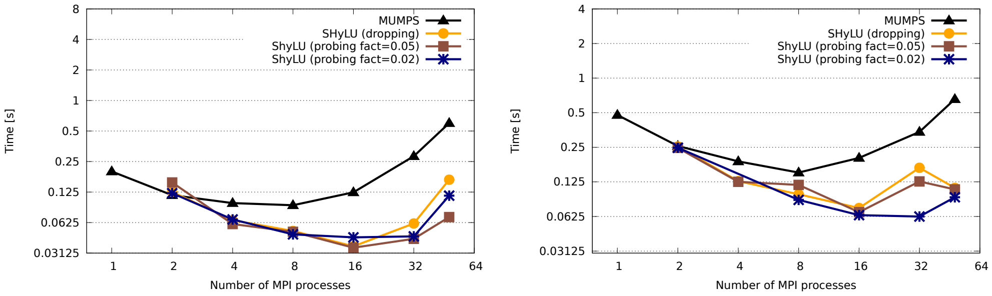
(a) 50k DOF

(b) 100k DOF



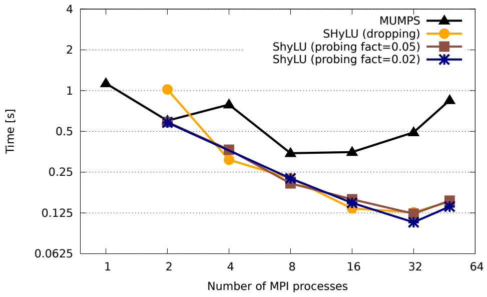
(c) 200k DOF

Figure 3.8: Number of GMRES iterations - 3D Laplacian problems



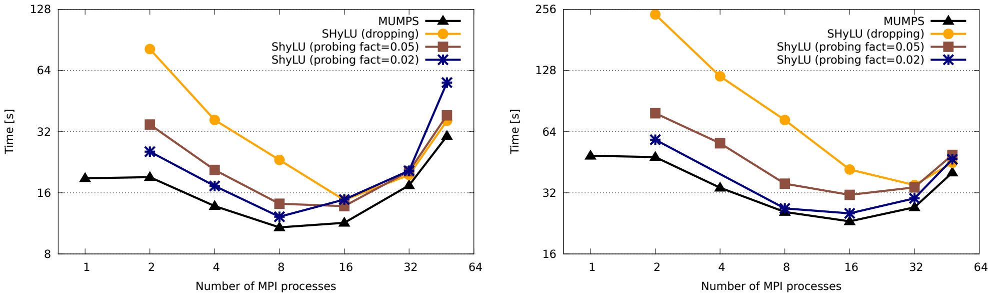
(a) 50k DOF

(b) 100k DOF



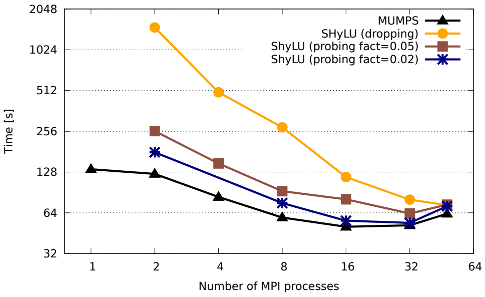
(c) 200k DOF

Figure 3.9: Time per GMRES iteration - 3D Laplacian problems



(a) 50k DOF

(b) 100k DOF



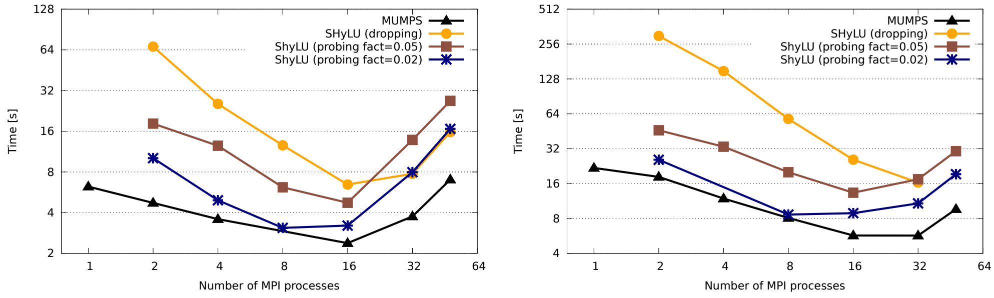
(c) 200k DOF

Figure 3.10: Total time to solution - 3D Laplacian problems

Advection-diffusion-reaction problems

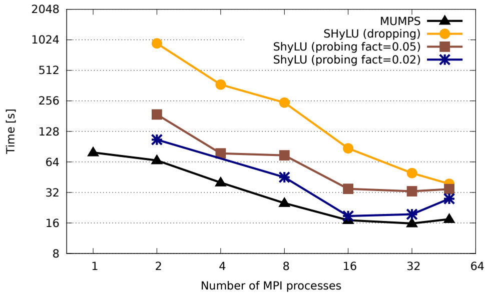
The results for the ADR problems are similar to the results for the Laplacian problems. The same conclusions apply as ShyLU and MUMPS have the same relative and absolute performance as in the previous set of tests.

For ShyLU, this means that the algorithm is a viable approach for both symmetric and asymmetric problems and allows us to consider its use in more complicated problems.



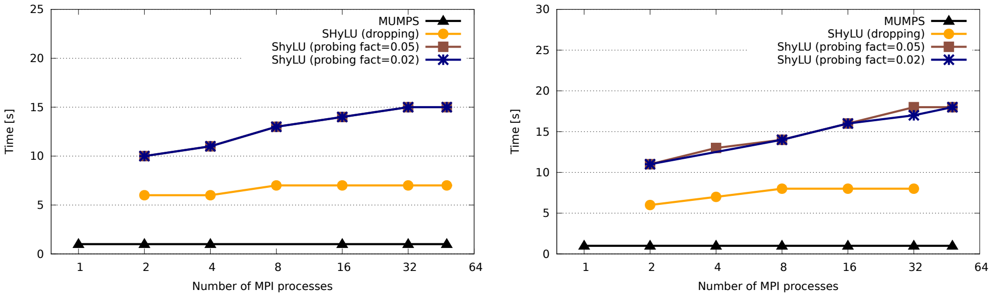
(a) 50k DOF

(b) 100k DOF



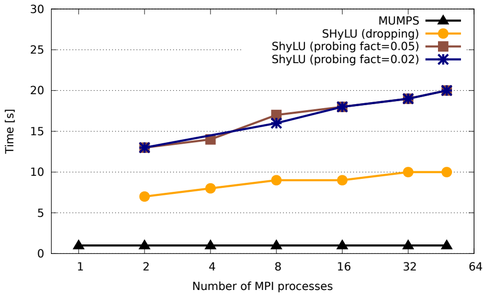
(c) 200k DOF

Figure 3.11: Time to compute the preconditioner - 3D ADR problems



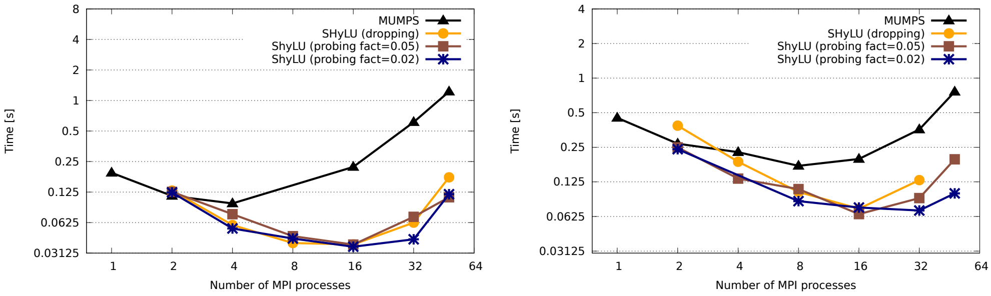
(a) 50k DOF

(b) 100k DOF



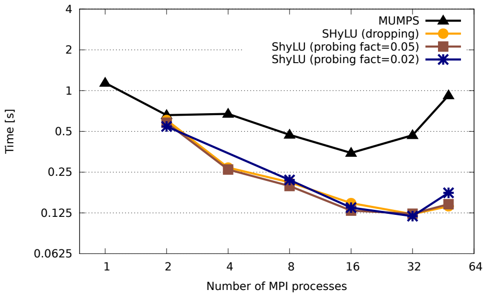
(c) 200k DOF

Figure 3.12: Number of GMRES iterations - 3D ADR problems



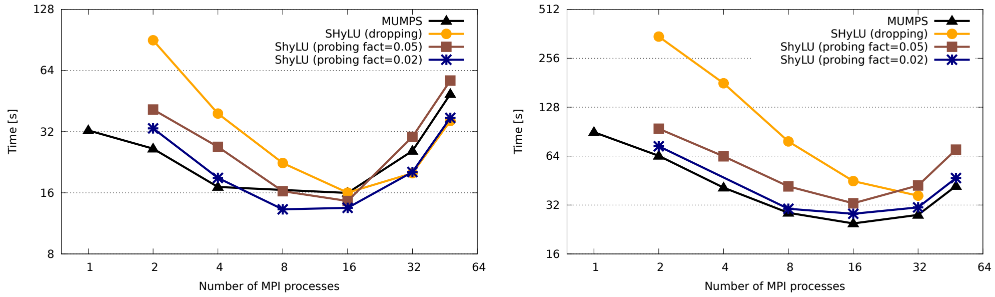
(a) 50k DOF

(b) 100k DOF



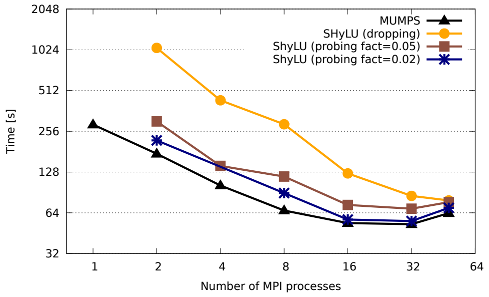
(c) 200k DOF

Figure 3.13: Time per GMRES iteration - 3D ADR problems



(a) 50k DOF

(b) 100k DOF



(c) 200k DOF

Figure 3.14: Total time to solution - 3D ADR problems

3.4.2 Multi-node measurements

Performing the single node measurements previously discussed allowed us to identify the most advantageous parameters for ShyLU. The benchmarks in this section build upon the previous measurements with the goal of determining the performance of the 2-level AAS preconditioning with parallel subdomain problems, using either ShyLU or MUMPS on the subdomain problems.

Two supercomputers were used for this set of benchmarks:

1. **Cray XE6 "Monte Rosa"** - at the Swiss Centre for Scientific Computing (CSCS) in Lugano. "Monte Rosa" is composed of 1496 compute nodes, each one equipped with 2 16-core AMD Opteron "Interlagos" processors and 32 GB of RAM.
2. **BlueGene/Q - "Lemanicus"** - at the Center for Advanced Modeling Science (CADMOS), hosted at EPFL. "Lemanicus" consists of 1024 nodes, each equipped with a 16-core PowerA2 processor and 16 GB RAM. The PowerA2 processor is able to support up to 4 threads per CPU core (64 threads).

We study Laplacian and ADR problems on regular cubic grids, discretized by P2 finite elements. For each problem type we consider two problem sizes: 500000 DOFs and 2 million DOFs. On the BlueGene/Q, we were only able to solve the smaller of the two problems.

For each problem type and problem size, a variable number of MPI processes are used, ranging from 16 to 512 in the case of the smaller problems and from 32 to 1024 in the case of the larger problems.

The linear system is solved using GMRES preconditioned by a 2-level AAS preconditioner with minimal overlap (as described at the beginning of this chapter). At the coarse level, 5 iterations of Gauss-Seidel are applied. We consider three different setups for the subdomain solvers used by AAS:

1. **Serial subdomain problems solved exactly with an exact LU factorization**
   This setup is used as a reference point for the other two strategies. To use the notation introduced in section 3.2, the number of processes per AAS subdomain is always $N_S = 1$, and the number of AAS subdomains is equal to the number of MPI processes ($N_{DD} = N_P$). On the Cray XE6 we use the PARDISO direct solver as a subdomain solver. PARDISO is not available on the BlueGene/Q, where we use UMFPACK instead.
2. **Parallel subdomain problems solved exactly with MUMPS** In this setup the number of AAS subdomains is kept constant at $N_{DD} = 32$, while the number of processes per subdomain is increased along with the total number of MPI processes in use ($N_S = N_P / N_{DD}$). The MPI parallel direct solver MUMPS is used to solve exactly the subdomain problems. MUMPS is unable to solve the largest problems in this configuration, crashing

with internal errors and another MPI parallel direct solver is not available for use on the benchmark machine. This setup is not tested on the BlueGene/Q due to software stability issues related to the MUMPS solver on this machine.

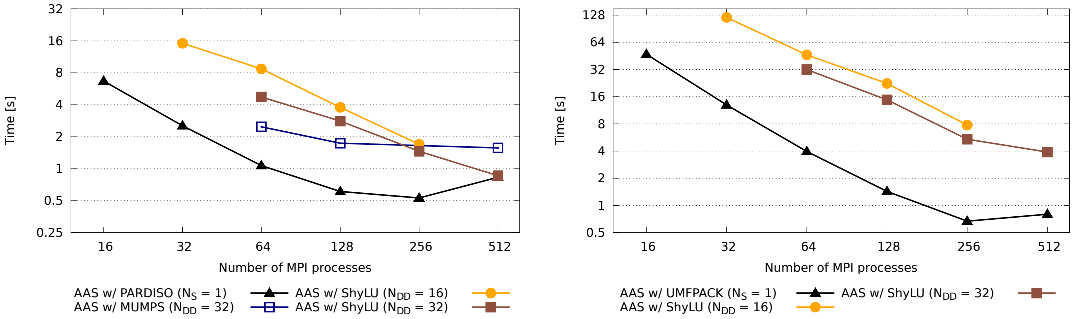
3. **Parallel subdomain problems solved inexactly with ShyLU** This setup is similar to the previous one. The number of AAS subdomains is fixed (we consider $N_{DD} = 16$ and $N_{DD} = 32$ for the small problems and $N_{DD} = 32$ and $N_{DD} = 64$ for the large problems) and the number of processes per subdomain is increased along with the total number of MPI processes ($N_S = N_P / N_{DD}$). The subdomain problem is solved inexactly with ShyLU. PARDISO, on the Cray XE6, and UMFPACK, on the BlueGene/Q, are used to solve the diagonal block problems in ShyLU, while the Schur complement is approximated using the probing technique, with the most favorable bandwidth factor (0.02) from the previous set of measurement. The approximate Schur complement problem is solved inexactly, using 5 iterations of GMRES using ILU as a preconditioner.

For each problem size a variable number of MPI processes is used, therefore this set of benchmarks describes the strong scalability of the global solve strategy. Additionally, the serial subdomain problems decrease in size when the number of MPI processes increases. The parallel subdomain problems are made up of multiple serial problems and in this benchmark the number of MPI processes per subdomain increases with the total number of MPI processes. This results in both ShyLU and MUMPS also being strongly scaled for this experiment. Finally, in the case of MUMPS and ShyLU, the total number of AAS subdomains is kept constant, while growing the number of MPI processes. By examining the number of iterations in the linear solver we get an idea about the preconditioner scalability metric of this configuration of AAS.

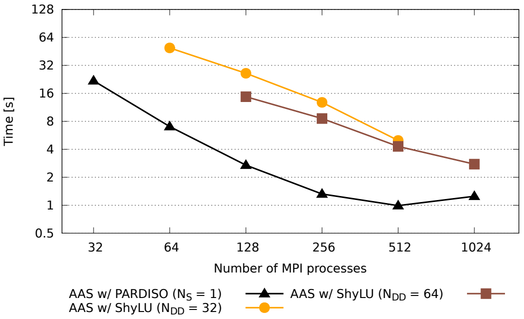
In each case, we measure the time to compute the preconditioners, the number of iterations performed by the GMRES solver and the time to GMRES convergence. Using these measurements, we compute two additional values: the average time per GMRES iteration, proportional to the time to perform one application of the preconditioner and the total time to solution as the sum between the time to compute the preconditioner and the time to GMRES convergence.

Laplacian problems

The measurements of the time to compute the preconditioners (Figure 3.15) show that using parallel subdomain problems leads to a longer time to compute the preconditioner than using serial subdomain problems. However, ShyLU again demonstrates better scalability than MUMPS. When the size of the serial subdomain problems becomes small enough, starting with 256 MPI processes for the small problem and 512 processes for the large one, the time to compute the AAS preconditioner configured with serial subdomain problems starts to increase. Due to the larger parallel subdomain problems, ShyLU does not lose scalability at this process count and actually takes an equal time to compute as when using an exact LU solver on the serial subdomain problems.



(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q



(c) 2M DOF - Cray XE6

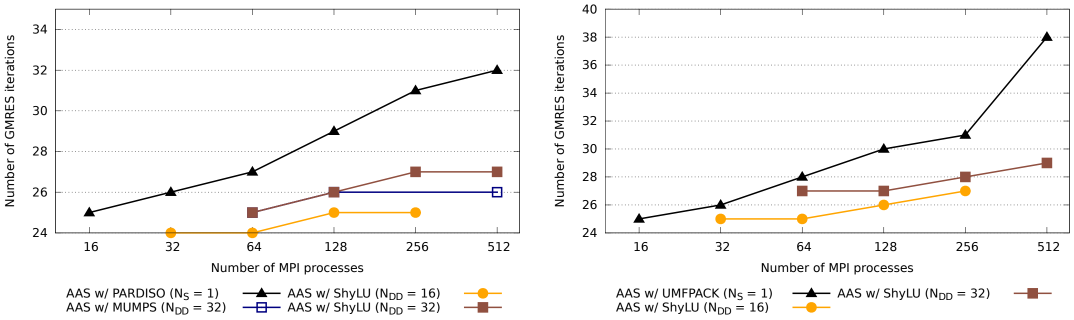
Figure 3.15: Time to compute the preconditioner - 3D Laplacian problems

When using parallel subdomain problems, the number of GMRES iterations to convergence is better kept under control (Figure 3.16). Due to the approximations performed during the Schur complement algorithm, we see that in the case of ShyLU the iteration creep is more pronounced than in the case of MUMPS.

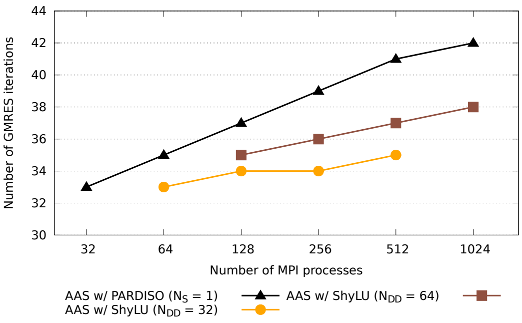
Measuring the time to GMRES convergence we see that AAS with ShyLU and AAS with serial subdomains have more or less the same scalability, although ShyLU is consistently slower. While AAS with MUMPS starts out faster than AAS with ShyLU when N_S is low, the poor scalability demonstrated by MUMPS leads to ShyLU being equally fast as MUMPS at $N_S = 16$.

The measurement of the time per GMRES iteration (Figure 3.18) shows the same trends as in the measurements of the time to GMRES convergence.

Overall, using AAS with parallel subdomain problems and ShyLU could be beneficial at a high process count where the size of the serial subdomain problems is too small for the standard AAS preconditioner to be effective. The AAS / ShyLU approach still achieves strong scalability in this situation (Figure 3.19).

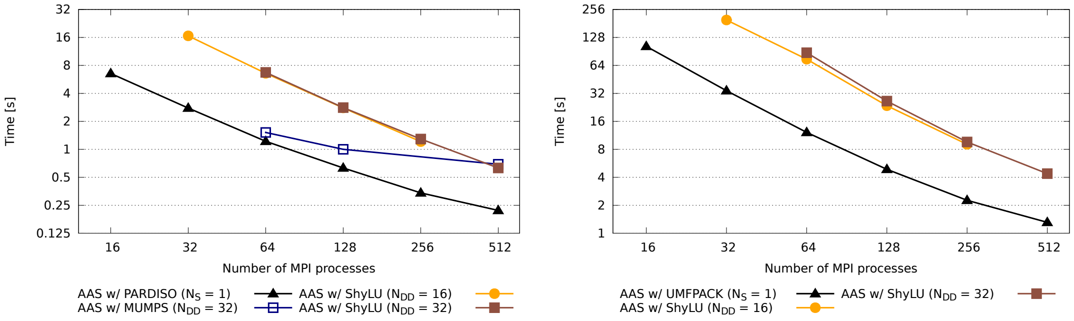


(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q

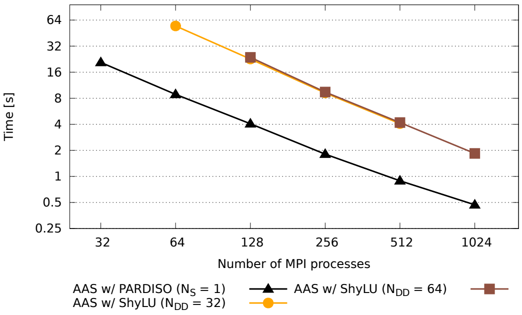


(c) 2M DOF - Cray XE6

Figure 3.16: Number of GMRES iterations - 3D Laplacian problems

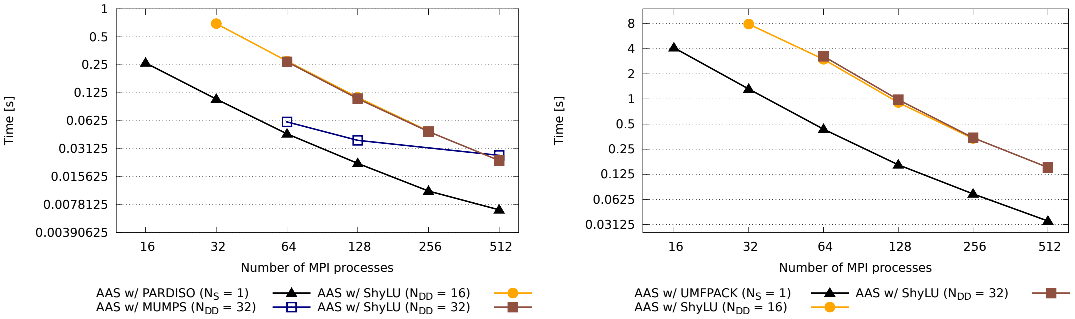


(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q



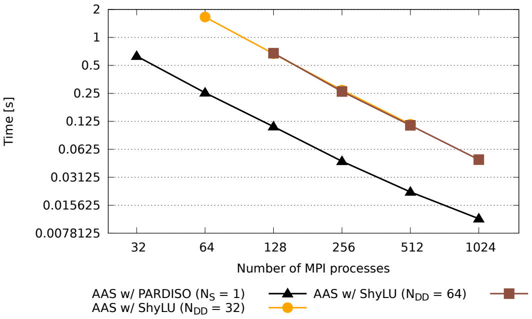
(c) 2M DOF - Cray XE6

Figure 3.17: Time to GMRES convergence - 3D Laplacian problems



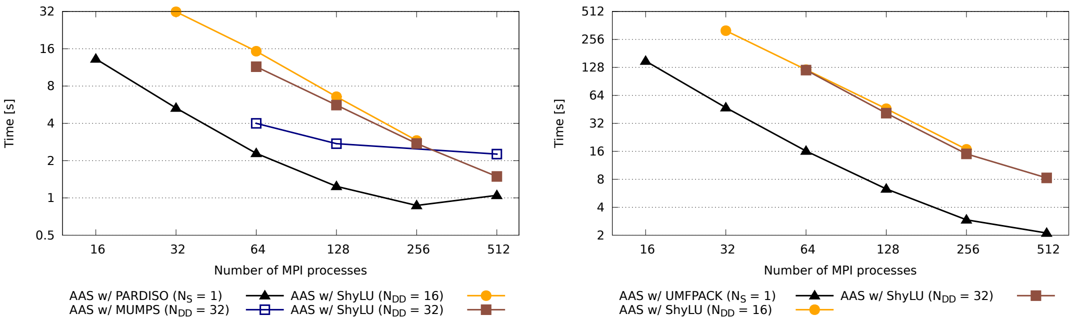
(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



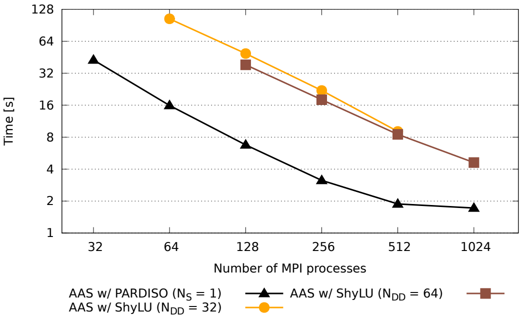
(c) 2M DOF - Cray XE6

Figure 3.18: Time per GMRES iteration - 3D Laplacian problems



(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q

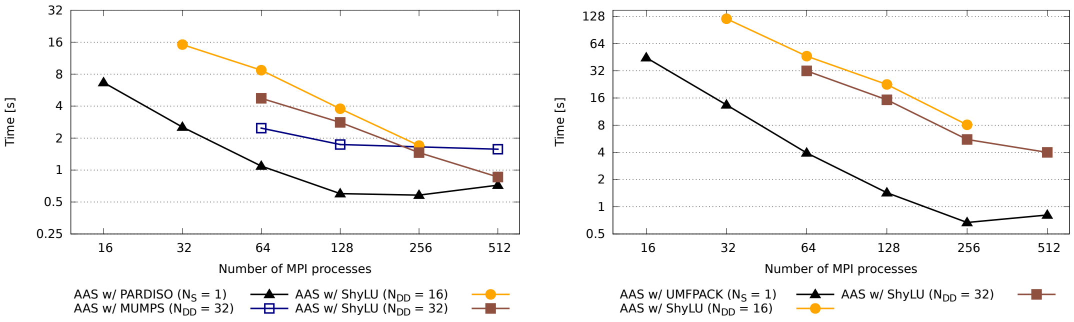


(c) 2M DOF - Cray XE6

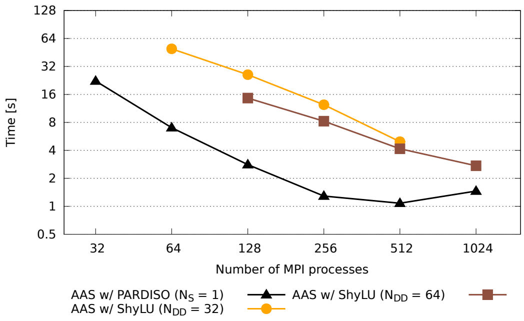
Figure 3.19: Total time to solution - 3D Laplacian problems

Advection-diffusion-reaction problems

As in the case of the single node measurements, the graphs for the ADR problems are nearly identical to the Laplacian case. ShyLU has the same behaviour and performance in both the symmetric Laplacian case and in the unsymmetric ADR case.

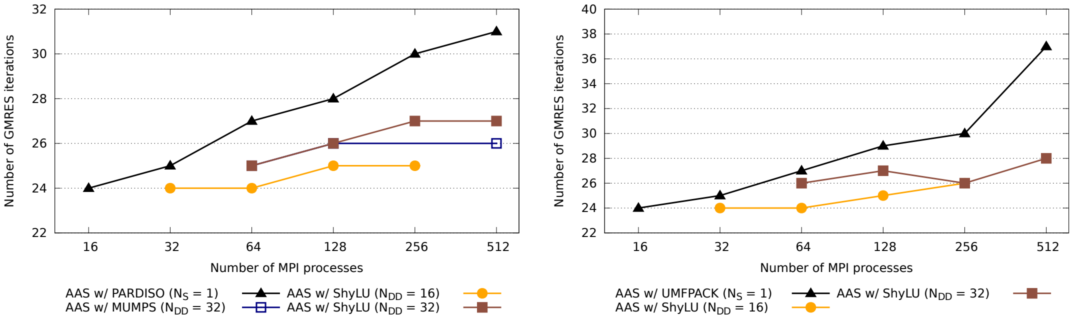


(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q

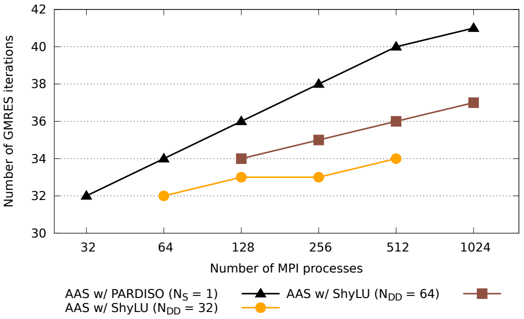


(c) 2M DOF - Cray XE6

Figure 3.20: Time to compute the preconditioner - 3D ADR problems

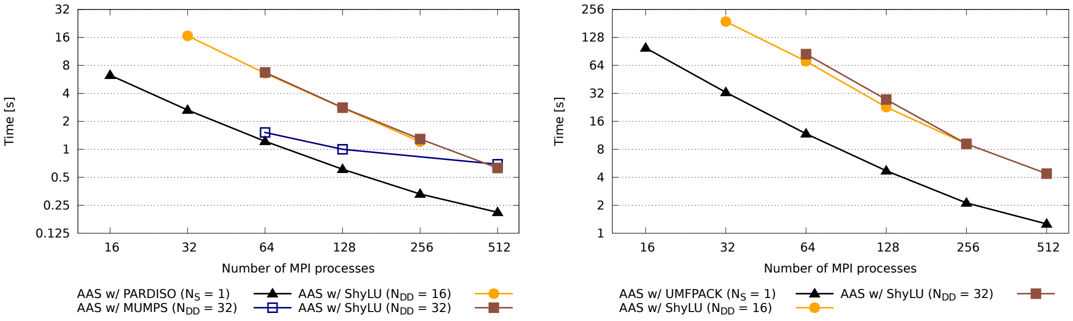


(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q

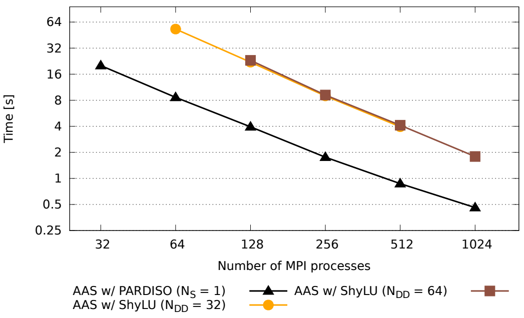


(c) 2M DOF - Cray XE6

Figure 3.21: Number of GMRES iterations - 3D ADR problems

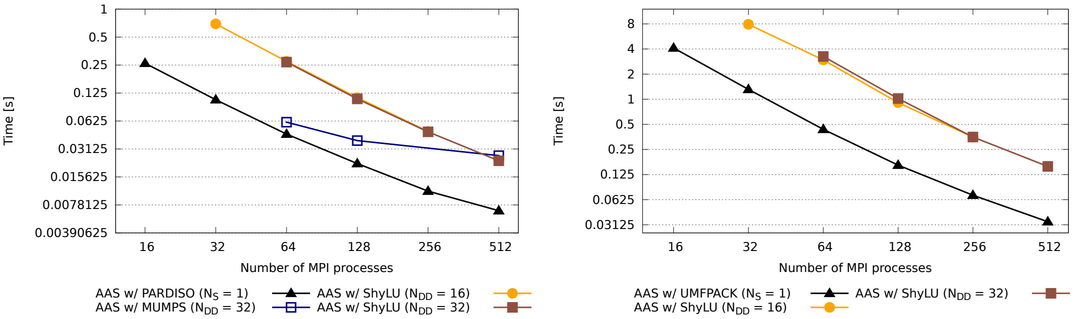


(a) 500k DOF - Cray XE6 (b) 500k DOF - BlueGene/Q



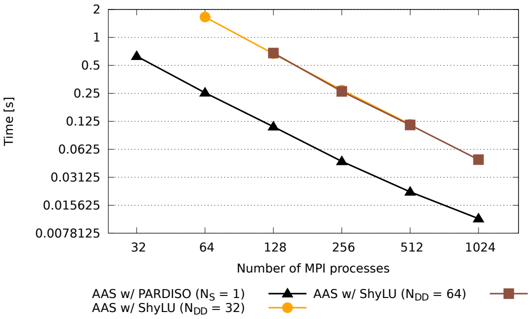
(c) 2M DOF - Cray XE6

Figure 3.22: Time to GMRES convergence - 3D ADR problems



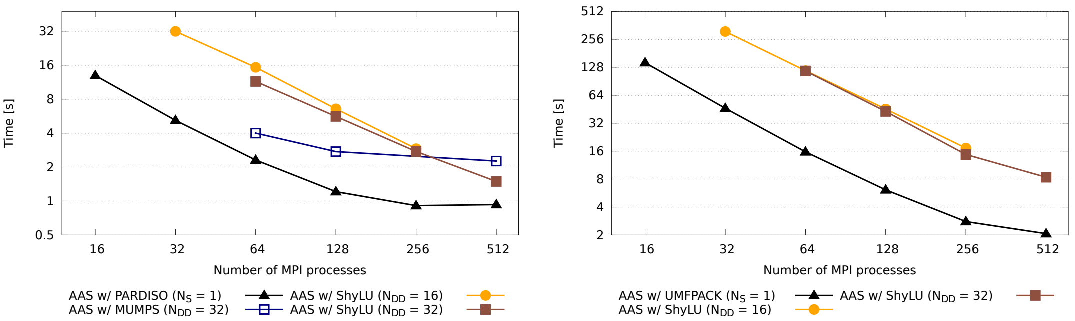
(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



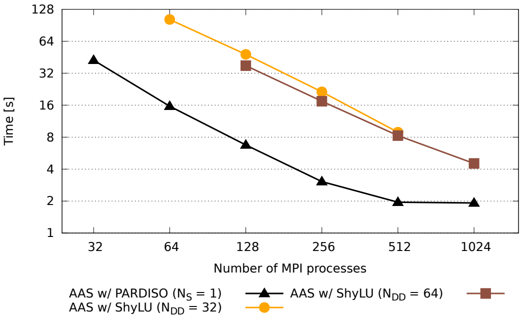
(c) 2M DOF - Cray XE6

Figure 3.23: Time per GMRES iteration - 3D ADR problems



(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



(c) 2M DOF - Cray XE6

Figure 3.24: Total time to solution - 3D ADR problems

3.5 Closing remarks

In this chapter we have introduced a new strategy to construct the AAS preconditioner, based on parallel subdomain problems and two levels of MPI parallelism. This was done in an attempt to improve the strong scalability of the default AAS preconditioner, which is based on serial subdomain problems.

The parallel subdomain problems are stored as distributed sparse matrices and can be solved with an MPI parallel solver. We have examined two different subdomain solvers for use in the new AAS preconditioner framework. The first is an exact parallel LU factorization, as implemented in MUMPS. The second is ShyLU, an inexact solver based on a Schur complement strategy.

The AAS preconditioner with parallel subdomain problems was benchmarked and compared to AAS with serial subdomain problems. We have seen that ShyLU has better strong scalability than MUMPS and is able to solve larger subdomain problems. Although it involves an inexact solve of the subdomain problems, using AAS with ShyLU leads to a lower rise in the number of GMRES iterations when increasing the number of MPI processes, than in the case of AAS with serial subdomains problems.

The performance of the novel approach, in terms of CPU time, is not competitive, however. The time needed to compute the preconditioners and the time to GMRES convergence, in the case of parallel subdomain problems solved with MUMPS or ShyLU, are consistently larger than with serial subdomain problems. The overhead introduced by the 2-level MPI parallelism can not be compensated by the increased strong scalability of a subdomain solver like ShyLU.

4 A preconditioner based on an incomplete QR factorization

4.1 Introduction

In this chapter we follow up on the results and benchmarks presented in Chapter 3. The ShyLU preconditioner was shown to have good strong scalability, although it was generally slower than the two-level AAS preconditioner with serial subdomain problems. We seek to improve the performance of ShyLU by eliminating or ameliorating its main performance hotspots. In this context, we introduce a novel preconditioner based on an incomplete QR factorization and we evaluate its performance when used in conjuction with ShyLU.

4.2 Rationale

Understanding what are the main performance limitations of ShyLU requires a breakdown of the total runtime of the ShyLU preconditioner, by measuring each step in Algorithm 5. We set up a strong scalability test, which involves a 3D Laplacian problem, discretized with P1 finite elements. We have considered three problem sizes: 23000, 45000 and 90000 DOFs. The linear system is solved with GMRES, preconditioned with ShyLU. The AAS domain-decomposition preconditioner is not used during this test and 2, 4 or 8 MPI processes are used for ShyLU. Keeping with the notation introduced in the previous chapter, we have in this case: $N_P = 2, 4, 8$, $N_S = N_P$, $N_{DD} = 1$. The sparse approximation of the Schur complement is computed using the probing method, which was proven to be faster than the threshold dropping method. We examine two different configurations for the treatment of the approximate Schur complement system

$$\bar{S}\mathbf{x_2} = \mathbf{b_2} - R\mathbf{z}. \tag{4.1}$$

In the first configuration, ShyLU solves this system exactly with an LU factorization, using the MPI parallel direct solver MUMPS. In the second one, ShyLU solves the system inexactly, with 5 iterations of the Generalized Minimum Residual Method (GMRES) using ILU as a preconditioner.

	N_S	2	4	8
C	Build blocks D, R, C, G	1.89%	2.98%	2.82%
C	Symbolic factorization of D	5.04%	5.12%	2.99%
C	Numeric factorization of D	16.9%	11.53%	5.22%
C	**Compute approximate $\bar{S}$**	**71.16%**	**61.64%**	**44.41%**
C	**Symbolic and numeric factorization of $\bar{S}$**	**4.10%**	**17.69%**	**43.66%**
S	Solve $D\mathbf{z} = \mathbf{b_1}$	0.43%	0.39%	0.27%
S	**Solve $\bar{S}\mathbf{x_2} = \mathbf{b_2} - R\mathbf{z}$**	**0.05%**	**0.21%**	**0.36%**
S	Solve $D\mathbf{x_1} = \mathbf{b_1} - C\mathbf{x_2}$	0.43%	0.44%	0.27%

Table 4.1: Breakdown of ShyLU runtime. All values represent percentages of the total ShyLU runtime (compute P_{ShyLU}^{-1} and apply P_{ShyLU}^{-1}. The approximate Schur complement system is solved exactly with MUMPS. Values in bold are related to the approximate Schur complement $\bar{S}$. The algorithmic step is listed in the first column: C - computation of P_{ShyLU}^{-1}, S - solution of P_{ShyLU}^{-1}.

	N_S	2	4	8
C	Build blocks D, R, C, G	1.89%	2.98%	2.82%
C	Symbolic factorization of D	5.04%	5.12%	2.99%
C	Numeric factorization of D	16.9%	11.53%	5.22%
C	**Compute approximate $\bar{S}$**	**71.73%**	**70.94%**	**74.22%**
C	**Compute ILU preconditioner for $\bar{S}$**	**0.10%**	**0.14%**	**0.34%**
S	Solve $D\mathbf{z} = \mathbf{b_1}$	0.43%	0.39%	0.27%
S	**Solve $\bar{S}\mathbf{x_2} = \mathbf{b_2} - R\mathbf{z}$**	**3.28%**	**5.37%**	**6.10%**
S	Solve $D\mathbf{x_1} = \mathbf{b_1} - C\mathbf{x_2}$	0.43%	0.44%	0.27%

Table 4.2: Breakdown of ShyLU runtime. All values represent percentages of the total ShyLU runtime (compute P_{ShyLU}^{-1} and apply P_{ShyLU}^{-1}. The approximate Schur complement system is solved inexactly with GMRES subiterations. Values in bold are related to the approximate Schur complement $\bar{S}$. The algorithmic step is listed in the first column: C - computation of P_{ShyLU}^{-1}, S - solution of P_{ShyLU}^{-1}.

The operations measured belong either to the computation or the solution steps of the algorithm. The computation step involves setting up all the data structures, the symbolic and numeric factorizations of the diagonal blocks in matrix D, as well as the computation of a sparse approximation of the Schur complement S. The final operation involved in the computation step depends on the solution strategy chosen for the approximate Schur complement system. If a direct solver is used, then during the computation step the symbolic and numeric factorization of the approximate Schur complement $\bar{S}$ is also performed. If GMRES subiterations are used, then the preconditioner (here ILU) is computed.

The computation step takes place before the solution phase of the global linear system of equations and is performed once per time step, in a time dependent simulation, if the preconditioner is not reused for multi time steps. It should be noted that ShyLU reuses, when possible, internal data structures such as the symbolic factorizations. The implementation of the preconditioner for the iterative solver of the global linear system in LifeV does not make use of this, however. A preconditioner can be reused, without any update, for multiple time steps, but if it is not reused, then it is completely reinitialized and all the intermediate computations are performed again.

The solution step takes place at each iteration of the global iterative linear solver and it represents the application of the inverse of the preconditioner. In the case of ShyLU, it involves the resolution of three linear systems of equations:

$$\begin{aligned} D\mathbf{z} &= \mathbf{b_1}, \\ \bar{S}\mathbf{x_2} &= \mathbf{b_2} - R\mathbf{z}, \\ D\mathbf{x_1} &= \mathbf{b_1} - C\mathbf{x_2}. \end{aligned} \tag{4.2}$$

Tables 4.1 and 4.2 contain the measurements for the two configurations of ShyLU which we examined. The breakdown of the total runtime of ShyLU is similar for all three problem sizes, consequently we only present the case of 45000 DOFs. We see that most of the runtime is spent computing the approximate Schur complement:

$$\bar{S} \approx G - RD^{-1}C. \tag{4.3}$$

When using an exact direct solver for the Schur complement system, an equal amount of time is additionally spent factorizing $\bar{S}$. We conclude that a faster means of computing this approximation would greatly benefit the overall performance of the ShyLU preconditioner.

4.3 The incomplete QR factorization

In this section, we propose an alternative for the computation of the approximate Schur complement and the inexact solution of the associated linear system of equations. The strategy is based on an incomplete QR factorization which is obtained using a customized implementation of a GMRES iterative solver. Although we intend to use the incomplete QR factorization for the approximate Schur complement linear system, the method is general and can also be used for linear systems associated with other types of problems. We begin with an overview of the GMRES method, before describing the incomplete QR factorization algorithm.

4.3.1 The Generalized Minimum Residual Method

The Generalized Minimum Residual Method (GMRES) [93] [35] is an iterative method for solving linear system of equations, part of the family of Krylov subspace projection methods.

Let us consider the linear problem:

$$\text{find} \quad \mathbf{x} \in \mathbb{R}^m : A\mathbf{x} = \mathbf{b}, \quad A \in \mathbb{R}^{m\times m}, \quad \mathbf{b} \in \mathbb{R}^m. \tag{4.4}$$

After n iterations of GMRES, the approximate solution $\mathbf{x_n}$ is sought in the n-th Krylov subspace:

$$K_n = \text{span}\left\{\mathbf{r_0}, A\mathbf{r_0}, A^2\mathbf{r_0}, ...A^{n-1}\mathbf{r_0}\right\} \subset \mathbb{R}^m, \tag{4.5}$$

where $\mathbf{r_0} = \mathbf{b} - A\mathbf{x_0}$ is the residual corresponding to the initial guess $\mathbf{x_0}$ of the solution. The approximate solution $\mathbf{x_n} \in K_n$ produced by GMRES is one that minimizes the residual $||A\mathbf{x_n} - \mathbf{b}||$.

Each GMRES iteration involves an Arnoldi iteration, which produces an orthogonal basis $\mathbf{v_1}, \mathbf{v_2}, ...\mathbf{v_n}$ of the current Krylov space. The vectors of the basis are stored as columns in the matrix $V_n \in \mathbb{R}^{m\times n}$. Additionally, through the Arnoldi iterations we obtain an upper Hessenberg matrix $\bar{H}_n \in \mathbb{R}^{(n+1)\times n}$ with the property:

$$AV_n = V_{n+1}\bar{H}_n. \tag{4.6}$$

Given that $\mathbf{x_n} = V_n\mathbf{y_n}$, $\mathbf{y_n} \in \mathbb{R}^n$ and that the matrix V_n is orthogonal, the following holds:

$$||A\mathbf{x_n} - \mathbf{b}|| = ||\bar{H}_n\mathbf{y_n} - \beta\mathbf{e_1}||, \tag{4.7}$$

where $\beta = \mathbf{r_0}/||\mathbf{r_0}||$ and $\mathbf{e_1} = (1, 0, 0, ...0)$ is the first vector of the cannonical basis of $\mathbb{R}^{n+1}$. Indeed, $\mathbf{r_0} = \beta\mathbf{v_1}$ by construction of the Krylov basis.

Computing the approximate solution $\mathbf{x_n}$ after n GMRES iterations is a two step process. First, find $\mathbf{y_n}$ that minimizes $||\bar{H}_n\mathbf{y} - \beta\mathbf{e_1}||$, which involves solving a linear least squares problem of size n, with n usually much smaller than the original problem size m. Finally, compute the solution as: $\mathbf{x_n} = V_n\mathbf{y_n}$.

Solving the linear least squares problem

The linear least squares problem

$$\mathbf{y_n} = \underset{\mathbf{y}\in\mathbb{R}^n}{\arg\min} ||\bar{H}_n\mathbf{y} - \beta\mathbf{e_1}|| \tag{4.8}$$

is solved using a QR factorization of the Hessenberg matrix $\bar{H}_n$. The orthogonal matrix

$Q_n \in \mathbb{R}^{(n+1)\times(n+1)}$ can be computed as the product of a series of matrices

$$\Omega_i = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_i & s_i & & & \\ & & & -s_i & c_i & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{pmatrix}, \quad i = 1,..,n, \quad \Omega_i \in \mathbb{R}^{(n+1)\times(n+1)}, \tag{4.9}$$

with $c_i^2 + s_i^2 = 1$. The coefficients (c_i, s_i) are on row i, while $(-s_i, c_i)$ are on row $i+1$. Each matrix Ω_i represents a Givens rotation that eliminates the coefficient $h_{i+1,i}$ of the Hessenberg matrix. In practice, the individual matrices Ω_i need not be stored explicitly, as only the sequence of coefficients c_i and s_i needs to be known to compute the action of Q_n. Multiplying Q_n with $\bar{H}_n$ will yield an upper triangular matrix $\bar{R}_n \in \mathbb{R}^{(n+1)\times n}$:

$$\bar{R}_n = Q_n \bar{H}_n = \Omega_1 \Omega_2 ... \Omega_n \bar{H}_n, \tag{4.10}$$

with zeroes on the last row:

$$\bar{R}_n = \begin{pmatrix} R_n \\ 0 \end{pmatrix}. \tag{4.11}$$

Since Q_n is orthogonal, the linear least squares problem becomes:

$$\mathbf{y_n} = \underset{\mathbf{y}\in\mathbb{R}^n}{\arg\min} ||\bar{H}_n \mathbf{y} - \beta \mathbf{e_1}|| = ||Q_n^T \bar{R}_n \mathbf{y} - \beta \mathbf{e_1}|| = ||\bar{R}_n \mathbf{y} - \mathbf{\bar{g}_n}||, \tag{4.12}$$

where $\mathbf{\bar{g}_n} = Q_n \beta \mathbf{e_1} \in \mathbb{R}^{n+1}$. The last row of the matrix $\bar{R}_n$ and right hand side $\mathbf{\bar{g}_n}$ are eliminated, formally $R_n = \Pi_n \bar{R}_n$ and $\mathbf{g_n} = \Pi_n \mathbf{\bar{g}_n}$, where Π_n is the natural projection from $\mathbb{R}^{n+1}$ to $\mathbb{R}^n$. Solving the following upper triangular system using backward substitution:

$$R_n \mathbf{y_n} = \mathbf{g_n}, \tag{4.13}$$

we obtain the solution $\mathbf{y_n}$ of the original linear least squares problem and the approximation $\mathbf{x_n} \in \mathbb{R}^m$ is computed as $\mathbf{x_n} = V_n \mathbf{y_n}$.

4.3.2 The incomplete QR factorization as a preconditioner

The incomplete QR (IQR) factorization preconditioner is based on the idea of using the iterates constructed during a GMRES solution process as a preconditioner to solve another linear system, originally described in [94] and further developed here. We consider two linear systems:

$$\begin{aligned} A_1\mathbf{z_1} &= \mathbf{g_1}, \\ A_2\mathbf{z_2} &= \mathbf{g_2}, \end{aligned} \tag{4.14}$$

where $A_1, A_2 \in \mathbb{R}^{m\times m}$, $\mathbf{z_1}, \mathbf{z_2}, \mathbf{g_1}, \mathbf{g_2} \in \mathbb{R}^m$ and A_1 and A_2 have similar spectral properties. An example of such a situation is inverting the Jacobian in a nonlinear solver.

Solving the first linear system, exactly or inexactly, using n iterations of GMRES, we obtain matrix V_n, with columns which form an orthogonal basis of the Krylov subspace K_n, and the matrices Q_n and R_n which represent the QR factorization of the Hessenberg matrix constructed by GMRES and of the restriction of A_1 on K_n:

$$A_1|K_n : K_n \to \mathrm{Im}(A_1|K_n) \subset K_{n+1} \subset \mathbb{R}^m \tag{4.15}$$

Once the first linear system is solved, we wish to reuse these components to define a preconditioner for A_2. The right hand side $\mathbf{g_2}$ of the second linear system is first projected on $\mathrm{Im}(A_1|K_n)$, which is noted as $\Pi_{\mathrm{Im}(A_1|K_n)}\mathbf{g_2}$. Then $\bar{\mathbf{g}}_\mathbf{n} \in \mathbb{R}^{n+1}$ is defined by

$$V_{n+1}Q_n^T\bar{\mathbf{g}}_\mathbf{n} = \Pi_{\mathrm{Im}(A_1|K_n)}\mathbf{g_2}, \tag{4.16}$$

i.e., $\bar{\mathbf{g}}_\mathbf{n} = Q_N V_{n+1}^T\mathbf{g_2}$. Solving $R_n\mathbf{y_n} = \mathbf{g_n}$ for $\mathbf{g_n} = \Pi\bar{\mathbf{g}}_\mathbf{n}$ and setting $\mathbf{z_n} = V_n\mathbf{y_n}$ yields $(A_1|K_n)^{-1}\Pi_{\mathrm{Im}(A_1|K_n)}\mathbf{g_2}$.

An additional term is added to make the preconditioner non-singular:

$$\frac{1}{\lambda}(\mathbf{g_2} - \Pi_{\mathrm{Im}(A_1|K_n)}\mathbf{g_2}), \tag{4.17}$$

where λ is a scalar to be chosen.

The full expression of the preconditioner reads:

$$P^{-1} = (A_1|K_n)^{-1}\Pi_{\mathrm{Im}(A_1|K_n)} + \frac{1}{\lambda}(\mathrm{Id} - \Pi_{\mathrm{Im}(A_1|K_n)}). \tag{4.18}$$

In algebraic form this reads

$$P^{-1} = V_nR_n^{-1}\Pi_nQ_nV_{n+1}^T + \frac{1}{\lambda}(\mathbb{1}_n - V_{n+1}V_{n+1}^T + V_{n+1}Q_n^T(\mathbb{1}_{n+1} - \bar{\Pi}_n)Q_nV_{n+1}^T), \tag{4.19}$$

where $\bar{\Pi}_n$ is the projection which sets the last coordinate to zero and $\mathbb{1}_n$ and $\mathbb{1}_{n+1}$ are the identity matrices in $\mathbb{R}^n$ and $\mathbb{R}^{n+1}$ respectively. Note that $V_{n+1}V_{n+1}^T$ represents the projection on K_{n+1} and the last part is the complement of the orthogonal projection from K_{n+1} to $\mathrm{Im}(A_1|K_n)$. Algorithm 6 details the steps to apply the preconditioner to a vector $\mathbf{b}$.

The preconditioner represents an *incomplete QR factorization* (IQR) of the system matrix of the first linear system. In the limit case that the size of the Krylov subspace n is equal to the problem size m, i.e. GMRES solves the first system to absolute precision, the preconditioner becomes a full QR factorization of A_1. In [94] it is shown that the IQR is non-singular.

4.4 Using IQR within ShyLU

The IQR as preconditioner is not limited in use to sequences of similar linear systems. In this section we describe its use within the ShyLU preconditioner, as a cheaper alternative to the existing resolution strategies for the Schur complement system.

Unlike the threshold dropping strategy and the probing strategy for computing $\bar{S}$, the IQR preconditioner has the advantage of being a matrix free method. No explicit computation of an approximation of the Schur complement S is required, as only its action on a vector needs to be computed. In ShyLU, S is implemented as an opaque operator object, which implements an "Apply" method (i.e. $\mathbf{x} = \mathbf{S}\mathbf{y}$), and this object can be used without any modification for IQR.

The existing implementations of the GMRES method in Trilinos do not offer the possibility of preserving the state of the solver after resolution and using the generated V_n, Q_n and R_n matrices in the manner described in the previous section. We have implemented a custom version of GMRES with a persistent state manager, which we use to compute and apply the IQR preconditioner. The new version is parallel, using MPI, and is compatible with the Trilinos parallel matrix and vector classes.

We describe now the specific steps of using IQR inside ShyLU. The ShyLU computation step is freed of the computation of $\bar{S}$ and only the opaque operator object for S is constructed in this phase. This considerably reduces the CPU time for computing the global preconditioner.

The IQR preconditioner (noted P_{IQR}^{-1}) is computed in the solution step of ShyLU, when the ShyLU preconditioner is applied to a vector during the first outer GMRES iteration. During this first iteration, the GMRES solver embedded in P_{IQR}^{-1} executes a prescribed number n of iterations and stores the V_n, Q_n and R_n matrices needed during the next outer GMRES iterations. The number of inner iterations n, which represents the size of the Krylov subspace

```
Compute: $\mathbf{b_p} = V_{n+1}^T \mathbf{b}$
Update: $\mathbf{x} = \mathbf{x} - V_{n+1}\mathbf{b_p}$
Update: $\mathbf{b_p} = Q_n^T \mathbf{b_p}$
Compute: $\mathbf{b_q} = Q_n^T(\mathbb{1}_{n+1} - \bar{\Pi}_n)\mathbf{b_p} = Q_n^T(0,0,0,...\mathbf{b_p}(n+1))^T$
Update: $\mathbf{x} = \frac{1}{\lambda}(\mathbf{x} + V_{n+1}\mathbf{b_q})$
Solve: $R_n \mathbf{y_n} = \Pi_n \mathbf{b_p}$
Return: $\mathbf{x} = \mathbf{x} + V_n \mathbf{y_n}$
```

Algorithm 6: The application of the IQR preconditioner to a vector **b**. V_n, Q_n, R_n are obtained after the GMRES iterations on the first linear system.

K_n and the size of the Q_n and R_n matrices, is imposed as a percentage of the size of the Schur complement matrix. During all the subsequent outer GMRES iterations, the steps outlined in Algorithm 6 are performed.

We provide two different ways to apply P_{IQR}^{-1}:

1. Use it as a preconditioner for one or more GMRES inner iterations.
2. Use it as a matrix free approximation of the inverse of the Schur complement matrix in (3.17), i.e.

   $$\bar{S}\mathbf{u_2} = \mathbf{f_2} - R\mathbf{z}, \tag{4.20}$$

   is replaced by

   $$P_{IQR}\mathbf{u_2} \approx \mathbf{f_2} - R\mathbf{z}. \tag{4.21}$$

   In this case, for the outer GMRES solver to converge, we experienced that the λ scaling parameter of IQR needs to be set to 1 (no scaling).

We have examined both scenarios and we have observed that there is no benefit to using P_{IQR}^{-1} as a preconditioner for inner iterations. Although it leads to a slightly lower number of outer GMRES iterations than when using it as an inexact solver, in both cases we observe the same increase in the number of outer GMRES iteration with the number of MPI processes. Additionally, the CPU time per (outer) GMRES iteration and the total time to (outer) GMRES convergence is larger in the former case. Unless it is explicitly stated otherwise, it should be assumed that IQR is used as an inexact solver in ShyLU in all benchmarks in this thesis.

4.4.1 Numerical benchmarks and discussion

To evaluate the performance of IQR we perform the tests described in Section 3.4.2 To summarize, the tests are Laplacian and (advection-diffusion-reaction) ADR problems (see (3.18) and (3.19)), discretized with P2 finite elements. Two problem sizes are considered: 500000 and 2 million DOFs. Strong scalability measurements are done varying the number of MPI processes N_p. The number of AAS subdomains is kept constant ($N_{DD} = 16$ and $N_{DD} = 32$, for the smaller problems and $N_{DD} = 32$ and $N_{DD} = 64$, for the larger problems), increasing the number of processes per subdomain $N_S = N_P / N_{DD}$.

The linear system of equations is solved with GMRES preconditioned with 2-level AAS preconditioner with minimal overlap. The coarse problem is solved inexactly with 5 Gauss-Seidel iterations. At the fine level, the parallel AAS subdomain problems are solved inexactly with ShyLU, using IQR for the Schur complement system (3.17), as described in (4.21). For the diagonal block solvers, PARDISO is used.

N_P	$N_{DD} = 16$		$N_{DD} = 32$	
	Dimension of Krylov subspace	Size of Schur complement	Dimension of Krylov subspace	Size of Schur complement
32	15	3123		
64	30	6037	8	1705
128	49	990	18	3770
256	71	14356	30	6090
512			43	8774

(a) 500000 DOFs

N_P	$N_{DD} = 32$		$N_{DD} = 64$	
	Dimension of Krylov subspace	Size of Schur complement	Dimension of Krylov subspace	Size of Schur complement
64	21	4337		
128	48	9736	14	2820
256	79	15871	30	6009
512	115	23057	48	9750
1024			74	14815

(b) 2 million DOFs

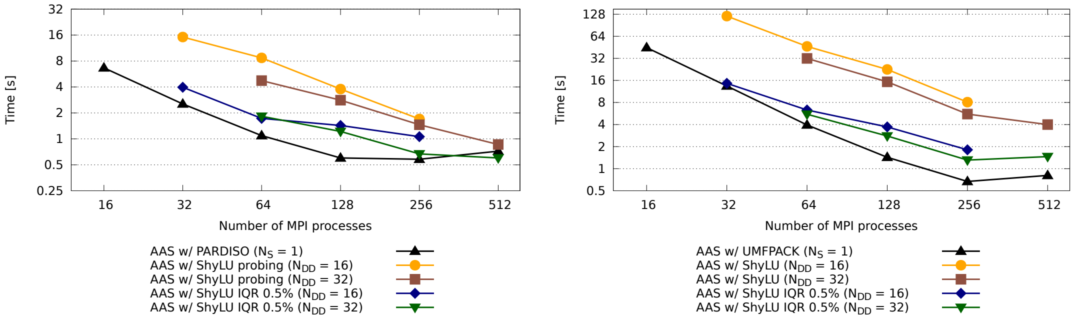
Table 4.3: Dimension of the IQR Krylov subspaces (0.5% of the size of Schur complement matrix) used for the benchmarks. Values represent the average over all the AAS subdomains.

For the IQR factorization, we have considered Krylov subspaces of different dimensions; in the configuration of ShyLU we prescribed the dimension of the Krylov subspace as a percentage of the size of the ShyLU Schur complement matrix S: 0.5%, 1%, 10%, 20%. We observed that using a size larger than 1% is not beneficial, the total CPU time increases without a reduction in outer GMRES iterations. We also reduced even further the dimension of the Krylov subspace but in that situation the number of outer iterations was too large. Additionally, since the results for 1% and 0.5% Krylov subspace sizes are very close to each other, we only plot the measurements for the smallest of the two, to keep the figures readable. Table 4.3 shows the dimension of the Krylov subspaces and sizes of the corresponding ShyLU Schur complement matrices for this case.

The preconditioner behaved equivalently for the Laplacian and the ADR problems, consequently we only present the results for the ADR problems.

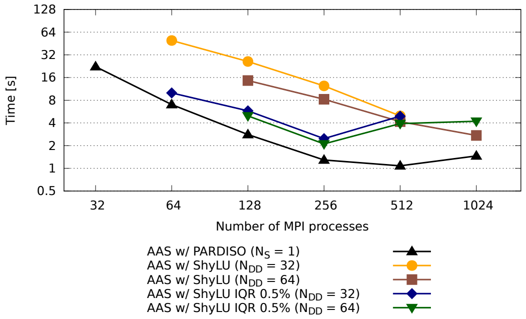
We compare the ShyLU/IQR preconditioner with the best configuration from the previous tests, 2-level AAS with PARDISO, $N_S = 1$ and with the previous ShyLU configuration.

Figure 4.1 shows the time spent in the computation phase of the ShyLU preconditioner. This time is greatly decreased in all cases, using IQR. This is to be expected, since the IQR preconditioner isn't actually computed until the first outer GMRES iteration. In the case of IQR, the time reported in this figure accounts for the time to construct the operator object for the Schur complement matrix S and the time to perform the symbolic and numeric factorization of the diagonal blocks D_i. The size of each serial subdomain problem in the references case



(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



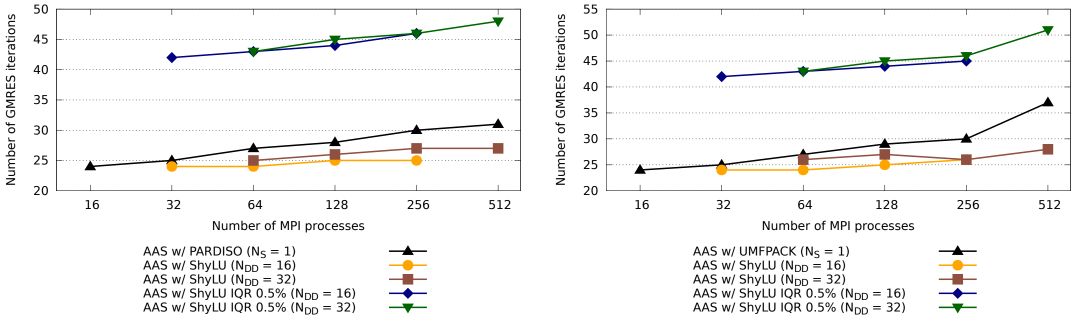
(c) 2M DOF - Cray XE6

Figure 4.1: Time to compute the preconditioner - 3D ADR problems

(AAS with PARDISO, $N_S = 1$) is approximately the same as the size of each diagonal block D_i in ShyLU, due to the mesh partitioning algorithm which will minimize the size of the interface between mesh parts. In this case, computing the ShyLU preconditioner always involves some extra amount of computation work compared to the reference case, due to the added operations involving the Schur complement. The time to compute the preconditioner can be considered a lower bound for the time to compute ShyLU.

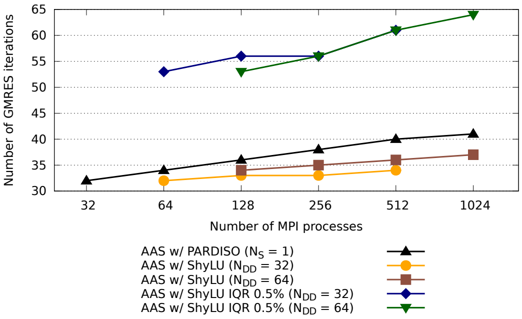
We see that the number of outer GMRES iterations (Figure 4.2), is considerably larger in the case of ShyLU with IQR. The rate of increase of the iterations with the number of MPI processes seems to be similar, though, to ShyLU with a probing method and to AAS with serial subdomain problems solved exactly with an LU factorization.

The total time to GMRES convergence (Figure 4.3), while higher than the AAS/PARDISO reference case, is lower than for ShyLU with a probing approximation of $\bar{S}$. It should be noted that this time also involves the computation of the IQR preconditioner at the first outer GMRES iteration. At a high MPI process count, globally ($N_P = 1024$) and per subdomain ($N_S = 16$), the two configurations of ShyLU achieve the same time to GMRES convergence. Although the configuration of ShyLU with the IQR preconditioner leads to a much larger number of outer GMRES iterations, the fact that the time per GMRES iteration (Figure 4.4) is always lower than the case of ShyLU with probing, leads to the improved time to GMRES convergence.



(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



(c) 2M DOF - Cray XE6

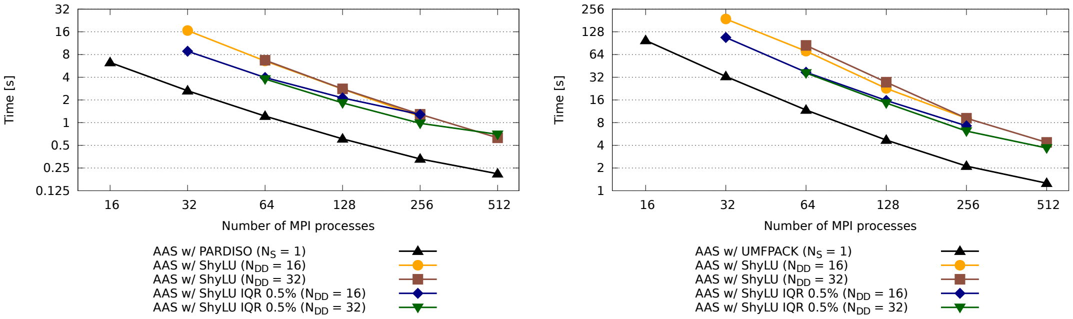
Figure 4.2: Number of GMRES iterations - 3D ADR problems

The same remark applies as for the time to compute the preconditioner. AAS with ShyLU always involves more computation work at each outer GMRES iteration than AAS with serial PARDISO. The time per GMRES iteration for ShyLU will never be lower than in the reference configuration.

Overall, the IQR configuration of ShyLU is on average twice as fast as the previous configuration using a sparse approximation of the Schur complement. We do not observe any reduced reliability, as both configurations of ShyLU have been able to solve all the test cases with all combinations of global and per-subdomain MPI processes.

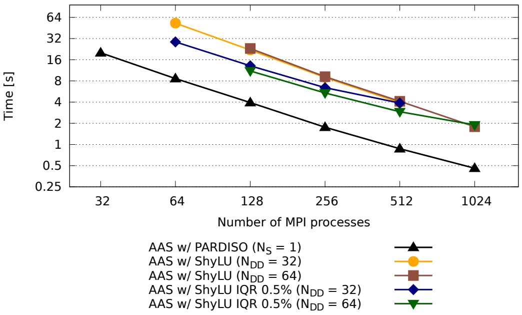
4.5 Conclusions

In this chapter we have described the algorithm of the incomplete QR factorization preconditioner. A parallel implementation of this preconditioner, which we developed in the Trilinos libraries is used as an alternative inexact solution strategy for the Schur complement system in the ShyLU preconditioner. The new implementation was benchmarked and it proved to be an overall improvement over the pre-existing configuration options of ShyLU. The approach based on ShyLU and IQR, although still generally slower than an AAS preconditioner with serial subdomain problems solved with LU factorizations, is on average twice as fast as the fastest configuration of ShyLU identified in the previous chapter. Given the generally good



(a) 500k DOF - Cray XE6

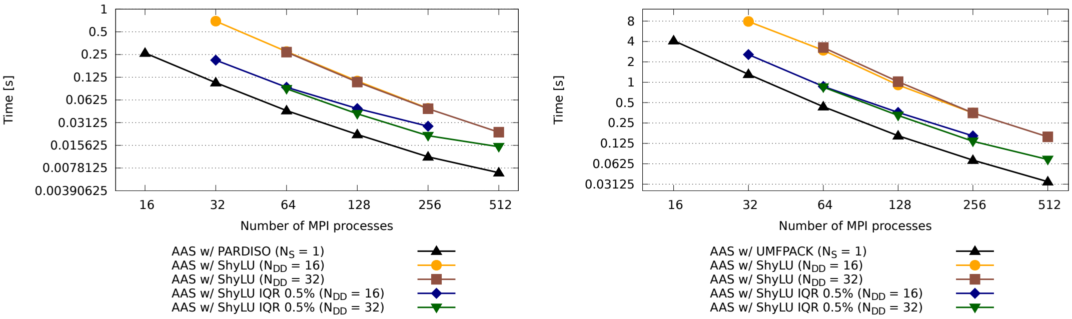
(b) 500k DOF - BlueGene/Q



(c) 2M DOF - Cray XE6

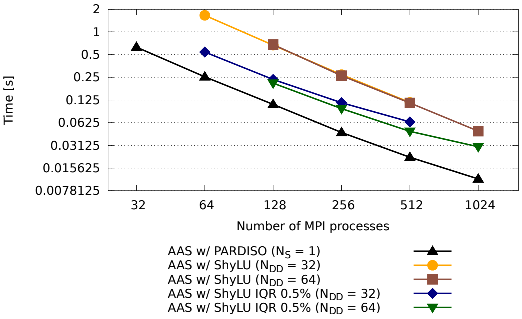
Figure 4.3: Time to GMRES convergence - 3D ADR problems

performance of IQR in these benchmarks, we evaluate it further in Chapter 7, as a component within a larger preconditioner framework for Navier-Stokes problems.



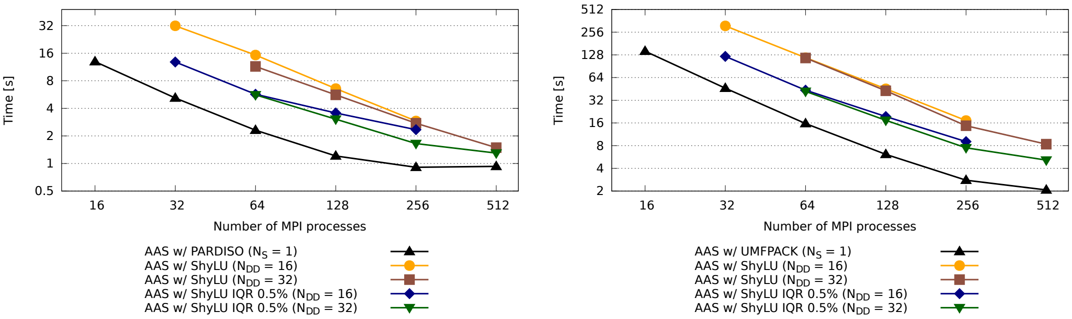
(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



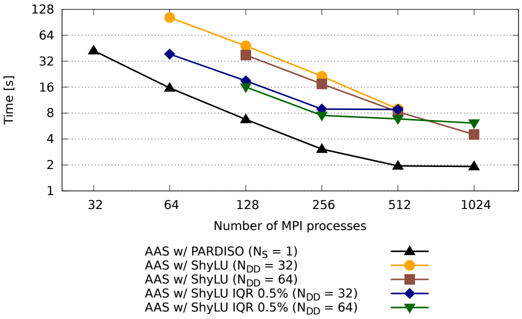
(c) 2M DOF - Cray XE6

Figure 4.4: Time per GMRES iteration - 3D ADR problems



(a) 500k DOF - Cray XE6

(b) 500k DOF - BlueGene/Q



(c) 2M DOF - Cray XE6

Figure 4.5: Total time to solution - 3D ADR problems

5 Preprocessing requirements

5.1 Introduction

In this chapter some aspects related to preprocessing are discussed. It begins with an overview of the mesh partitioning process, as it was also implemented in LifeV at the time when this thesis began, and describes the limitations of this approach. Then, there is a discussion of alternate mesh partitioning techniques which alleviate the problems listed in the second section of the chapter. The final section describes an implementation of efficient mesh loading for parallel simulations.

5.2 Runtime mesh partitioning

During a typical finite element simulation performed with LifeV, there exist two ways to obtain the initial computational mesh of the problem domain. The first one is to generate the mesh procedurally at runtime. The second one is to load the mesh, which has been generated by a different software, from disk, at the beginning of the simulation. The former is appropriate for structured meshes of simple geometries, while for complicated geometries or unstructured meshes it is necessary to resort to the latter.

Each MPI process in a parallel simulation opens the mesh file and builds the mesh object which contains the lists of all the mesh elements, faces, edges and vertices in the global mesh. The mesh object for the global mesh is not a parallel data structure, it is identical on each process. These lists are used to construct the dual graph of the mesh, which describes the connectivity of the mesh elements. Figure 5.1 depicts the dual graph associated with a structured triangular mesh of a square domain.

Mesh partitioning is performed with a graph partitioning library, such as ParMETIS or Zoltan [95], which operates on the dual graph of the mesh. The objective of the graph partitioning is to cut the dual graph into a given number of subgraphs, here equal to the number of MPI processes, while ensuring that the subgraphs are balanced in size and that the number of edge

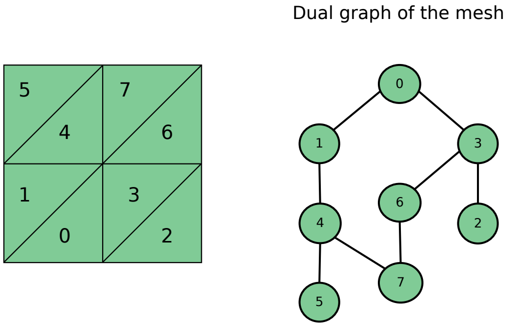


Figure 5.1: The dual graph associated with a structured triangular mesh of a square domain. The nodes of the dual graph correspond to the elements of the mesh and there is an edge between two graph nodes if the mesh elements associated with the graph nodes have a common face.

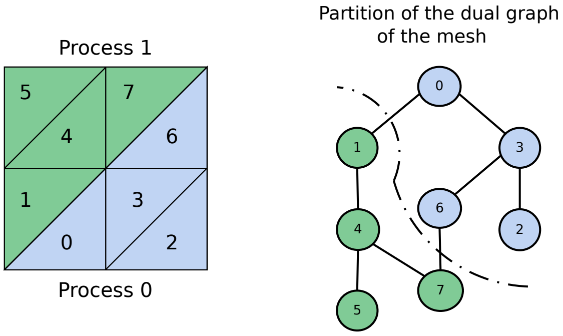
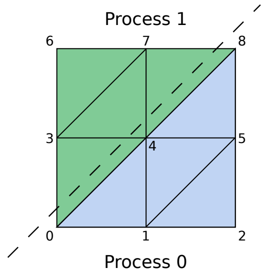


Figure 5.2: The partition of the dual graph associated with a structured triangular mesh of a square domain. The dotted line represents the cut through the dual graph. Blue elements belong to process 0, while green ones belong to process 1.

cuts is minimized. As a result, the interface between the mesh parts (i.e. the set of element faces that the mesh parts share) produced by the partitioning is minimized. In a parallel setting, this leads to a minimal amount of communication between processes. The graph partitioning process is parallel and each process is responsible of a portion of the graph. The processes receive an initial list of elements that make up their subgraph, choice of elements which is arbitrary, and at the end of the process receives the final list of elements in their mesh part. The partitioning of the dual graph associated to a given mesh is illustrated in Figure 5.2. Of course, the final partitioning is not univocal, as different software, or even different implementations, may produce a different partitioning.

With the final list of local elements computed, each process is able to build its own lists of the elements, faces, edges and vertices contained in its mesh part. All the entities are renumbered locally in this new object. At the end of this step, the mesh object corresponding to the original unpartitioned mesh is no longer needed and is discarded, freeing up memory.



	Process 0	Process 1
Unique map:	0, 1, 2, 4, 5, 8	3, 6, 7
Repeated map:	0, 1, 2, 4, 5, 8	0, 3, 4, 6, 7, 8

Figure 5.3: The contents of the unique and repeated maps in the case of a mesh partition between two processes. Blue elements belong to process 0, while green ones belong to process 1. Process 0 receives ownership of the DOFs on the partition interface.

5.2.1 Building the DOF maps

Following the mesh partition process, it is necessary to construct the set of degrees of freedom (DOFs) that make up our discrete problem, according to the choice of finite element space. This step can be done in parallel by each process, on its own mesh part. Both the mesh entities and the DOFs in the local mesh part are identified according to a local numbering scheme. Thus it is necessary to associate the local numbering of the DOFs on each process with a global numbering on the whole problem. This part requires communication between processes and results in two different local-to-global maps for the DOFs, which serve two different purposes in the course of a simulation.

The first map, called unique map, contains the DOFs that are owned by the process. For DOFs which reside on the interface between two or more mesh parts, assigning them to a process is a matter of convention. In LifeV, they are assigned to the process with the lowest MPI rank. The second one, which we call repeated map keeps track of these DOFs on all processes which share this interface, in addition to the unique DOFs. For a simplified example describing the contents of the two types of DOF maps, see Figure 5.3.

The unique map is used when there aren't multiple processes updating the same matrix or vector rows, e.g. during the resolution of the linear problem. The repeated map is needed for finite element assembly operations of matrices and vectors. In this context, there are cases when matrix coefficients are computed on a different process than the one which owns the row of the matrix. In the example given in Figure 5.3, process 0 owns the rows 0, 4 and 8 of the system matrix. The matrix coefficients $a(0,3)$, $a(4,3)$, $a(4,7)$ and $a(8,7)$ are computed by process 1 and communicated to the owner process at the end of the assembly procedure. The repeated map is also used when dealing with nonlinearities. For example, when looping on an element to compute the local matrix entries of the convective term, it is necessary to access the values of the velocity in all the element's DOFs. It can happen that the unique map does not provide them and they are retrieved using the repeated map. The volume of each mesh part decreases much faster than the surface of the interfaces between mesh parts, when the number of parts is increasing. At high process counts, the cost in terms of CPU time of the communication associated with the interface DOFs is no longer negligible compared to the

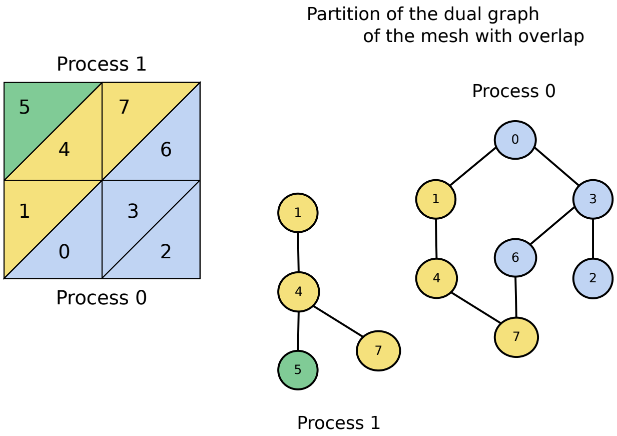


Figure 5.4: Mesh partitioning with overlap. The elements in the overlap region are in yellow. Process 0, owner of the DOFs on the interface, receives the elements in the overlap region to complete the support of the basis functions associated with the interface DOFs. Process 1 does not require additional elements.

cost of the computations involved in the assembly of the linear system.

5.2.2 Overlap support for partitioning

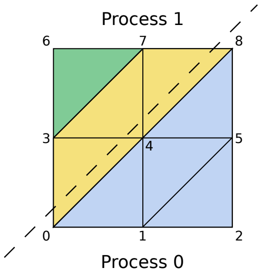
An alternative approach, which removes the communication cost during the finite element assembly, is to partition the mesh with an overlap (see Figure 5.4). This way, on each process, all the tetrahedra which represent the support of the basis functions associated with the DOFs in the unique map are locally available.

Elements in the overlap region are added to the processes which have the interface DOFs in their unique maps. An additional map of DOFs, called ghost map in LifeV, is constructed on each process with the new DOFs from the overlap region. In Figure 5.4, process 1 does not need ghost elements, while process 0 has elements 1, 4 and 7 as ghosts. Figure 5.5 describes the contents of the DOFs in a simple mesh partitioning case with overlap.

In the end, the ghost map is merged into the repeated map. The matrix and vector coefficients associated with interface DOFs are now entirely computed with local DOF information from the unique and repeated maps, without any MPI communication.

5.2.3 Limitation of runtime partitioning

The main bottleneck in the mesh partitioning step is the memory consumption, which is due to the loading of the original unpartitioned mesh on each process. When using a small mesh with not too many elements, the extra memory usage is negligible. Large meshes need usually a large number of MPI processes, however, the memory requirements of the complete mesh may become too limiting. The current trend in supercomputer design is to increase the



	Process 0	Process 1
Unique map:	0, 1, 2, 4, 5, 8	3, 6, 7
Repeated map:	0, 1, 2, 4, 5, 8	0, 3, 4, 6, 7, 8
Ghost map:	3, 7	0, 4, 8

Figure 5.5: The contents of the unique, repeated and ghost maps in the case of a mesh partition with overlap between two processes. DOFs 0, 4 and 8, on the partition interface, belong to process 0. This process requires DOFs 3 and 7 to complete the support of the basis functions on the interface. Process 1 requires DOFs 0, 4 and 8 to complete the support of basis functions associated with DOFs 3 and 7. At the end, the ghost maps are merged into the repeated maps.

number of processor cores available on each node. In contrast, the total memory available per node is increasing at a much slower rate. This leads to a situation where the available memory per processor core and, in an MPI setting, per process, is decreasing. The inefficient memory usage in the mesh partitioning greatly limits the maximum size of simulations that can be performed on supercomputers. The overlapping mesh partitioning removes the MPI communication costs from the FE assembly phase, but it does not improve the memory usage.

5.3 Alternate mesh partitioning techniques

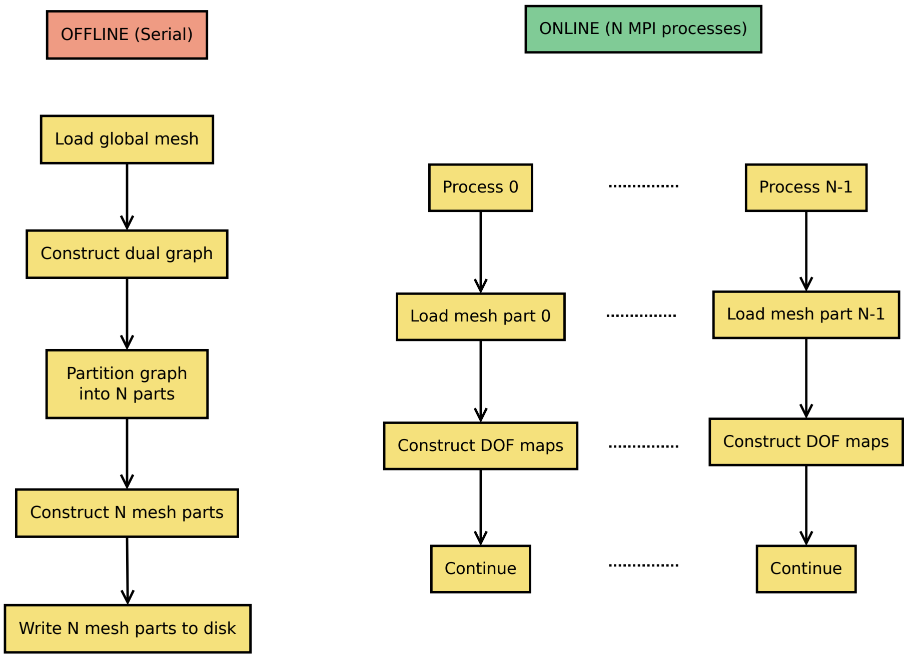
This section describes two mesh partitioning techniques developed for LifeV, in the course of this thesis. The first one improves the memory usage for large simulations, while the second strategy is strictly tied to the use of the ShyLU subdomain preconditioner, which was described in the previous chapter. The two techniques are not mutually exclusive and can be combined when needed.

5.3.1 Offline partitioning

The mesh partitioning strategy described in Section 5.2 is refered to as an *online* strategy, as it takes place entirely during the run-time of the simulation. In an effort to lower the run-time memory requirements, the mesh can be partitioned offline, i.e. before the simulation and possibly on a different computer. In this situation, some data structures need also to be precomputed, stored and loaded online in parallel.

Offline step

The offline step is performed serially on a different machine than the one which will run the simulation. This machine is typically a desktop workstation with a large amount of RAM, of the order of tens of GB, much higher than the 512MB to 2GB of RAM that are available per



(a) Offline stage

(b) Online stage

Figure 5.6: The two steps of the offline partitioning strategy. Using N MPI processes in the online stage.

process on supercomputers. In the extreme cases of very large meshes, this step can be run as a job on a compute cluster that has high memory nodes available, usually with a few hundred GB of RAM per node.

First the global mesh is loaded into memory and the dual graph is constructed. In the online strategy the number of parts for the partitioning is always the same as the number of MPI processes. In this case, the graph is partitioned into the desired number of parts for the future simulation and each mesh part is built in turn. At the end of the offline phase, the original mesh is discarded from memory and all the mesh parts are written to disk. The choice and implementation of the read and write routines for the mesh parts is discussed in Section 5.4. The offline step is summarized in Figure 5.6a.

Online step

All the mesh partitioning operations have been performed in the offline stage. During the run-time of the simulation, each process only needs to read its own mesh part from disk (see Figure 5.6b). Having done this, the simulation continues as in the case of online partitioning.

Benefits and limitations

Moving the mesh partitioning stage offline and performing it serially results in much more efficient memory usage. The global unpartitioned mesh is loaded only once into memory, opposed to once per mesh part in the online strategy. The memory usage is thus independent of the number of mesh parts that are produced. Given the large amount of memory available in workstations compared to the low amount of memory per process that is made available on a supercomputer, we are able to partition much larger meshes with the offline strategy.

A disadvantage of this approach is that the offline mesh partitioning can take a longer amount of time, since it it performed serially. This time is however easily amortized given the fact that a saved partitioned mesh can be reused for multiple runs of a simulation, as long as the number of mesh parts (and, as a result, the number of MPI processes) is unchanged.

5.3.2 Hierarchical partitioning

At the end of the mesh partitioning, no assumptions can be made on the relative position of any two mesh parts, especially in the case of an unstructured mesh of a complicated geometry that is cut into a large number of parts. It is possible that consecutively numbered mesh parts do not have a common interface (no common face, edge or vertex).

In the AAS strategy based on 2 levels of MPI introduced in Chapter 3 the parallel subdomain problems are built by combining the serial subdomain problems assigned to consecutively numbered processes. Consequently, these serial subdomain problems are built using consecutively numbered mesh parts. In the situation where one of these parts is not connected to the others, the ShyLU preconditioner cannot be used, due to the Schur complement framework that is used (the process associated with the isolated mesh part would not have any contribution to the Schur complement.)

The remedy is a hierarchical partitioning: the global mesh is first partitioned into a number of parts equal to the number of subdomains N_{DD} for the AAS preconditioner. A second partitioning is performed individually on each mesh part obtained at the first step, using the number of MPI processes per subdomain N_S as the number of parts. This two step process ensures that the mesh parts that form each parallel subdomain form a connected volume.

5.3.3 Runtime partitioning with MPI-3.0 shared memory regions

In this section we outline a possible alternative implementation strategy for mesh partitioning in LifeV, based on some recent additions to the MPI standard.

Originally, MPI was designed around the concept that multiple processes should not share any memory space and should only communicate by sending and receiving messages. This fundamental concept of the programming model leads to software implementations that

```
Begin simulation: Use $N$ MPI processes, on $M$ supercomputer nodes.
On each node there are $N/M = n$ MPI processes, which can use the same shared memory region.
Define: $M$ process groups, one per node.
Define: One shared memory region usable by the members of each group.
Load: The first process of each group loads the global mesh of the simulation from disk into the shared memory region of his group.
Partition: Mesh partitioning then takes place unchanged, $N$ mesh parts are produced.
Cleanup: The global mesh objects can be discarded from memory at this point, along with the shared memory regions that were used to store them.
```

Algorithm 7: The alternative mesh partitioning algorithm, using MPI-3.0 shared memory regions

minimize the communication between processes, which represents a very expensive type of operation in terms of CPU time. Inter-process communication can be avoided through data duplication, which, however, leads to an increased memory consumption.

Current trends in supercomputing hardware design (see Section 2.3) and the current level of maturity of software tools such as OpenMP justify the use of both MPI parallelism and multi-threading for the producing computational software that is more memory efficient than MPI-only implementations. Mixing MPI and multi-threading introduces, unfortunately, additional complexity to the code and requires an increased amount of programmer effort for development, maintenance and tuning.

The most recent version of the MPI standard, MPI-3.0 (see [96]), introduces MPI shared memory regions, which can be used as an alternative for shared memory multi-threading, for some types of algorithms. It is now possible to define memory regions that are accessible, for read and write operations, to all processes within a given group. Processes can use these memory regions to share large resources, such as static lookup tables, removing the need to duplicate them in the private memory space of each process. A discussion of the shared memory features of MPI-3.0 and the results of some initial experiments can be found in [97] and [98].

Although the programming interface for using shared memory regions has been stabilized in the final version of the MPI-3.0 standard, the standard is yet to be implemented in the current versions of the MPI library made available by supercomputer vendors.

Shared memory regions in MPI represent a promising tool which could also be used in the mesh partitioning stage in LifeV, eliminating the memory bottleneck associated with the runtime partitioning approach. The mesh objects in LifeV are effectively sets of lookup tables that could be stored in shared memory regions. The modified implementation of the runtime partitioning process is described in Algorithm 7.

The online/offline approach described earlier in this chapter effectively removes all the redundant memory usage during the online stage, although the proposed alternative approach using MPI-3.0 shared memory would still reduce the memory consumption of the original runtime partitioning strategy by a factor of n (the number of MPI processes per node, for the process configuration defined earlier). An additional benefit would be greater flexibility, due to the removal of the offline stage.

We consider this strategy a valid approach which we intend to explore as soon as stable implementations of the MPI-3.0 standard become available on our target supercomputing platforms.

5.4 High speed parallel mesh loading

This section discusses the implementation of the read and write routines for the mesh parts produced by the offline partition strategy, as well as the optimization needed to ensure the high performance of these routines.

The mesh object can be represented as a set of arrays enumerating the vertex coordinates and the vertices that make up each edge, face and element in the mesh. Additional arrays are used to encode the connectivity of the entities composing the mesh. For large meshes, this represents a considerable volume of data which needs to be written to disk; using a specialized file format is recommended.

The file format that we chose to store the mesh parts is the Hierarchical Data Format version 5 (HDF5) [53] which represents the de-facto standard format for parallel input and output (I/O) of scientific data. The file is in a binary format, data structures are represented as multi-dimensional arrays which are stored hierarchically in folders, similarly to a computer file system. It is intended for use when large amounts of tabular data is written or read in a parallel MPI environment and it is implemented using the parallel I/O subsystem of MPI libraries (MPI-IO).

Our preliminary implementation of the mesh I/O routines for LifeV creates an individual folder for the tables related to each partition. In these folders, a separate table exists for each array: the three vertex coordinates, array of edge vertices, array of face vertices etc. A simplified representation of this structure is shown in Figure 5.7.

While functional, this implementation proved to be extremely inefficient when using a large number of MPI processes. With more than 1000 MPI processes, the time to load the mesh parts clearly dominated the entire run-time of a simulation, making this version of the routines unusable.

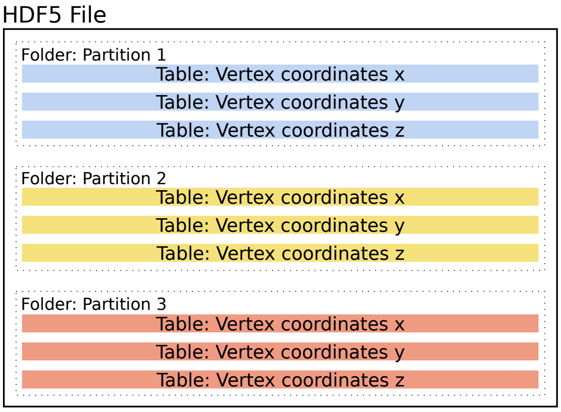

HDF5 File
Folder: Partition 1
Table: Vertex coordinates x
Table: Vertex coordinates y
Table: Vertex coordinates z
Folder: Partition 2
Table: Vertex coordinates x
Table: Vertex coordinates y
Table: Vertex coordinates z
Folder: Partition 3
Table: Vertex coordinates x
Table: Vertex coordinates y
Table: Vertex coordinates z

Figure 5.7: Simplified representation of the initial implementation of HDF5 storage for mesh parts. Each mesh part has its own folder and for each array that is stored a separate table is defined in the HDF5 file.

5.4.1 Optimization with MPI-IO collectives

A profiling analysis of the first implementation revealed that the internal structure chosen for the HDF5 file was causing the performance issues. As each MPI process was requesting to read an individual table from the file, this resulted in a very large number of requests to the HDF5 library. Most importantly, these requests could not be satisfied in terms of MPI-IO collective operations, which are essential to MPI-IO parallel performance.

An MPI-IO collective operation is performed when multiple processes request to simultaneously write or read data to or from the same table at regular positions inside the table. For example, processes request consecutive rows, consecutive columns or adjacent blocks from a table. The performance of this type of operations comes from the fact that the number of effective read and write operations is greatly reduced. A table can be read entirely using one MPI-IO operation, which satisfies the requests of all the processes. Furthermore, the I/O subsystem of a cluster or a supercomputer is able to agregate the I/O operations of groups of MPI processes and efficiently delegate them to a smaller number of processes that perform the actual read and write operations, placing a much lower demand on the storage system. For more information on MPI-IO and collective operations, we refer to [99] and [100].

The new implementation makes exclusive use of MPI-IO collective operations. The structure of the HDF5 file is different in that the data of each mesh part is no longer placed in separate folders. Separate tables aren't created for each mesh part and the data is placed in shared tables in consecutive rows, as is described in Figure 5.8.

The new routines for mesh part I/O perform predictably and efficiently. The fraction of the run-time needed to read the mesh parts, when using upwards of 1000 MPI processes, remains below 10% for simulations of stationary problems. For longer running simulations of time-dependent problems, this fraction becomes negligible.

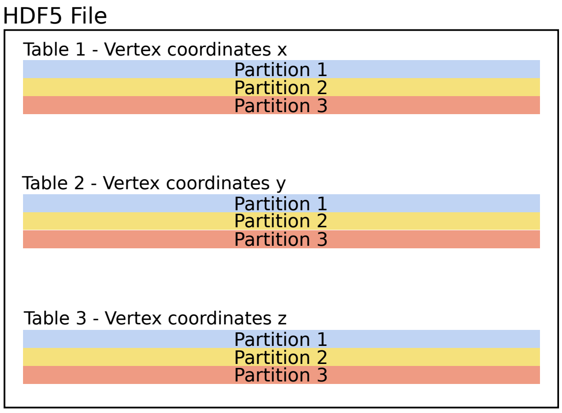


Figure 5.8: Simplified representation of the optimized implementation of HDF5 storage for mesh parts. Processes place data in common tables, which enables the use of MPI-IO collective operations.

6 Multi-threaded finite element assembly

6.1 Introduction

The assembly of the linear system of equations can represent a considerable part of the total simulation time, depending on the size of the problem and the type of FE discretization. Nonlinear simulations require that the stiffness matrix and right hand side vector be updated at each linear step, further increasing the percentage of the total run-time spent inside the assembly loop.

The existing implementation of the finite element assembly in LifeV, based on MPI parallelism, has already been discussed in Chapter 2. It maps properly to the domain partitioning strategy and has good scalability with the number of MPI processes. The current trend in supercomputer design is to increase the number of cores available on each node, although the amount of memory per node is increasing much slower. The reduced amount of memory per core is limiting the effectiveness of this MPI only approach. On this type of machines, it is often impossible to fully subscribe the nodes with MPI processes (i.e. run a number of MPI processes equal to the number of cores), due to the limited amount of memory per core. With a hybrid approach, using MPI at the global distributed memory level and using multiple threads inside each shared memory region, it is possible to better exploit the available resources, while keeping the efficiency of the implementation high.

This chapter describes a multi-threaded implementation in LifeV of the assembly of linear system matrix. The issues related to this sort of design are discussed as well as the impact that different implementation details have on strong. Finally, the implementation is benchmarked on two different supercomputing architectures and the performance of the strategy discussed.

6.2 Requirements of the multi-threaded implementation

Before describing the details of the multi-threaded implementation, it is necessary to explain two concepts related to shared memory programming: thread-safety and reentrant functions.

A function is said to be reentrant if it can be safely executed in parallel multiple times, for instance, if it is called by multiple threads of execution. To ensure this behaviour, a reentrant function must not maintain any state between calls and it must not operate on global or static data, but only on data provided by the caller. While the concept of reentrancy is essential in the context of parallel or asynchronous programming, it is not strictly related to this domain. It is also needed, for instance, in recursive algorithms. It is generally good practice to write reentrant code, whenever possible.

Thread-safety is a property of functions, related to reentrancy, but it only regards the implementation of the functions, not their interface. A thread-safe function can safely be called by multiple threads of execution. It can make use of shared resources, such as memory and open files, but it serializes all use of these resources. It is possible that non-reentrant functions are thread-safe, but making functions reentrant can often make them thread-safe, too [5].

Some constraints need to be placed on the routines composing the FE assembly loop. Figure 6.1 contains the flowchart of the assembly loop in LifeV, in the case of MPI only parallelism.

In Figure 6.1, the **COMPUTE ELEMENTAL MATRIX** routine computes the coefficients of the local elemental matrix, which are associated with the DOFs of one element. In a domain decomposition setting, each MPI process calls this routine on his own set of elements. In our case, we are interested in a second level of parallelism represented by multiple threads. This

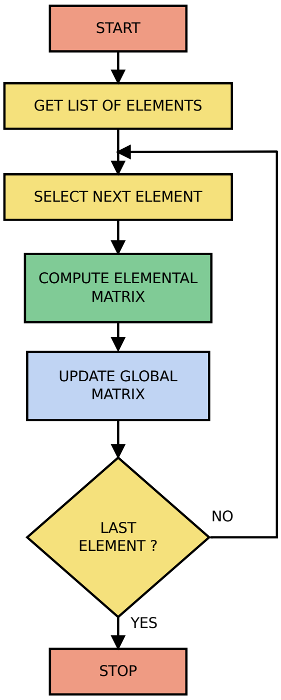


Figure 6.1: FE assembly loop in the MPI single-threaded case. Green denotes routines which, if well coded, are reentrant, while blue denotes routines which are not reentrant but need to be thread-safe.

routine contains no communication between MPI processes, but is called simultaneously, for different finite elements, by multiple threads. Consequently, it needs to be written in a reentrant fashion.

The routine **UPDATE GLOBAL MATRIX** performs all the write operations to the shared global memory. The coefficients of the local elemental matrix are inserted in the proper locations of the global FE matrix. Since it is possible that two elements send contributions to the same coefficient in the global matrix, this routine needs to be thread-safe. The way thread-safety is implemented has little impact on the correctness of the code, save for a small variability due to the non-associativity of floating point operations [101]. However, it can have a considerable impact on the performance and scalability of the implementation.

The thread-safety of this routine can be ensured in multiple ways. At higher level of the program, the entire call to this routine can be serialized; this gives poor performance, as it considerably increases the serial portion of the program. At a lower level, thread-safety can be imposed at the level of actual individual memory operations. The latter approach is described in Section 6.3.

The multi-threaded mode of operation has some additional requirements which parallel sparse matrix class in Trilinos must satisfy. Objects of this matrix class exist in two states:

1. **Open** - A matrix can be built in this state and it does not contain any elements at the start, the sparsity pattern is not defined. Coefficient values and indices are inserted into temporary data structures. At the end of the insertion process, the matrix needs to be closed. The sparsity pattern of the matrix is encoded as a graph and elements are moved into the permanent data structures.
2. **Closed** - If a matrix is build directly in this state, the graph representing the sparsity pattern of the matrix needs to be computed beforehand. In the closed state, existing coefficients can be updated, but no new coefficients can be added to the matrix, as the sparsity pattern cannot be modified. This state is optimized for linear algebra operations (matrix-vector product, matrix-matrix product). It is worth noting that due to the static nature of the data structures used in this state, the update operations are faster than the insertion/update operations in the open matrix state.

The internal mechanisms of the Trilinos matrix class differ between these states. Memory reallocation operations, which are used in the open state of the matrix, cannot be performed in a thread-safe manner. In a multi-threaded setting, it is imperative that the matrix is built in the closed, optimized state.

In the context of the LifeV library, at the beginning of this thesis, the typical usage scenario was to create open matrices for the assembly stage, close them and use them in the closed, optimized, state only in the linear solver. We implemented the needed support for precomputing the sparsity graph, which permits building the matrices directly in the closed, optimized state.

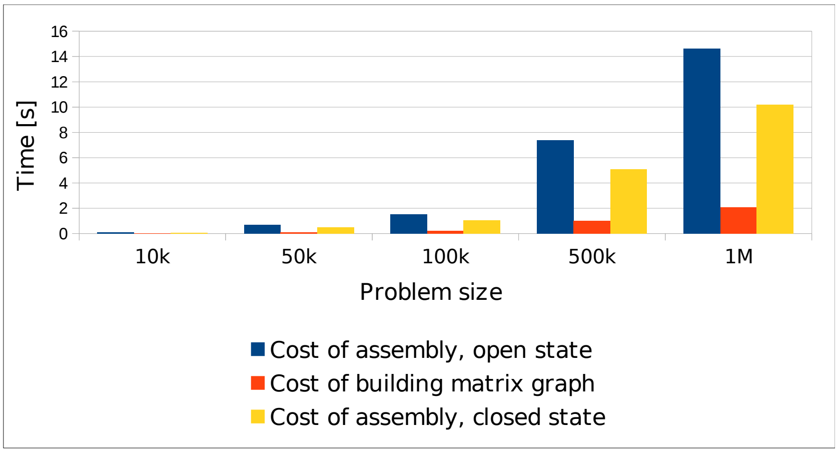


Figure 6.2: Cost of the finite element matrix assembly with or without a precomputed graph, depending on the problem size, for 3D Laplacian problems with P2 finite element.

The graph of the matrix is computed within a separate loop over the finite elements. In this loop, the coefficients of the matrix are not computed, instead, only the indices of the non-zero values of the matrix are computed and stored. The graph object which is computed with this procedure is later used to construct the system matrix. As result, the construction of the matrix object is changed, while the assembly loop is mostly unchanged, with the exception that a different method is called to insert coefficients into the their final position in the matrix, on account of its closed state. Figure 6.2 shows a comparison of the CPU time needed to assembly a matrix without using a precomputed graph, the CPU time to precompute the graph and the time to assemble the matrix with the precomputed graph, for different problem sizes. The test problem is a 3D Laplacian on a regular grid, using P2 finite elements. It can be observed that using a precomputed graph is always beneficial as the combined time to compute the graph and to assemble the matrix using the graph is always lower than the time to assemble the matrix without the graph. In a simulation that involves a recomputation of the matrix coefficients, the gains in terms of CPU time can become substantial.

6.3 Multi-threaded implementation

The multi-threading mechanism for the assembly loop has been implemented using OpenMP, a pragma based approach to multi-threaded programming. Serial (single-threaded) code is annotated with instructions to the compiler (compiler pragmas) regarding which parts of the code are to be executed in parallel, which variables are to be shared between threads and which parts of the code are to be serialized. The compiler can be instructed to ignore the OpenMP pragmas, in which case the resulting code is fully functional, but single-threaded. This approach is not intrusive, but at the same time it is expressive enough that the details of the parallelism are exposed to the programmer.

The existing implementation of the linear system assembly in LifeV, with MPI parallelism,

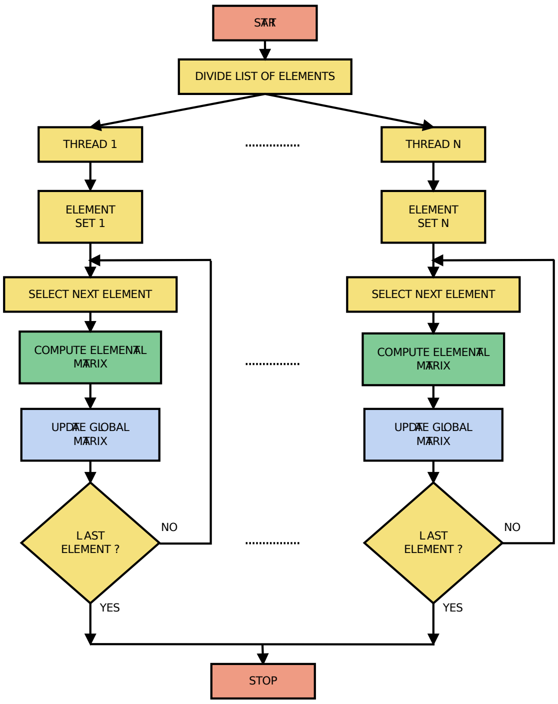


Figure 6.3: FE assembly loop in the multi-threaded case (N threads), showing fork-and-join parallel model.

is described in detail in [24]. Using this framework as a starting point, the multi-threading is implemented in two steps. First, the total amount of work is divided among the threads. Second, it must be ensured that the threads synchronize safely and correctly, if needed.

6.3.1 Work sharing

The parallelisation of the FE assembly loop is performed with a fork-and-join approach [7]. The desired number of threads is spawned before the assembly loop and all the thread-local data that is required is constructed. The list of all the elements stored on the current MPI process is divided among the available threads and each thread will then perform the computations for its own subset of elements, one element at a time. The lifetime of the threads extends until right after the assembly loop, at which point the threads are stopped and the program becomes once again single-threaded. This process is summarized in Figure 6.3.

6.3.2 Synchronization

In parallel programming, communication and synchronization represent bottlenecks that are to be avoided, if possible. OpenMP provides two main types of mechanisms that ensure thread-safety through synchronization. Critical regions are the first mechanism. A block of code can be marked as a critical region, which will force the threads to execute it serially. Any portion of code can be placed in a critical region, but this mechanism comes with a large overhead in terms of CPU time. This makes critical regions feasable only when the code they contain is short or when they are encountered infrequently during the run-time of the program.

Atomic regions represent the other synchronization mechanism. They are much more efficient than critical regions, but their use is limited to serializing operations performed on individual memory locations (updating a single floating point value, incrementing an integer counter etc.). The advantage over critical regions is that the atomic region will result in serialization only if two threads attempt to simultaneously modify the same memory location. This leads to better performance compared to using critical regions.

The IBM BlueGene/Q architecture offers an additional thread-safety mechanism, based on transactional memory [102]. It is used on the same type of memory operations as an atomic region, but differs in implementation. Threads execute the operations in parallel, observing the initial state of the memory location that is to be modified. If this state is changed by the time the thread is ready to update the memory, refered to as a thread collision, the thread will discard the results of the operations it performed and will redo these operations until able to perform the update. This approach can be beneficial as it doesn't involve the serialization of any portion of the code, although in the event of many collisions there is considerable overhead due to the operations that need to be repeatedly performed.

The multi-threaded implementation of the assembly loop requires some form of synchronization when inserting the coefficients of the local elemental matrix into the global matrix. The parallel sparse matrix class in Trilinos contains two code paths: one for updating locally stored rows and one for updating the non-local rows. For local rows, the static data structures of the closed matrix class are used and it is possible to implement thread-safety using only atomic updates. For updating non-local rows, some temporary data structures are used to collect the indices and values that are then communicated to other MPI processes. This is done regardless of whether the matrix is in the open or closed state. The implementation of these temporary containers is not thread-safe, therefore these operations need to be serialized using critical regions.

If the overlapping mesh partition process, described in the Section 5.2.2, is used, in the assembly loop there are only calls to update local rows in the global matrix and there are no more costly critical regions. The benchmarks of the assembly of the sytem matrix, discussed in the following section, are conducted both with and without an overlapping mesh partition.

6.4 Benchmarks

6.4.1 Experiment setup

The multi-threaded implementation is tested on two modern high performance computing architectures:

1. **Cray XE6 - "Monte Rosa"** - this system is located at the Swiss Center for Scientific Computing (CSCS) in Lugano, Switzerland. It is composed of 1496 nodes, each node is equipped with two 8-core AMD Interlagos processors and 32 GB RAM, and high performance networking with the proprietary Gemini 3D torus interconnect. This machine has a non-uniform memory architecture (NUMA): all the RAM installed in a node is visible to all the CPU cores of a node, but each core has a preferential memory region, with operations to this region being faster than to other non-preferential regions. In the particular case of the Cray XE6, the memory on each node is divided in two preferential regions, one for each half of the CPU cores.
2. **BlueGene/Q - "Lemanicus"** - this system is owned and operated by the Center for Advanced Modeling Science (CADMOS) and it is located at Ecole Polytechnique Federale de Lausanne (EPFL). It consists of 1024 nodes, each equipped with a 16-core PowerA2 processor and 16 GB RAM. The PowerA2 processor is able to support up to 4 threads per CPU core (64 threads). The machine has uniform memory nodes (UMA) and is designed with large scale hybrid (MPI and threads) applications in mind. On this machine we examine transactional memory synchronization, in addition to OpenMP atomic regions.

The benchmark consists of measuring the CPU time needed for the assembly of the matrix of linear system of equations using the multi-threaded implementation. We examine two problems types, a 3D Laplacian problem and an advection-diffusion-reaction (ADR) problem, both on regular cubic meshes[1]. Additionally, for each problem type, we use in turn two different finite elements, P1-Bubble and P2. We have examined multiple problem sizes, corresponding to approximately 10000, 50000 and 1000000 DOFs per MPI process. We observed that the strong scalability of the assembly process was identical in all cases. Hence, we will only show the results for the case of 50000 DOFs per MPI process. The BlueGene/Q architecture imposes the use of a minimum of 64 nodes for a job. We run the job with 64 MPI processes, with one MPI process per node, on both machines, which leads to an equal amount of inter-node commmunication on both machines. This amounts to a global problem size of approximately 3.2 million DOFs. The number of threads per MPI process used for the assembly is varied from 1 to 16. Both the overlapping and the non-overlapping approaches are tested, to expose the cost of the critical regions in the non-overlapping case. In the case of the BlueGene/Q, we also examine transactional memory as an alternative to OpenMP atomic regions.

---

[1] In general, we are interested in unstructured meshes, therefore, even if the cubic grid is structured, it is stored as an unstructured one.

In all cases, we also provide as a reference the CPU time to perform the assembly of the system matrix using only MPI parallelism, with an equal number of processors as in the multi-threaded case.

6.4.2 Measurements and discussion

Figures 6.4 and 6.5 show the results of the experiment on the BlueGene/Q and the Cray XE6, respectively. We observe that on the BlueGene/Q machine our implementation achieves better strong scalability than on the Cray XE6, due to the uniform memory architecture of the former. In the case of the NUMA architecture of the Cray, memory operations outside of the preferential memory region are quite costly and affect strong scalability. It is therefore advisable to use a maximum of 4 or 8 threads on this machine, to maintain high efficiency.

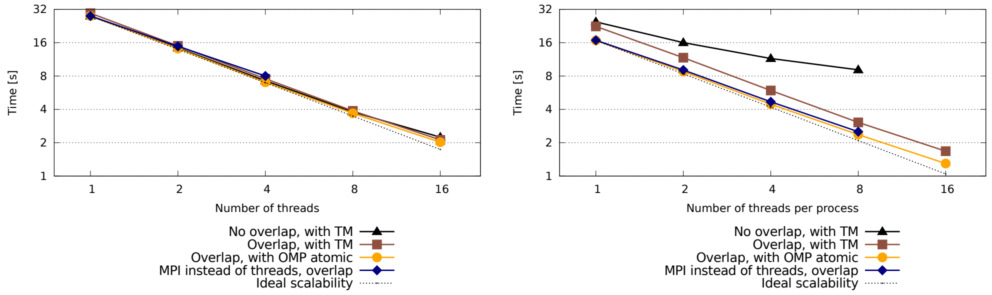
The assembly is less scalable in the case of the ADR problem than for the Laplacian one, due to the computation of the advection term which involves additional memory access operations to the global advection field. This effect is most apparent when using P2 finite elements, when the multi-threaded approach is visibly less scalable than the MPI only one. Since this loss of scalability is visible on both machines, we believe that the cause is an inefficient use of memory in this computation in a multi-threaded setting, which stems from the design of the fundamental data structures, such as the parallel and serial matrices and vectors that are in use, as well as the manner in which they are accessed. There is a strong development effort in the Trilinos project with the goal of implementing a full linear algebra stack which is optimized for multi-threaded applications [103], [104]. At the time of the writing of this thesis, this new development branch has not yet achieved full feature parity with the current one, on which LifeV is developed. Consequently, it is not currently possible to make use of the new Trilinos development branch in LifeV without a substantial loss in capability.

The assembly is slightly less scalable when using P2 elements, rather than P1-Bubble elements. This is to be expected, as in the P2 case, all of the DOFs reside on the faces of the elements. In this case there is a larger chance that two threads attempt to update the same row in the matrix, invoking an OpenMP atomic lock. For P1-Bubble elements, the basis function associated with the DOF in the center of the element has its support limited to the element itself. Consequently, all the coefficients in the matrix row corresponding to this DOF are computed in one step of the loop over the finite elements, without the possibility of a thread collision.

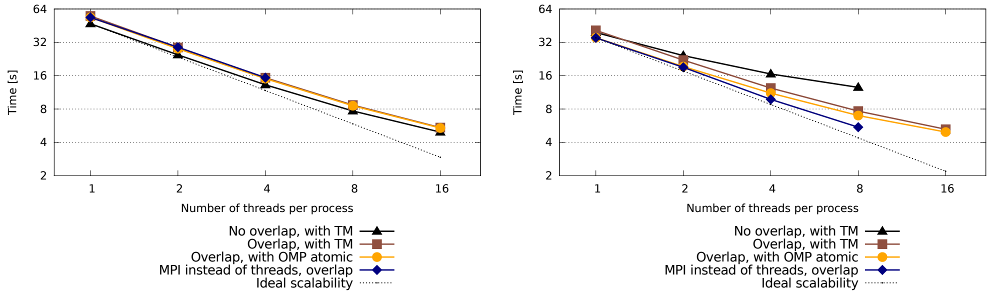
We also observe that the use of overlapping mesh partitioning is only visibly beneficial for P2 finite elements. The number of DOFs residing on the interface between mesh parts is much lower in the case of P1-Bubble elements and the cost of communicating non-local row coefficients is much lower.

The transactional memory synchronization, available on the BlueGene/Q proves to be either equivalent in terms of CPU time, to the OpenMP atomic regions, or slightly slower.

Tables 6.1a and 6.1b contain the best case parallel speedup and efficiency observed during



(a) Laplacian, P1-Bubble (b) Laplacian, P2



(c) ADR, P1-Bubble (d) ADR, P2

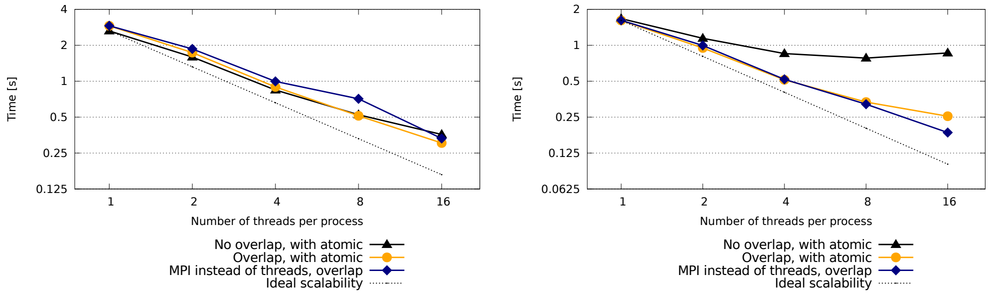
Figure 6.4: Strong scalability of multi-threaded assembly on the IBM BlueGene/Q.

the tests, which corresponds to the overlapping case, using P1-Bubble elements and OpenMP atomic regions for synchronization. Efficiency is computed as the ratio between the measured speedup and the ideal one. The approach is more efficient on the BlueGene/Q, where at 16 threads the efficiency is about 86% for the Laplacian problem and 62% for the ADR problem, compared to the Cray, where it is approximately 60% for the Laplacian problem and 46% in the case of the ADR problem.

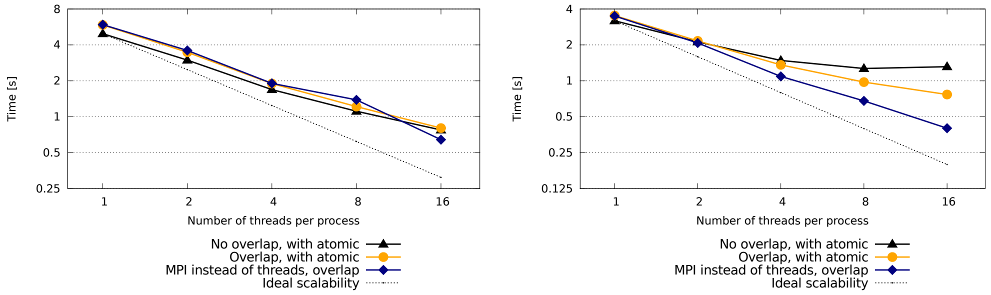
The scalability of the multi-threaded approach is equal, in most cases, to the MPI only approach. When the MPI approach performs better, such as the case of ADR problems and P2 finite elements, the multi-threaded approach can be used in conjuction with MPI, i.e. using a combination of MPI processes and threads which occupies all the processors on a node. In such a case, the full resources available on a node can still be exploited, with only a small loss in efficiency.

6.5 Conclusions

In this chapter we described the work done to implement a multi-threaded implementation of the linear system matrix assembly in LifeV. The implementation was benchmarked on two modern high performance computing machines, the IBM BlueGene/Q and the Cray XE6, which have very different characteristics. The scalability of the code was measured up to 16 threads and the efficiency of the approach was computed. We observed that the multi-



(a) Laplacian, P1-Bubble (b) Laplacian, P2



(c) ADR, P1-Bubble (d) ADR, P2

Figure 6.5: Strong scalability of multi-threaded assembly on the Cray XE6.

Problem size	Number of threads	1	2	4	8	16
Laplacian	Speedup	1.00	1.97	3.97	7.57	13.79
	Efficiency (%)	100	98.52	99.23	94.59	86.19
ADR	Speedup	1.00	1.92	3.58	6.28	9.97
	Efficiency (%)	100	95.97	89.39	78.51	62.30

(a) IBM BlueGene/Q

Problem size	Number of threads	1	2	4	8	16
Laplacian	Speedup	1.00	1.67	3.25	5.67	9.57
	Efficiency (%)	100	83.72	81.21	70.83	59.83
ADR	Speedup	1.00	1.69	3.10	4.83	7.31
	Efficiency (%)	100	84.70	77.45	60.41	45.71

(b) Cray XE6

Table 6.1: Best case speedup and efficiency of multi-threaded assembly. P1-Bubble elements, overlapping partitioning and OpenMP atomic regions are used.

threaded approach is sensitive to the type of finite element used and that it is less efficient on non-uniform memory architectures, as is the case of Cray machine, than on uniform memory ones, like the BlueGene/Q. The scalability of the multi-threaded assembly strategy is equivalent, in most cases, to the existing implementation, based only on MPI parallelism. Multi-threading can be used in this situation without sacrificing efficiency. When the architecture does not allow that all the hardware is exploited using only MPI parallelism, multi-threading can be used together with MPI, resulting in a better usage of hardware resources.

7 Large scale simulations

In this chapter we investigate the performance of the preconditioning methods introduced in Chapters 3 and 4 in the context of the numerical solution of the Navier-Stokes equations. We begin with a short description of the discretization of the Navier-Stokes equations using finite elements, as well as the preconditioners for Navier-Stokes problems which are available in LifeV, see also [77]. In the second part of this chapter, we present the benchmark problems that we use to evaluate our preconditioners. The chapter concludes with a discussion of the numerical results from the benchmarks.

7.1 The Navier-Stokes equations

7.1.1 Weak form

The incompressible Navier-Stokes equations, which describe the motion of a fluid with constant density ρ in a domain $\Omega \subset \mathbb{R}^d$, $d = 2,3$, are written as follows:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} - \operatorname{div}\left[\nu(\nabla\mathbf{u} + \nabla\mathbf{u}^T)\right] + (\mathbf{u}\cdot\nabla)\mathbf{u} + \nabla p = \mathbf{f}, \text{ in } \Omega, \quad \forall t > 0 \\ \operatorname{div}\mathbf{u} = 0, \text{ in } \Omega, \quad \forall t > 0 \end{cases} \tag{7.1}$$

Here, $\mathbf{u}$ represents the fluid velocity, while p represents the pressure divided by the fluid density. ν is the kinematic viscosity, defined as the dynamic viscosity μ divided by the fluid density and $\mathbf{f}$ is a forcing term per unit mass.

The first and second equations impose, respectively, a conservation of momentum and of mass. The first equation is nonlinear, due to the convective term $(\mathbf{u}\cdot\nabla)\mathbf{u}$. In the case of constant kinematic viscosity ν, the diffusion term can be simplified:

$$\operatorname{div}\left[\nu(\nabla\mathbf{u} + \nabla\mathbf{u}^T)\right] = \nu(\Delta\mathbf{u} + \nabla\operatorname{div}\mathbf{u}) = \nu\Delta\mathbf{u}, \tag{7.2}$$

which results in an equivalent form of the equations:

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} - \nu \Delta \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} + \nabla p = \mathbf{f}, \text{ in } \Omega, \quad \forall t > 0 \\ \operatorname{div} \mathbf{u} = 0, \text{ in } \Omega, \quad \forall t > 0 \end{cases} \tag{7.3}$$

System (7.1) or (7.3) are complemented with an initial velocity, $\mathbf{u}(\cdot, 0) = \mathbf{u_0}$ in Ω, and boundary conditions which close the system; for example, of Dirichlet and Neumann type:

$$\begin{cases} \mathbf{u} = \varphi, \text{ on } \Gamma_D, \quad \forall t > 0, \\ \nu \dfrac{\partial \mathbf{u}}{\partial \mathbf{n}} - p\mathbf{n} = \psi, \text{ on } \Gamma_N, \quad \forall t > 0, \end{cases} \tag{7.4}$$

where φ and ψ are given vector functions and Γ_D and Γ_N represent a partition of the domain boundary $\partial\Omega$, such that $\Gamma_D \cup \Gamma_N = \partial\Omega$ and $\mathring{\Gamma}_D \cap \mathring{\Gamma}_N = \emptyset$ and $\mathbf{n}$ is the outward unit normal vector to $\partial\Omega$. For simplicity, we consider the case of homogeneous Dirichlet boundary conditions, i.e. $\mathbf{u} = 0 \text{ on } \Gamma_D, \ \forall t > 0$. The initial condition and boundary conditions yield well posed problems in the two-dimensional case [33]. The same is not necessarily true in the three-dimensional case. The existence of the solution has been proven for the weak formulation of the Navier-Stokes equations [105], although the uniqueness of the solution is still an open problem.

Using similar steps as in Section 2.2, we obtain the weak formulation of the Navier-Stokes equations: $\forall t > 0$, find $\mathbf{u} \in V$, $p \in Q$, such that:

$$\begin{cases} \displaystyle\int_\Omega \frac{\partial \mathbf{u}}{\partial t} \cdot \mathbf{v} \, d\Omega + \nu \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} \, d\Omega + \int_\Omega [(\mathbf{u} \cdot \nabla)\mathbf{u}] \cdot \mathbf{v} \, d\Omega - \int_\Omega p \operatorname{div} \mathbf{v} \, d\Omega \\ \displaystyle\qquad = \int_\Omega \mathbf{f} \cdot \mathbf{v} \, d\Omega + \int_{\Gamma_N} \psi \cdot \mathbf{v} \, d\Gamma \quad \forall \mathbf{v} \in V, \\ \displaystyle\int_\Omega q \operatorname{div} \mathbf{u} \, d\Omega = 0 \quad \forall q \in Q. \end{cases} \tag{7.5}$$

The space V is chosen such that the test functions vanish on the portion of the boundary where Dirichlet boundary conditions are applied:

$$V = [H^1_{\Gamma_D}(\Omega)]^d = \{\mathbf{v} \in [H^1(\Omega)]^d : \mathbf{v}|_{\Gamma_D} = 0\}. \tag{7.6}$$

If $\Gamma_N \neq \emptyset$, the space of the pressure test functions is chosen as:

$$Q = L^2(\Omega). \tag{7.7}$$

If $\Gamma_N = \emptyset$, the pressure p appears in the equations only through its gradient and can be determined only up to a constant. To avoid this, the pressure could be imposed in one point of the domain, which is inconsistent with the choice of Q, or it can be required that the pressure has a null average in the domain. In the latter case, the space Q becomes:

$$Q = L_0^2(\Omega) = \{p \in L^2(\Omega) : \int_\Omega p \, \mathrm{d}\Omega = 0\}. \tag{7.8}$$

The Reynolds number Re is a measure of the extent at which convection dominates the diffusion. It is defined as:

$$Re = \frac{UL}{\nu}, \tag{7.9}$$

where L is a representative length of the problem domain and U is a representative velocity. At low values of the Reynolds number ($Re \ll 1$), the *generalized Stokes problem* is an acceptable simplification of (7.3):

$$\begin{cases} \dfrac{\partial \mathbf{u}}{\partial t} - \nu\Delta\mathbf{u} + \nabla p = \mathbf{f}, \text{ in } \Omega, \quad \forall t > 0 \\ \operatorname{div}\mathbf{u} = 0, \text{ in } \Omega, \quad \forall t > 0 \\ \mathbf{u} = \mathbf{0}, \text{ on } \partial\Omega, \quad \forall t > 0. \end{cases} \tag{7.10}$$

In case this linear evolution problem is advanced in time by an implicit finite difference method, we obtain (now $\mathbf{u}$ and p denote the value of $\mathbf{u}^{n+1}$ and p^{n+1} at the new time level)

$$\begin{cases} \alpha\mathbf{u} - \nu\Delta\mathbf{u} + \nabla p = \mathbf{f}, \text{ in } \Omega, \quad \alpha \geq 0 \\ \operatorname{div}\mathbf{u} = 0, \text{ in } \Omega, \\ \mathbf{u} = \mathbf{0}, \text{ on } \partial\Omega, \end{cases} \tag{7.11}$$

where $\mathbf{f}$ is a new right hand side (still denoted with the same symbol) and α is a coefficient proportional to the inverse of the time step δt.

The weak formulation of (7.11) is written: find $(\mathbf{u}, p) \in V \times Q$ such that:

$$\begin{cases} a(\mathbf{u},\mathbf{v}) + b(\mathbf{v}, p) = (\mathbf{f},\mathbf{v}) \quad \forall \mathbf{v} \in V, \\ b(\mathbf{u}, q) = 0 \quad \forall q \in Q, \end{cases} \tag{7.12}$$

where V and Q were defined earlier and the following notations hold:

$$\begin{aligned} &a : V \times V \to \mathbb{R}, \quad a(\mathbf{u},\mathbf{v}) = \int_\Omega (\alpha\mathbf{u}\cdot\mathbf{v} + \nu\nabla\mathbf{u}\cdot\nabla\mathbf{v}) \, \mathrm{d}\Omega, \\ &b : V \times Q \to \mathbb{R}, \quad b(\mathbf{u}, q) = -\int_\Omega q \operatorname{div}\mathbf{u} \, \mathrm{d}\Omega, \\ &\qquad (\mathbf{f},\mathbf{v}) = \sum_{i=1}^{d} \int_\Omega f_i v_i \, \mathrm{d}\Omega. \end{aligned} \tag{7.13}$$

7.1.2 Finite element discretization

The Galerkin approximation of (7.12) is written as follows: find $(\mathbf{u_h}, p_h) \in V_h \times Q_h$ such that:

$$\begin{cases} a(\mathbf{u_h}, \mathbf{v_h}) + b(\mathbf{v_h}, p_h) = (\mathbf{f}, \mathbf{v_h}) \quad \forall \mathbf{v_h} \in V_h, \\ \qquad\qquad b(\mathbf{u_h}, q_h) = 0 \quad \forall q_h \in Q_h, \end{cases} \tag{7.14}$$

where $V_h \subset V$ and $Q_h \subset Q$ are finite dimensional subspaces depending on the real parameter h (the grid size).

The existence and uniqueness of the solution, cf. [106], is guaranteed if the bilinear form $a(\cdot,\cdot)$ is coercive and continuous, the bilinear form $b(\cdot,\cdot)$ is continuous and if there exists a positive constant β such that:

$$\forall q_h \in Q_h, \quad \exists \mathbf{v_h} \in V_h : b(\mathbf{v_h}, q_h) \geq \beta ||\mathbf{v_h}||_{H^1(\Omega)} ||q_h||_{L^2(\Omega)}. \tag{7.15}$$

(7.15) is also called the *inf-sup condition*, since it is equivalent to the following: a positive constant β exists, such that:

$$\inf_{q_h \in Q_h, q_h \neq 0} \sup_{\mathbf{v_h} \in V_h, \mathbf{v_h} \neq \mathbf{0}} \frac{b(\mathbf{v_h}, q_h)}{||\mathbf{v_h}||_{H^1(\Omega)} ||q_h||_{L^2(\Omega)}} \geq \beta. \tag{7.16}$$

The choice of finite element spaces is therefore guided by the *inf-sup condition.* The space of the velocity solution V_h needs to be larger than the space of the pressure solution Q_n for the solution to be unique. Two finite element spaces that satisfy the *inf-sup condition* are said to be compatible. Examples of common choices of compatible finite element spaces are: $V_h = \left\{\mathbf{v_h} \in [C^0(\bar{\Omega})]^d, \mathbf{v}_h|_K \in [\mathbb{P}_2(K)]^d \ \forall K \in \mathcal{T}_h, \mathbf{v}_h|_{\Gamma_D} = \mathbf{0}\right\}$, $Q_h = \left\{p_h \in C^0(\bar{\Omega}), p_h|_K \in \mathbb{P}_1(K) \ \forall K \in \mathcal{T}_h\right\}$ (in brief $\mathbb{P}_2 - \mathbb{P}_1$); another option is provided by ($\mathbb{P}_1$ – bubble , $\mathbb{P}_1$) in which case $\mathbf{v}_h|_K$ is a linear polynomial plus a cubic one that vanishes on ∂K (a “bubble”), a further one is ($\mathbb{P}_2$ - $\mathbb{P}_0$). Note that in the third case the pressure function is discontinuous across interelement boundaries. In the case of incompatible spaces, the solution is unstable and a stabilization method needs to be employed. For the purpose of the numerical benchmarks described later in this chapter, only compatible finite element spaces have been used. For a discussion on stabilization techniques for Stokes equations, we refer e.g. to [107]. Additionally, an in depth analysis of the numerical approximation of Navier-Stokes equations is available in literature [33] [108].

7.1.3 The solution of the nonlinear system of equations

Using the notations and choice of finite element spaces from the previous section, the Galerkin approximation of (7.3) reads: for every $t > 0$, find $(\mathbf{u_h}(t), p_h(t)) \in V_h \times Q_h$ such that:

$$\begin{cases} \left(\dfrac{\partial \mathbf{u_h}(t)}{\partial t}, \mathbf{v_h}\right) + a(\mathbf{u_h}(t), \mathbf{v_h}) + c(\mathbf{u_h}(t), \mathbf{u_h}(t), \mathbf{v_h}) + b(\mathbf{v_h}, p_h(t)) \\ \qquad\qquad = (\mathbf{f_h}(t), \mathbf{v_h} \quad \forall \mathbf{v_h} \in V_h, \\ b(\mathbf{u_h}(t), q_h) = 0 \quad \forall q_h \in Q_h. \end{cases} \tag{7.17}$$

The convective term of the original equations is represented by the trilinear form:

$$c(\mathbf{w}, \mathbf{z}, \mathbf{v}) = \int_\Omega [(\mathbf{w} \cdot \nabla)\mathbf{z}] \cdot \mathbf{v} \, d\Omega \quad \forall \mathbf{w}, \mathbf{z}, \mathbf{v} \in V. \tag{7.18}$$

The nonlinear system of equations that corresponds to (7.17) can be written in compact form as follows: for all $t > 0$,

$$\begin{cases} M\dfrac{d\mathbf{u}(t)}{dt} + A\mathbf{u}(t) + C(\mathbf{u}(t))\mathbf{u}(t) + B^T\mathbf{p}(t) = \mathbf{f}, \\ \qquad\qquad\qquad\qquad B\mathbf{u}(t) = \mathbf{0}, \end{cases} \tag{7.19}$$

with initial condition $\mathbf{u}(0) = \mathbf{u_0}$. There are many ways to discretize the problem in time. For example, the θ-method can be used to perform the time discretization of the system. By setting:

$$\begin{aligned} \mathbf{u}_\theta^{n+1} &= \theta\mathbf{u}^{n+1} + (1-\theta)\mathbf{u}^n, \\ \mathbf{p}_\theta^{n+1} &= \theta\mathbf{p}^{n+1} + (1-\theta)\mathbf{u}^n, \\ \mathbf{f}_\theta^{n+1} &= \theta\mathbf{f}(\theta t^{n+1} + (1-\theta)t^n), \end{aligned}$$

we obtain the following system of algebraic equations:

$$\begin{cases} M\dfrac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + A\mathbf{u}_\theta^{n+1} + C(\mathbf{u}^*)\mathbf{u}_\theta^{n+1} + B^T\mathbf{p}_\theta^{n+1} = \mathbf{f}_\theta^{n+1}, \\ \qquad\qquad\qquad\qquad B\mathbf{u}_\theta^{n+1} = \mathbf{0}. \end{cases} \tag{7.20}$$

The choice of $\mathbf{u}^*$ may lead to a nonlinear problem, e.g. if $\mathbf{u}^* = \mathbf{u}_\theta^{n+1}$, or to a linear one, if $\mathbf{u}^*$ is an extrapolation of $\mathbf{u}_\theta^{n+1}$ based on $\mathbf{u}^n$ and $\mathbf{u}^{n-1}$. Here, M represents the mass matrix, with entries:

$$m_{ij} = \int_\Omega \varphi_i \varphi_j d\Omega.$$

Based on the choice of parameter θ, there are multiple alternatives to the solution of this system.

If $\theta = 0$, we obtain the forward Euler method, which in this case leads to an overdetermined system for the velocity unknown:

$$\begin{cases} M\mathbf{u}^{n+1} = H(\mathbf{u}^n, \mathbf{p}^n, \mathbf{f}^n) \\ B\mathbf{u}^{n+1} = \mathbf{0}. \end{cases}$$

Replacing $\mathbf{p}^n$ by $\mathbf{p}^{n+1}$ leads to a *semi-explicit* discretization:

$$\begin{cases} \dfrac{1}{\Delta t} M\mathbf{u}^{n+1} + B^T\mathbf{p}^{n+1} = \mathbf{G}, \\ B\mathbf{u}^{n+1} = \mathbf{0}, \end{cases} \tag{7.21}$$

where $\mathbf{G}$ is a suitable known vector. This involves solving for pressure a reduced system $BM^{-1}B^T\mathbf{p}^{n+1} = BM^{-1}\mathbf{G}$ and recovering the velocity $\mathbf{u}^{n+1}$ from the first equation. Given a choice of compatible finite element space V_h and Q_h, the reduced system is non-singular. The time discretization is stable under the condition [33]

$$\Delta t \le C \min\left(\frac{h^2}{\nu}, \frac{h}{\max_{\mathbf{x}\in\Omega}} |\mathbf{u}^n(\mathbf{x})|\right).$$

In the case of an *implicit* discretization such as the backward Euler method ($\theta = 1$ and $\mathbf{u}^* = \mathbf{u}_\theta^{n+1}$), which is unconditionally stable, we obtain the nonlinear algebraic system of equations:

$$\begin{cases} M\dfrac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + A\mathbf{u}^{n+1} + C(\mathbf{u}^{n+1})\mathbf{u}^{n+1} + B^T\mathbf{p}^{n+1} = \mathbf{f}^{n+1}, \\ B\mathbf{u}^{n+1} = \mathbf{0}. \end{cases} \tag{7.22}$$

Solving the problem in the *implicit* case requires three nested loops:

1. the temporal iterations
2. iterations of the Newton method (or another linearization method)
3. iterations of a preconditioned Krylov method (typically GMRES or BiCGStab)

If the convective term $\mathbf{u}^*$ is treated explicitly, this results in the *semi-implicit* scheme, which avoids the use of Newton iterations for the solution of the problem. The algebraic system of linear equations that is formed reads:

$$\begin{cases} \dfrac{1}{\Delta t} M\mathbf{u}^{n+1} + A\mathbf{u}^{n+1} + C(\mathbf{u}^n)\mathbf{u}^{n+1} + B^T\mathbf{p}^{n+1} = \mathbf{G}, \\ B\mathbf{u}^{n+1} = \mathbf{0}, \end{cases} \tag{7.23}$$

with $\mathbf{G}$ a suitable vector. This method comes with the following restriction on the time step

[33]:

$$\Delta t \leq C \frac{h}{\max_{\mathbf{x}\in\Omega} |\mathbf{u}^n(\mathbf{x})|}. \tag{7.24}$$

7.1.4 Preconditioners for Navier-Stokes equations

An efficient class of preconditioners for Navier-Stokes equations is derived from block factorizations of the system matrix in:

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \mathbf{U} \\ \mathbf{P} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix}, \tag{7.25}$$

where $F = \frac{1}{\Delta t} M + A + C(\mathbf{u}^n)$ for the case of (7.23). The matrix can be written as a block LU factorization:

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & B^T \\ 0 & -S \end{bmatrix}, \tag{7.26}$$

where $S = BF^{-1}B^T$ is the pressure Schur complement. Computing these factors is too expensive, due to the presence of S, which would require the computation of the inverse F^{-1}. To alleviate this computational cost, the approximations $\hat{F}$ and $\hat{S}$ can be used. To obtain the Pressure Correction Diffusion (PCD) preconditioner [109], S is replaced by

$$\hat{S}_{PCD} = A_p F_p^{-1} M_p, \tag{7.27}$$

where M_p is the pressure mass matrix, A_p the pressure Laplacian matrix and F_p is the convection-diffusion pressure matrix, where the advection term corresponds to the discretization of $\mathbf{u}^n \nabla p$; these matrices are complemented by homogeneous Dirichlet or Neumann conditions, see [110] for more details. F^{-1} can be approximated by a suitable preconditioner for advection-diffusion problems. Applying this preconditioner involves one pressure Poisson solve, a mass matrix solve and a matrix-vector product with F_p.

Starting from a different LU factorization of the matrix in (7.25)

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} F & 0 \\ B & -S \end{bmatrix} \begin{bmatrix} I & F^{-1}B^T \\ 0 & I \end{bmatrix}, \tag{7.28}$$

and approximating S by $\hat{S} = BD^{-1}B^T$ we obtain the *SIMPLE* preconditioner, first introduced in [111]. D is a triangular matrix which is easy to invert, such as the diagonal of the F block. F is also approximated by D in the right factor. The SIMPLE preconditioner reads:

$$P_{SIMPLE} = \begin{bmatrix} F & 0 \\ B & -\hat{S} \end{bmatrix} \begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix} \tag{7.29}$$

The application of the simple preconditioner involves an F-solve for the velocity and a problem similar to the Laplacian for the pressure.

The *SIMPLE* preconditioner can be regarded as a particular case of a general family of inexact factorizations:

$$\begin{bmatrix} F & 0 \\ B & -B\mathscr{L}B^T \end{bmatrix} \begin{bmatrix} I & \mathscr{U}B^T \\ 0 & I \end{bmatrix}. \tag{7.30}$$

The two matrices $\mathscr{L}$ and $\mathscr{U}$ both represent approximations of F^{-1}. Using this factorization, the solution of the linear system involves the following steps:

1. Solve: $F\mathbf{u}^* = \mathbf{b}$ for the intermediate velocity
2. Solve: $-B\mathscr{L}B^T\hat{\mathbf{p}} = -B\mathbf{u}^*$ for pressure
3. Compute final velocity: $\hat{\mathbf{u}} = \mathbf{u}^* - \mathscr{U}B^T\hat{\mathbf{p}}$.

Two possibilities for the choice of $\mathscr{L}$ and $\mathscr{U}$ have been investigated in [112]:

$$\mathscr{L} = \mathscr{U} = \left(\frac{1}{\Delta t} M_l\right)^{-1}, \tag{7.31}$$

where M_l is the lumped mass matrix, which is called the Chorin algebraic approximation, and:

$$\begin{aligned} \mathscr{L} &= \left(\frac{1}{\Delta t} M_l\right)^{-1}, \\ \mathscr{U} &= F^{-1}, \end{aligned} \tag{7.32}$$

which is refered to as the Yosida approximation. Choosing $\mathscr{U}$ as an approximation of the inverse F^{-1}, we obtain the approximate Yosida preconditioner.

The preconditioners listed here can be all used within the LifeV library. The analysis of the preconditioners and a discussion of their implementation is found in [77].

7.2 Numerical benchmarks

7.2.1 Description of test problems

The numerical benchmarks we propose in this chapter involve the solution of the Navier-Stokes equations using the finite element approximation, on a physiological problem geometry, that of an arterial aneurysm. The geometry was obtained through medical measurements of a real patient pathology [77]. Figure 7.1 shows the problem domain geometry that was used.

Based on this geometry, computational meshes of two different sizes were produced: one with approximately 1 million tetrahedra, the other with approximately 8.7 million tetrahedra.

For the purpose of the numerical tests, the velocity and pressure are discretized using $\mathbb{P}_2$ and $\mathbb{P}_1$ finite elements, respectively, which results in approximately 4.8 million DOFs for the small problem and 37 million DOFs for the large problem. The following characteristic measures correspond to this benchmark: characteristic length $L_{char} = 0.35cm$, characteristic velocity $U_{char} = 22cm/s$, density $\rho = 0.001g/cm^3$ and kinematic viscosity $\nu = 0.035cm^2/s$. Using these values, the Reynolds number computed $Re = \frac{L_{char}U_{char}}{\nu} = 220$. For the time discretization, a semi-implicit backward Euler scheme is used, with a time step $\Delta t = 10^{-3}s$. A suitable initial solution is obtained by solving a Stokes problem, with the same discretization parameters, on the same problem domain.

The linear system of equations is solved at each time step with GMRES without restart, with a tolerance of 10^{-7}, using the PCD preconditioner described in the previous section. We approximate the inverse of the velocity convection-diffusion block F^{-1} with an application of the 2-level AAS preconditioner introduced in Chapter 3. In the interest of brevity, for the remainder of this chapter we will simply use AAS to refer to the 2-level AAS preconditioner with minimal overlap, where the coarse level problem is solved inexactly with 5 Gauss-Seidel iterations. We perform strong scalability measurements, comparing multiple configurations of this preconditioner:

1. As a reference, we setup the AAS preconditioner with serial subdomain problems ($N_S = 1$), solved exactly with a serial LU factorization.
2. Alternatively, AAS is configured to use a given number of parallel subdomain problems, and the number of processes assigned per subdomain is computed as $N_S = N_P/N_{DD}$. On each subdomain, the ShyLU preconditioner is used with the best-case pre-existing configuration, as obtained in Chapter 3: exact LU factorization on diagonal blocks, sparse approximation of the Schur complement using the probing method and 5 subiterations

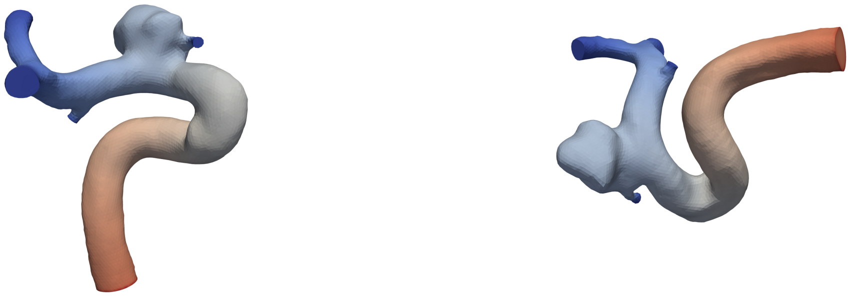


Figure 7.1: The geometry of the aneurysm problem, viewed from two angles.

of GMRES on the Schur complement system.

3. The ShyLU preconditioner is used again with the difference that the Schur complement system is solved inexactly using the IQR method, introduced in Chapter 4. We use a value of 5% for the size of the Krylov subspace, relative to the size of the Schur complement matrix. To ensure convergence in the case of the current benchmarks, the scaling parameter λ was set to the average of the diagonal elements of the upper triangular matrix R. Additionally, the computation of the IQR preconditioner is performed using a vector of ones as right hand side. With the exception of IQR, the configuration of ShyLU is identical to the previous case.

4. An additional case is considered on the Cray machine, where the multi-threaded direct solver PARDISO is available. In this case, we keep the number of MPI processes constant at $N_P = 128$ for the small problem and $N_P = 1024$ for the larger one and we increase the total processor usage by using multiple threads per MPI process. The number of AAS subdomains is constant in this case, equal to the number of MPI processes.

In cases where parallel subdomain problems are used, the number of subdomains is imposed at $N_{DD} = 128$ in the case of the smaller problem and $N_{DD} = 1024$ for the larger one.

The tests are run on two supercomputers: the Cray XE6 “Monte Rosa” at CSCS and the IBM BlueGene/Q “Lemanicus” of CADMOS. A variable number of MPI processes is used: from 128 to 1024 processes in the case of the smaller problem and from 1024 to 8192 in the case of the larger one.

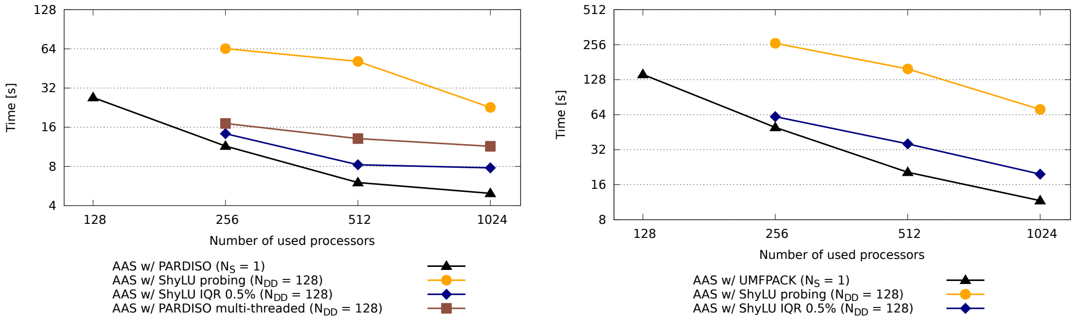
In all test cases, the serial LU factorization is performed using PARDISO on the Cray and UMFPACK on the BlueGene/Q. The use of different LU solvers is due to the fact that PARDISO, which is not open source software, is not available on the BlueGene/Q.

For each case, we measure the time to compute the global preconditioner, the number of outer GMRES iterations that are performed and the total time to outer GMRES convergence. As in the previous numerical tests, the time per outer GMRES iteration is computed using the acquired measurements, as well as a total time to solution, which is the sum of the time to compute the preconditioner and the time to GMRES convergence.

7.2.2 Results and discussion

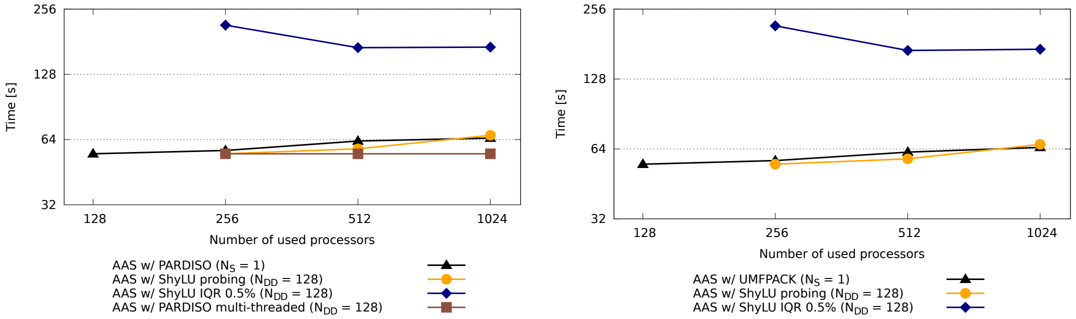
Smaller problem - 4.8 million DOFs

From the point of view of the time to compute the preconditioner (see Figure 7.2) we see that the IQR configuration of ShyLU is much faster than the one using the probing method to approximate the Schur complement matrix. We would like to state, again, the observation from Chapter 4 that the time to compute the preconditioner in the reference case ($N_S = 1$, exact LU) represents a lower bound for the time to compute the ShyLU preconditioner. ShyLU,



(a) Cray XE6 (b) BlueGene/Q

Figure 7.2: Time to compute the preconditioner - 4.8 million DOFs



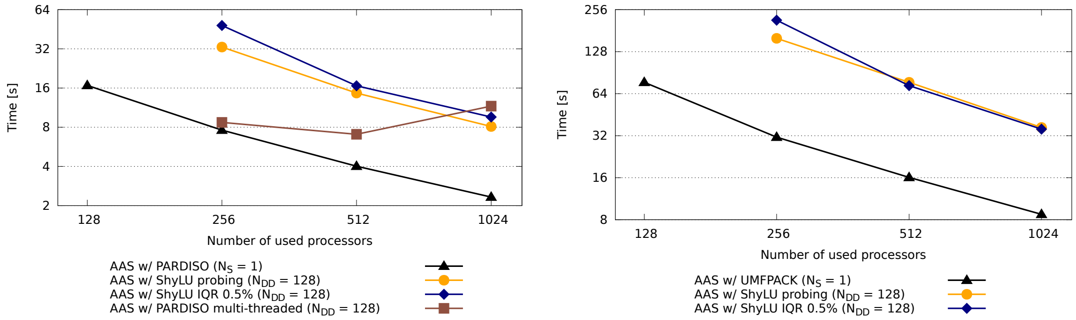
(a) Cray XE6 (b) BlueGene/Q

Figure 7.3: Number of GMRES iterations - 4.8 million DOFs

in all configurations, performs additional work in the preconditioner computation phase, with respect to the reference case. Computing the LU factorization of $N_{DD} \times N_S = N_P$ diagonal blocks is equivalent, in terms of the amount of work, to the the factorization of $N_{DD} = N_P$ serial subdomain matrices (in the reference case). However, ShyLU must perform additional computations related to the Schur complement system. The same observations apply to the time per outer GMRES iteration. The multi-threaded PARDISO is slower to compute the LU factorization than both the reference case and ShyLU with IQR preconditioner.

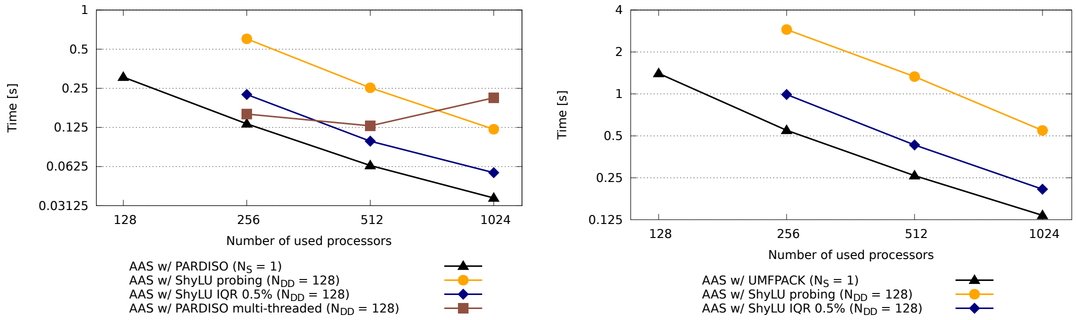
The IQR preconditioner, when used inside ShyLU, leads to a much larger number of outer GMRES iterations than both the other configuration of AAS with ShyLU and the reference configuration for AAS (Figure 7.3). As is expected due to the constant number of AAS subdomains and the fact that multi-threaded PARDISO performs an exact LU factorization of the subdomain matrices, the number of outer iterations is kept constant in this case.

However, the IQR configuration is twice as fast, in terms of time per GMRES iteration, as the configuration of ShyLU with probing the approximation (see Figure 7.5. However, due to the much increased number of outer GMRES iteration needed to converge, the actual time until GMRES convergence is either comparable or slightly slower than ShyLU with probing, as is shown in Figure 7.4. We observe a loss of scalability in the case of multi-threaded PARDISO, in



(a) Cray XE6 (b) BlueGene/Q

Figure 7.4: Time to GMRES convergence - 4.8 million DOFs



(a) Cray XE6 (b) BlueGene/Q

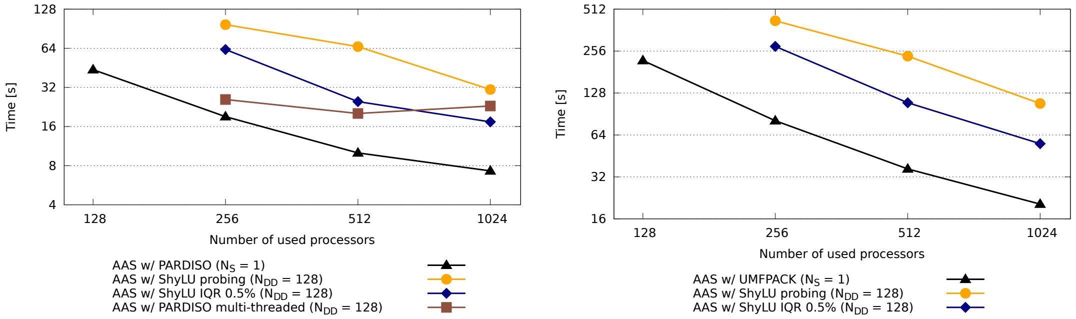
Figure 7.5: Time per GMRES iteration - 4.8 million DOFs

terms of time per outer GMRES iteration which also leads to an increasing time to GMRES convergence. The triangular solves performed by PARDISO do not represent a highly scalable algorithm from the point of view of strong scalability and the performance is further hindered by the NUMA architecture of the Cray node.

Overall, IQR represents a considerable improvement over the pre-existing configuration of ShyLU. In terms of global time to solution (i.e. time to compute the preconditioner and to solve the linear system) we see that IQR brings a two-fold decrease in CPU time. Even with the addition of IQR, using parallel subdomain problems for AAS and ShyLU remains more than twice as slow as the reference configuration involving serial subdomain problems. The multi-threaded approach is overall equivalent to the reference case at 256 MPI processes, but due to the poorly scalable GMRES iterations it loses the advantage over ShyLU at 1024 processes.

Larger problem - 37 million DOFs

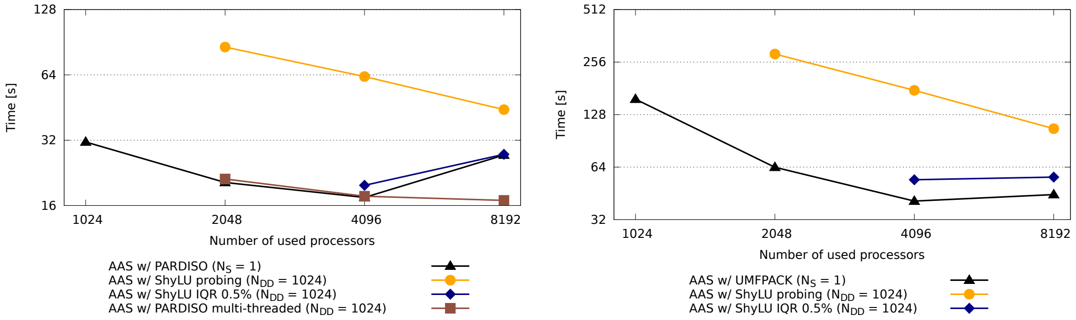
In the case of the larger problem, with $N_P = 2048$, $N_{DD} = 1024$, $N_S = 2$ the IQR configuration did not converge globally. We also observe a loss of scalability in the computation of the pre-



(a) Cray XE6

(b) BlueGene/Q

Figure 7.6: Total time to solution - 4.8 million DOFs



(a) Cray XE6

(b) BlueGene/Q

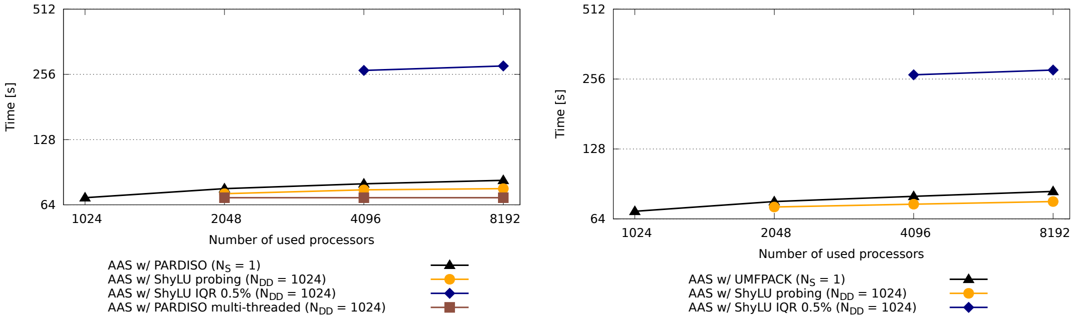
Figure 7.7: Time to compute the preconditioner - 37 million DOFs

conditioner in the reference case, when moving from 4096 to 8192 processes, which indicated that the problem size is not large enough for this number of MPI processes.

Otherwise, we see the same relative performance between the three preconditioner strategies. The time to compute the IQR configuration of ShyLU is a massive improvement over the probing configuration, approaching the reference case in terms of CPU time (Figure 7.7).

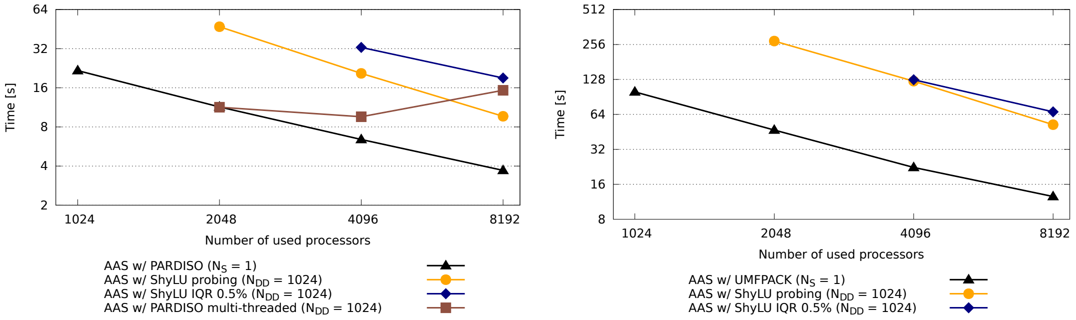
For the larger problem, the AAS preconditioner with multi-threaded PARDISO subdomain solver exhibits the same strong scalability as in the case of the smaller problem. The time to compute the preconditioner is however much closer to the reference case and lower than for either configuration of ShyLU. We see the same loss of strong scalability in the GMRES iterations, but overall the multi-threaded case is equivalent to the case with serial subdomain problems and is faster than when using the ShyLU preconditioner with parallel subdomain problems.

It leads, however, to a much large number of outer GMRES iterations (Figure 7.8) which causes the time to GMRES convergence to be larger than in the case of probing (Figure 7.9), although for IQR the time per GMRES iteration is more comparable to the reference case when $N_S = 1$.



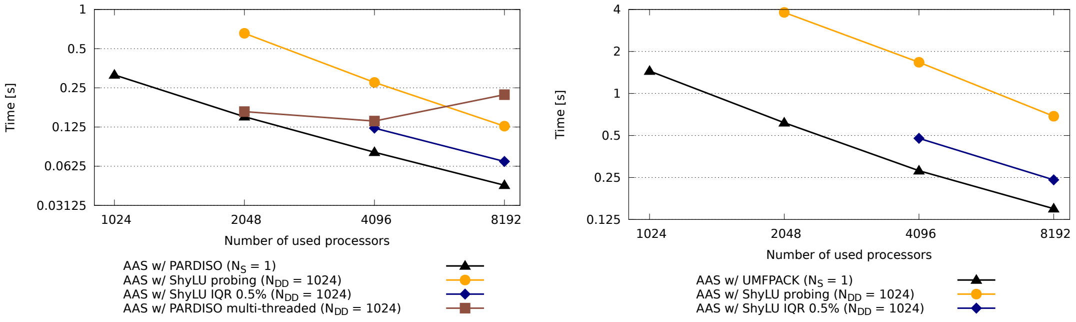
(a) Cray XE6 (b) BlueGene/Q

Figure 7.8: Number of GMRES iterations - 37 million DOFs



(a) Cray XE6 (b) BlueGene/Q

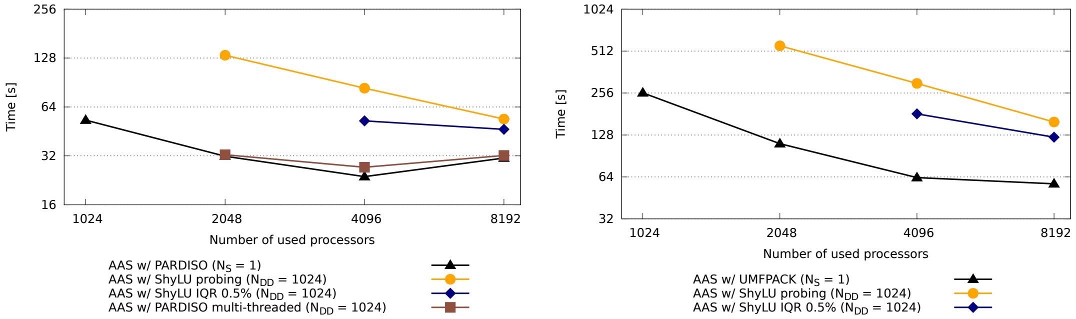
Figure 7.9: Time to GMRES convergence - 37 million DOFs



(a) Cray XE6 (b) BlueGene/Q

Figure 7.10: Time per GMRES iteration - 37 million DOFs

As in the case of the smaller problem, the use of IQR leads to a visible decrease in CPU with respect to the previous configuration of ShyLU (Figure 7.11). The reference configuration remains, in most cases, twice as fast as the configuration using ShyLU.



(a) Cray XE6

(b) BlueGene/Q

Figure 7.11: Total time to solution - 37 million DOFs

7.3 Closing remarks

In this chapter we have continued the investigation on the performance of the ShyLU subdomain preconditioner and of the IQR preconditioner, which is used as a component for the Schur complement algorithm employed by ShyLU (see 5). A set of benchmarks was set up, involving the numerical solution of Navier-Stokes on physiological problem domains. The performance of the reference preconditioner available in LifeV for this type of problems, the PCD preconditioner, was evaluated for different approximation strategies for the inverse of the convection-diffusion block. The reference strategy, involving a 2-level AAS preconditioner with serial subdomain problems solved exactly with an LU factorization, was compared to the 2-level AAS preconditioner with parallel subdomain problems introduced ealier in this thesis.

For solving, inexactly, the parallel subdomain problems, the ShyLU preconditioner was used in two configurations: one involving the approximation of the Schur complement matrix using the probing method, the other using the IQR preconditioner introduced in Chapter 4. The reference configuration of AAS remained faster than the one involving ShyLU, however, the IQR preconditioner that we developed represents a considerable improvement to the performance of ShyLU. Although the IQR is in an early stage of development, the positive results obtained so far motivate further study in this direction, which will bring improvements both algorithmically and to the implementation side.

8 Summary of contributions, recommendations and future work

In this chapter we summarize the contributions made in the course of this thesis to the Trilinos and LifeV software libraries. They have been integrated into the code bases of both projects and are, or will soon be, available for public use. Additionally we issue some recommendations to the solution of various problems, based on the findings presented in the previous chapters.

8.1 Contributions to Trilinos

Two levels of parallelism for AAS

One of the central contributions to the Trilinos library collection is the addition of support for parallel subdomain problems in the Algebraic Additive Schwarz (AAS) preconditioner framework. Previously, the implementation of AAS in the IFPACK package supported only serial subdomain problems, which imposed the limitation that the number of AAS subdomains is strictly tied to the number of MPI processes in use. Our attempt to remove this limitation was justified by two facts. First, the number of iterations needed for an iterative solver to converge, when preconditioned with AAS, in the absence of a coarse problem, will increase with the number of subdomains in the AAS formulation. This concern is secondary, since in our solution process the AAS preconditioner is never used without a coarse problem. Second, it may be possible to achieve better scalability, in terms of CPU time, on modern multi-core nodes if an additional level of parallelism is employed at the subdomain level.

We introduced the needed support in the IFPACK package and the new implementation was proven to be numerically equivalent to the previous one, i.e. with an exact solve of the parallel subdomain problems, keeping the number of subdomains constant and increasing the number of MPI processes used leads to an equal number of linear solver iterations as in the case of serial subdomain problems.

Subdomain solvers for AAS

With the support of parallel subdomain problems in place in AAS, the issue of suitable parallel (using MPI) subdomain solvers has been investigated. Trilinos offers an interface to the most efficient direct solvers, such as UMFPACK, MUMPS and PARDISO. The MUMPS solver uses MPI parallelism and is suitable for solving the parallel subdomain problems although the interface did not allow the use of MUMPS in any other situation than as a linear solver for the global problem. Performing some modification to this interface, we were able to integrate MUMPS with the new AAS preconditioner.

An alternative strategy for solving the parallel subdomain problems was found in the ShyLU preconditioner. ShyLU, based on a Schur complement algorithm, was originally developed in Trilinos as an inexact solver or preconditioner for the very sparse linear problems that are associated with electrical circuit simulations. We sought to use ShyLU as an inexact subdomain solver in the AAS preconditioner for the linear systems associated with the finite element discretization of partial differential equations, which represents a different setting than its original one. We made the needed modifications to its implementation to allow its use in this new context. The combination of AAS and ShyLU showed very good strong scalability, although in terms of absolute CPU time it performed worse than a reference configuration of AAS with serial subdomain problems.

The incomplete QR factorization preconditioner

Our further work was to increase the absolute performance of ShyLU. Performance analysis helped us identify that most of the CPU time used by ShyLU was spent with the computation of the sparse approximation of the Schur complement and the solution of the corresponding linear system.

We investigated an alternative solution to this problem in the form of the incomplete QR factorization (IQR). The IQR is based on performing a QR factorization of the system matrix projected on a Krylov subspace of much smaller size than the problem at hand. The algorithm uses GMRES iterations to obtain the Q and R factors. By reusing the Q and R factors, this algorithm can be used as a preconditioner for solving a series of linear systems which are spectrally equivalent, such as the case of the linear solves occuring during nonlinear iterations.

We implemented the IQR in Trilinos and we used it as an inexact solver for the Schur complement linear system in the ShyLU preconditioner, which brought a considerable increase in the absolute performance of ShyLU. IQR is implemented independently of ShyLU, which potentially allows its use in other contexts. Indeed, we showed that IQR is generally beneficial when used within ShyLU; other applications are the subject of future work.

Thread safety for finite element matrices

Additional changes were needed for the parallel sparse matrix classes of Trilinos. In LifeV, we developed support for the multi-threaded finite element assembly of the linear system matrix. The Epetra package, used by LifeV, was not developed with multi-threading in mind, but rather MPI parallelism only. We performed a series of modifications to the parallel sparse matrix classes that ensured that the constructions of these matrices could safely be performed using multiple threads simultaneously.

8.2 Contributions to LifeV

The new additions to Trilinos listed in the previous section can be used in LifeV; this did not require significant changes to the LifeV code directly. Here we describe other contributions to LifeV.

Mesh partitioning operations

At the time this thesis was started, the mesh partitioning process in LifeV was always performed at runtime and represented a considerable memory bottleneck in the simulation process. We implemented a set of classes which offload the mesh partition process to an earlier, offline, stage. With these new classes it is possible to partition the mesh of the simulation before-hand, on a workstation and save the mesh parts to disk, using the efficient binary storage format HDF5.

Additionally, the new mesh partitioning classes offer support for hierarchical mesh partitioning, which was needed to ensure the connectivity of the mesh parts that make up individual parallel subdomains in the new AAS framework.

Efficient input and output of meshes

The offline partitioning process brought the need to have a fast and efficient way to read, at runtime, the mesh parts that are saved during the offline stage. At high processor counts this process puts considerable pressure on the input/output subsystem of compute clusters or supercomputers and, if done inefficiently, can grow to dominate the runtime of a simulation. We implemented the support to load the mesh parts in a fully parallel manner, using efficient MPI-IO collective operations, with the end result that the loading operation now represents only a small fraction of the total simulation runtime.

Multi-threaded finite element assembly

The last addition to LifeV represents a multi-threaded implementation of the finite element assembly of the linear system matrix. This development was motivated by the need to regain parallel efficiency in this stage of the simulation, in cases when the available memory on supercomputer nodes limits the use of MPI processes, such that the number of processes is lower than the number of available cores. The multi-threaded assembly can currently be used in LifeV independently of problem type.

The new assembly process has the requirement that the sparse matrix objects that are used to store the linear system matrix have to be constructed with a static sparsity graph. Support for precomputing this sparsity graph was implemented in LifeV. The option to use a precomputed graph when constructing the system matrix is also available without the multi-threaded assembly and we saw that it improves the performance of the system matrix assembly in all cases.

8.3 Recommendations

In this final section we identify the best configuration options and parameters values to be used at various steps of the simulation process in LifeV.

8.3.1 Preprocessing and finite element assembly

The offline mesh partitioning strategy, described in Chapter 5, represents an efficient solution for the severe memory bottleneck that is encountered when partitioning the mesh at runtime into a large number of parts. For all but the most trivial simulations, the computational mesh of the domain should be partitioned before-hand, into the desired number of parts, on a workstation with a large amount of memory available. The mesh parts are loaded at simulation runtime using the efficient mesh loading routines based on HDF5 and MPI-IO collective operations.

In order to reduce the amount of MPI communication during the finite element assembly stage, the mesh partitioning should be performed with an overlap. This way, off-processor rows in the linear system do not need to be updated during the system assembly, leading to a decrease in CPU time and increased scalability of the assembly stage.

The linear system matrix should be constructed in the closed state, with the use of a precomputed sparsity graph. While in the case of a stationary simulation the benefit of this approach is minimal, in the case of time dependent and non-linear simulations where multiple matrix updates are performed, using the precomputed graph is essential to good performance.

On supercomputers with large multi-core nodes, with a reduced amount of available memory per CPU core, it may not be possible to fully subscribe the CPU cores using only MPI processes. In such a situation, the multi-threaded finite element assembly should be used in conjuction with MPI to better utilize the available hardware resources.

8.3.2 Linear solver and preconditioner

LifeV offers a selection of specialized preconditioners for the solution of the Navier-Stokes equations, in the form of the approximate block factorization preconditioners PCD, SIMPLE and Yosida. All these preconditioners require, however, an approximation of the inverse of the advection-diffusion block of the system matrix. The 2-level AAS preconditioner is an efficient way to provide such an approximation and its performance and scalability is key to the global performance of the simulation.

We have observed that an inexact solver is suitable for the coarse level of the preconditioner. Using a few iterations of Gauss-Seidel is enough to keep the number of outer GMRES iterations under control, when the number of AAS subdomains increases. Additionally, the use of minimal overlap between subdomains is recommended. This results in smaller subdomain problems and removes much of the MPI communication cost associated with the construction of the overlap regions.

There are two strategies for constructing the AAS subdomain problems. The first is to define serial subdomain problems, which will correspond directly to the distribution of the degrees of freedom of the global problem, resulting from the mesh partitioning process. In this case, the recommended way to solve the subdomain problems is with an exact LU factorization, which leads to a minimal number of GMRES iterations. There is a variety of direct solver packages which can be used for this: UMFPACK, MUMPS or PARDISO. While PARDISO generally offers superior performance to the other solvers, it is not free open source software and is not available on all target platforms. An additional feature of PARDISO is multi-threading. In the situation described in the previous section, when all the CPUs on a supercomputer node can not be fully subscribed using MPI processes, the PARDISO solver can be used in multi-threaded mode in order to supplement the MPI parallelism and make better use of the hardware.

A second approach to building the AAS preconditioner is with parallel subdomain problems, constructed by putting together the rows in the linear system associated with multiple connected mesh parts. This approach, developed and studied in the course of this thesis, is at this point slower than the case of serial subdomain problems, although it exhibits very good strong scalability. The ShyLU preconditioner, based on a Schur complement algorithm, should be used to solve inexactly the parallel subdomain problems.

The two main factors for the performance of ShyLU are the choice of the LU solver for the diagonal blocks and the treatment of the Schur complement system. For the inversion of the diagonal blocks, the same solvers as in the case of serial subdomain problems should be used. PARDISO is preferred, but ultimately the choice is affected by availability. For the resolution of the Schur complement linear system, the probing method can be used to compute a sparse approximation of the Schur complement matrix and the resulting linear system can be solved inexactly using a small number of GMRES subiterations. Alternatively, the IQR preconditioner represents an alternative to the probing method. With IQR, the Schur complement linear system can be solved inexactly without the need to explicitly compute the Schur complement

<table>
<tr><td>Preprocessing</td><td colspan="3">• offline mesh partitioning, runtime loading
• mesh partitioning with overlap of 1 finite element strip
• hierarchical partitioning when $N_S > 1$</td></tr>
<tr><td>FE assembly</td><td colspan="3">• precomputed matrix graph
• if $N_P < N_{CPU}$ use multi-threaded FE assembly</td></tr>
<tr><td>Solver</td><td colspan="3">• preconditioned iterative solver: CG, GMRES</td></tr>
<tr><td rowspan="9">Preconditioner</td><td colspan="3">• for Navier-Stokes problems: PCD, SIMPLE, Yosida
• for the AD block or Laplacian, ADR problems: 2-level AAS</td></tr>
<tr><td rowspan="2">2-level AAS</td><td>Coarse level:</td><td colspan="2">3-5 Gauss-Seidel iterations</td></tr>
<tr><td>Fine level:</td><td colspan="2">AAS with minimal overlap ($N_{DD} = N_P / N_S$)</td></tr>
<tr><td rowspan="6">subdomain solver (fine level)</td><td>$N_S = 1$</td><td colspan="2">$N_S > 1$</td></tr>
<tr><td>Exact LU factorization</td><td colspan="2">ShyLU</td></tr>
<tr><td rowspan="4">UMFPACK, MUMPS, PARDISO (multi-threaded if $N_P < N_{CPU}$)</td><td colspan="2">Block solver: exact LU</td></tr>
<tr><td colspan="2">Schur complement approximation: IQR or Probing</td></tr>
<tr><td>IQR</td><td>Probing</td></tr>
<tr><td>Dimension of Krylov subspace 0.5% of size of Schur complement matrix</td><td>Diagonal factor 2% of size of Schur complement matrix
5 GMRES subiterations</td></tr>
</table>

Table 8.1: Recipe table for finite element simulations with LifeV and Trilinos. N_P is the number of MPI processes in use, N_{CPU} is the total number of CPUs available, N_{DD} is the number of subdomains for the AAS preconditioner and N_S is the number of MPI processes per subdomain.

matrix or a sparse approximation of it. This latter approach, which we have introduced and described in this thesis, leads to much improved CPU time to compute and apply the ShyLU preconditioner, at the cost of a larger number of iterations in the outer linear solver.

“Recipe” table

Finally, we provide Table 8.1, synthetizing the ideas in Section 8.3, to serve as a quick reference for the setup and configuration of finite element simulations with LifeV and Trilinos, using the new developments introduced in this thesis.

8.4 Future work

The work we have done in the course of this thesis has helped us identify a number of research and development directions, which we believe are worth being pursued.

The IQR preconditioner, used as an inexact solver for the Schur complement system inside ShyLU proved to give a valuable increase in performance. We would like to further investigate IQR and bring improvements both to the algorithmic side and to its implementation. Additionally we would like to investigate other uses for IQR, separate from the ShyLU preconditioner.

The ShyLU preconditioner has good strong scalability but comes with the disadvantage that the Schur complement algorithm is implemented using MPI only, due to memory limitations. We have identified scenarios when it is not possible to fully utilize available hardware resources using only MPI parallelism. The multi-threaded direct solver PARDISO, which can be used as a subdomain solver on the fine level of AAS, has good performance but it is not free open source software and its availablity is limited. We would like to develop a multi-threaded implementation of the ShyLU algorithm, also making use of IQR. The new implementation will be integrated in the Trilinos software project, where it will serve as an open source and freely available alternative to existing multi-threaded preconditioners and inexact solvers.

Saving the solution during large scale simulations in LifeV we encounter similar performance problems as with the early implementation of the parallel mesh loading routines. It would be worthwhile to apply the improvements that have been done on the parallel mesh loading routines also to the solution output. This will lead to a much better usage of the allocated resources which are available for the LifeV project on supercomputers.

Finally, we would like to continue the refactoring of LifeV's linear algebra foundation classes, migrating them to the newer branch of linear algebra packages from Trilinos. The benefit of this endeavour would be improving the support and perfomance of multi-threaded operation, which could be added, in LifeV, also to other stages of the simulation process.

Bibliography

[1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "Top500 supercomputing sites," 2011.

[2] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: the complete reference.* MIT press, 1995.

[3] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 427–436, IEEE, 2009.

[4] B. Barney, "POSIX threads programming," *Lawrence Livermore National Laboratory, https://computing. llnl. gov/tutorials/pthreads/, available online*, vol. 2010, 2009.

[5] D. Butenhof, *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[6] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[7] B. Chapman, G. Jost, and R. V. d. Pas, *Using OpenMP: portable shared memory parallel programming.* MIT Press, 2008.

[8] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2010.

[9] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pp. 35–43, IEEE, 2007.

[10] S. Krakiwsky, L. Turner, and M. Okoniewski, "Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU)," in *Microwave Symposium Digest, 2004 IEEE MTT-S International*, vol. 2, pp. 1033 – 1036 Vol.2, June 2004.

[11] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, (Aire-la-Ville, Switzerland, Switzerland), pp. 112–119, Eurographics Association, 2003.

[12] I. Buck, "Taking the plunge into GPU computing," *GPU Gems*, vol. 2, pp. 509–519, 2005.

[13] C. CUDA, "Programming guide," *NVIDIA Corporation, July*, 2012.

[14] A. M. Bayoumi, M. Chu, Y. Y. Hanafy, P. Harrell, and G. Refai-Ahmed, "Scientific and engineering computing using ATI stream technology.," *Computing in Science and Engineering*, vol. 11, no. 6, pp. 92–97, 2009.

[15] D. Komatitsch, "Fluid–solid coupling on a cluster of GPU graphics cards for seismic wave propagation," *Comptes Rendus Mécanique*, vol. 339, pp. 125–135, Feb. 2011.

[16] N. Goedel, T. Warburton, and M. Clemens, "GPU accelerated discontinuous galerkin FEM for electromagnetic radio frequency problems," in *Antennas and Propagation Society International Symposium, 2009. APSURSI '09. IEEE*, pp. 1 –4, June 2009.

[17] A. Klöckner, T. Warburton, J. Bridge, and J. Hesthaven, "Nodal discontinuous galerkin methods on graphics processors," *Journal of Computational Physics*, vol. 228, pp. 7863–7882, Nov. 2009.

[18] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, vol. 54. Springer, 2008.

[19] N. Goedel, N. Nunn, T. Warburton, and M. Clemens, "Scalability of higher-order discontinuous galerkin FEM computations for solving electromagnetic wave propagation problems on GPU clusters," *Magnetics, IEEE Transactions on*, vol. 46, pp. 3469 –3472, Aug. 2010.

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, *et al.*, "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[21] P. Crosetto, P. Reymond, S. Deparis, D. Kontaxakis, N. Stergiopulos, and A. Quarteroni, "Fluid–structure interaction simulation of aortic blood flow," *Computers & Fluids*, vol. 43, no. 1, pp. 46–57, 2011.

[22] P. Crosetto, S. Deparis, G. Fourestey, and A. Quarteroni, "Parallel algorithms for fluid-structure interaction problems in haemodynamics," *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 1598–1622, 2011.

[23] M. Discacciati, D. Hacker, A. Quarteroni, S. Quinodoz, S. Tissot, and F. M. Wurm, "Numerical simulation of orbitally shaken viscous fluids with free surface," *International Journal for Numerical Methods in Fluids*, vol. 71, no. 3, pp. 294–315, 2013.

[24] S. Quinodoz, *Numerical Simulation of Orbitally Shaken Reactors*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2012.

[25] A. C. I. Malossi, P. J. Blanco, S. Deparis, and A. Quarteroni, "Algorithms for the partitioned solution of weakly coupled fluid models for cardiovascular flows," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 27, no. 12, pp. 2035–2057, 2011.

[26] A. C. I. Malossi, *Partitioned Solution of Geometrical Multiscale Problems for the Cardiovascular System: Models, Algorithms, and Applications.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2012.

[27] A. Quarteroni and A. Valli, *Domain Decomposition Methods for Partial Differential Equations.* Oxford: Oxford Science Publications, 1999.

[28] M. Sala and M. Heroux, "Robust algebraic preconditioners with IFPACK 3.0," Tech. Rep. SAND-0662, Sandia National Laboratories, 2005.

[29] M. Dryja and O. B. Widlund, "Additive schwarz methods for elliptic finite element problems in three dimensions," in *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[30] J. J. Hu, C. M. Siefert, I. Karlin, R. S. Tuminaro, S. P. Domino, and A. C. Robinson, "Highly scalable linear solvers on thousands of processors," tech. rep.

[31] S. Rajamanickam, E. G. Boman, and M. A. Heroux, "ShyLU: a hybrid-hybrid solver for multicore platforms," *Accepted for proceedings of IPDPS 2012*, 2012.

[32] S. Deparis, M. Discacciati, and A. Quarteroni, "domain decomposition framework for fluid-structure interaction problems," in *Third International Conference on Computational Fluid Dynamics (ICCFD3)*, 2004.

[33] A. Quarteroni, *Numerical models for differential problems*, vol. 2. Springer Verlag, 2009.

[34] T. A. Davis, *Direct Methods for Sparse Linear Systems.* SIAM, Sept. 2006.

[35] Y. Saad, *Iterative methods for sparse linear systems.* Society for Industrial Mathematics, 2003.

[36] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical Mathematics.* New York: Springer-Verlag, 2000.

[37] H. Elman, D. Silvester, and A. Wathen, "Finite elements and fast iterative solvers: with applications in incompressible fluid dynamics," Oxford: Oxford University Press, 2005. xiv+400 pp. ISBN: 978-0-19-852868-5; 0-19-852868-X.

[38] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[39] M. Gschwind, "Blue gene/q: design for sustained multi-petaflop computing," in *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 245–246, ACM, 2012.

[40] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione, “Early experiences with the intel many integrated cores accelerated computing technology,” in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG ’11, (New York, NY, USA), pp. 21:1–21:8, ACM, 2011.

[41] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, “An evaluation of vectorizing compilers,” in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 372–382, IEEE, 2011.

[42] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, “Automatic simd vectorization of fast fourier transforms for the larrabee and avx instruction sets,” in *Proceedings of the international conference on Supercomputing*, ICS ’11, (New York, NY, USA), pp. 265–274, ACM, 2011.

[43] T. A. Davis, “A column pre-ordering strategy for the unsymmetric-pattern multifrontal method,” *ACM Trans. Math. Softw.*, vol. 30, p. 165–195, June 2004.

[44] P. Amestoy, I. Duff, and J.-Y. L’Excellent, “Multifrontal parallel distributed symmetric and unsymmetric solvers,” *Computer Methods in Applied Mechanics and Engineering*, vol. 184, pp. 501–520, Apr. 2000.

[45] C. Kelley, “Iterative methods for linear and nonlinear equations, siam, philadelphia, 1995,” *MR 96d*, vol. 65002.

[46] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), (New York, NY, USA), p. 483–485, ACM, 1967.

[47] J. L. Gustafson, “Reevaluating amdahl’s law,” *Communications of the ACM*, vol. 31, p. 532–533, 1988.

[48] G. Karypis, K. Schloegel, and V. Kumar, “ParMETIS: parallel graph partitioning and sparse matrix ordering library version 3.1,” *University of Minnesota, Minneapolis*, 2003.

[49] C. Chevalier and F. Pellegrini, “PT-Scotch: a tool for efficient parallel graph ordering,” *Parallel Computing*, vol. 34, pp. 318–331, July 2008.

[50] X. Li and J. Demmel, “SuperLU DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems ACM trans,” *Math. Soft*, vol. 29, no. 2, p. 110–140, 2003.

[51] A. Toselli and O. Widlund, *Domain decomposition methods—algorithms and theory*, vol. 34 of *Springer Series in Computational Mathematics*. Berlin: Springer-Verlag, 2005.

[52] W. Hackbusch, *Multi-grid methods and applications*, vol. 4. Springer-Verlag Berlin, 1985.

[53] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, “An overview of the hdf5 technology suite and its applications,” in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD ’11, (New York, NY, USA), pp. 36–47, ACM, 2011.

[54] C. Multiphysics, “3.4, comsol inc,” 2007.

[55] A. Fluent, “12.0 theory guide,” *Ansys Inc*, vol. 5, 2009.

[56] A. Version, “6.4 user’s manual,” *ABAQUS Inc. Pawtucket USA*, 2003.

[57] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, “Efficient management of parallelism in object oriented numerical software libraries,” in *Modern Software Tools in Scientific Computing* (E. Arge, A. M. Bruaset, and H. P. Langtangen, eds.), pp. 163–202, Birkhäuser Press, 1997.

[58] G. Van Rossum and F. L. Drake, *The Python Language Reference Manual.* Network Theory Ltd., 2011.

[59] A. Logg, K.-A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method: The fenics book*, vol. 84. Springer, 2012.

[60] W. Bangerth, R. Hartmann, and G. Kanschat, “deal. ii—a general-purpose object-oriented finite element library,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 24, 2007.

[61] A. Dedner, *Advances in DUNE.* Springer, 2012.

[62] H. Jasak, A. Jemcov, and Z. Tukovic, “Openfoam: A c++ library for complex physics simulations,” in *International Workshop on Coupled Methods in Numerical Dynamics, IUC, Dubrovnik, Croatia*, pp. 1–20, 2007.

[63] C. Prud’Homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena, “Feel++: A computational framework for galerkin methods and advanced numerical methods,” in *ESAIM: Proceedings*, vol. 38, pp. 429–455, EDP Sciences, 2012.

[64] M. Lyly, J. Ruokolainen, and E. Järvinen, “Elmer–a finite element solver for multiphysics,” *CSC-report on scientific computing*, vol. 2000, pp. 156–159, 1999.

[65] D. Göddeke, R. Strzodka, and S. Turek, “Accelerating double precision FEM simulations with GPUs,” in *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sept. 2005.

[66] D. Goddeke, S. Buijssen, H. Wobker, and S. Turek, “GPU acceleration of an unmodified parallel finite element navier-stokes solver,” in *High Performance Computing Simulation, 2009. HPCS ’09. International Conference on*, pp. 12 –21, June 2009.

[67] G. Haase, M. Liebmann, C. Douglas, and G. Plank, “A parallel algebraic multigrid solver on graphics processing units,” in *High Performance Computing and Applications* (W. Zhang, Z. Chen, C. Douglas, and W. Tong, eds.), vol. 5938 of *Lecture Notes in Computer Science*, pp. 38–47, Springer Berlin / Heidelberg, 2010.

[68] A. Dziekonski, A. Lamecki, and M. Mrozowski, "GPU acceleration of multilevel solvers for analysis of microwave components with finite element method," *Microwave and Wireless Components Letters, IEEE*, vol. 21, pp. 1 –3, Jan. 2011.

[69] A. Dziekonski, A. Lamecki, and M. Mrozowski, "Tuning a hybrid GPU-CPU v-cycle multilevel preconditioner for solving large real and complex systems of FEM equations," *Antennas and Wireless Propagation Letters, IEEE*, vol. 10, pp. 619 –622, 2011.

[70] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, pp. 451–460, May 2009.

[71] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, pp. 7692–7714, Oct. 2010.

[72] G. R. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations on GPU – application to neurosurgical simulation," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 3305–3314, Dec. 2010.

[73] N. Goedel, S. Schomann, T. Warburton, and M. Clemens, "GPU accelerated adams-bashforth multirate discontinuous galerkin FEM simulation of high-frequency electromagnetic fields," *Magnetics, IEEE Transactions on*, vol. 46, pp. 2735 –2738, Aug. 2010.

[74] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 4, p. 17, 2008.

[75] D. Göddeke, R. Strzodka, and S. Turek, "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4, pp. 221–256, 2007.

[76] U. Villa, *Scalable Efficient Methods for Incompressible Fluid-dynamics in Engineering Problems.* PhD thesis, Emory University, Atlanta, GA, 2012.

[77] G. Grandperrin, *Parallel and scalable preconditioners for the Navier-Stokes equations: applications to hemodynamic problems.* PhD thesis, EPFL, 2013.

[78] P. Crosetto, *Fluid-Structure Interaction Problems in Hemodynamics: Parallel Solvers, Preconditioners, and Applications.* PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, 2011.

[79] M. Heroux, "Design issues for numerical libraries on scalable multicore architectures," in *Journal of Physics: Conference Series*, vol. 125, p. 012035, 2008.

[80] H. A. Schwarz, "Ueber einige abbildungsaufgaben.," *Journal für die reine und angewandte Mathematik*, vol. 70, pp. 105–120, 1869.

[81] M. Dryja and O. B. Widlund, “Some recent results on schwarz type domain decomposition algorithms,” *CONTEMPORARY MATHEMATICS*, vol. 157, pp. 53–53, 1994.

[82] M. Dryja, B. F. Smith, and O. B. Widlund, “Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions,” *SIAM journal on numerical analysis*, vol. 31, no. 6, pp. 1662–1694, 1994.

[83] G. Golub and C. Van Loan, *Matrix computations*. Johns Hopkins University Press, third ed., 1996.

[84] S. C. Brenner, “Lower bounds for two-level additive schwarz preconditioners with small overlap,” *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1657–1669, 2000.

[85] S. F. McCormick, *Multigrid methods*, vol. 19. siam, 1987.

[86] E. Efstathiou and M. J. Gander, “Why restricted additive schwarz converges faster than additive schwarz,” *BIT Numerical Mathematics*, vol. 43, no. 5, pp. 945–959, 2003.

[87] O. Schenk, M. Bollhöfer, and R. A. Römer, “On large-scale diagonalization techniques for the anderson model of localization,” *SIAM review*, vol. 50, no. 1, pp. 91–112, 2008.

[88] T. Davis and K. Stanley, “Klu: a" clark kent" sparse lu factorization algorithm for circuit matrices,” in *2004 SIAM Conference on Parallel Processing for Scientific Computing (PP04)*, 2004.

[89] J. Gaidamour and P. Henon, “A parallel Direct/Iterative solver based on a schur complement approach,” in *Computational Science and Engineering, 2008. CSE ’08. 11th IEEE International Conference on*, pp. 98 –105, July 2008.

[90] L. Giraud, A. Haidar, and L. Watson, “Parallel scalability study of hybrid preconditioners in three dimensions,” *Parallel Computing*, vol. 34, no. 6, pp. 363–379, 2008.

[91] A. Haidar, “On the parallel scalability of hybrid linear solvers for large 3d problems,” 2008.

[92] T. F. Chan and T. P. Mathew, “The interface probing technique in domain decomposition,” *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 212–238, 1992.

[93] Y. Saad and M. H. Schultz, “Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems,” *SIAM Journal on scientific and statistical computing*, vol. 7, no. 3, pp. 856–869, 1986.

[94] S. Deparis, *Numerical analysis of axisymmetric flows and methods for fluid-structure interaction arising in blood flow simulation*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2004.

[95] K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan, “Design of dynamic load-balancing tools for parallel applications,” in *Proceedings of the 14th international conference on Supercomputing*, ICS ’00, (New York, NY, USA), pp. 110–118, ACM, 2000.

[96] Message Passing Interface Forum, "A Message-Passing Interface Standard - Version 3.0," 2012.

[97] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Leveraging mpi's one-sided communication interface for shared-memory programming," in *Recent advances in the message passing interface*, pp. 132–141, Springer, 2012.

[98] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi+ mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, pp. 1–16, 2013.

[99] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pp. 182–189, IEEE, 1999.

[100] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pp. 23–32, ACM, 1999.

[101] J.-M. Muller, *Handbook of floating-point arithmetic.* Birkhauser Boston, 2010.

[102] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, *et al.*, "The IBM BlueGene/Q compute chip," *Micro, IEEE*, vol. 32, no. 2, pp. 48–60, 2012.

[103] D. Heroux and A. Michael, "Tpetra, and the use of generic programming in scientific computing," *Scientific Programming*, vol. 20, no. 2, 2012.

[104] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012.

[105] R. Temam, *Navier–Stokes Equations.* American Mathematical Soc., 1984.

[106] F. Brezzi, "On the existence, uniqueness and approximation of saddle-point problems arising from lagrangian multipliers," *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique*, vol. 8, no. R2, pp. 129–151, 1974.

[107] F. Brezzi and M. Fortin, *Mixed and hybrid finite element methods.* Springer-Verlag New York, Inc., 1991.

[108] A. Quarteroni and A. Valli, *Numerical approximation of partial differential equations (Series in computational mathematics, Vol. 23).* Springer, 2008.

[109] H. C. Elman, D. J. Silvester, and A. J. Wathen, *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics.* Oxford University Press, 2005.

[110] H. Elman, V. E. Howle, J. Shadid, R. Shuttleworth, and R. Tuminaro, “Block preconditioners based on approximate commutators,” *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1651–1668, 2006.

[111] S. V. Patankar, *Numerical heat transfer and fluid flow.* Taylor & Francis, 1980.

[112] A. Quarteroni, F. Saleri, and A. Veneziani, “Factorization methods for the numerical approximation of Navier–Stokes equations,” *Computer methods in applied mechanics and engineering*, vol. 188, no. 1, pp. 505–526, 2000.

Chemin du Stand 19B
1024, Ecublens, Switzerland
☎ *+41 76 704 9694*
✉ *i.radu.popescu@gmail.com*
Doctoral assistant
High Performance Computing
Diploma Engineer
Electrical Engineering

Radu Popescu

Education

2010–2013 **Doctoral studies**, *Ecole Polytechnique Federale de Lausanne, Chair of Modelling and Scientific Computing*, Lausanne, Switzerland.

- Subject of thesis: *Parallel algorithms and efficient implementation techniques for finite element approximations.*
- Co advisors: Simone Deparis and Alfio Quarteroni
- Oct - Nov 2011 - Internship at Sandia National Laboratories, Albuquerque, New Mexico, USA, for the development of two-level MPI parallelism for the Algebraic Additive Schwarz preconditioner in the Trilinos numberical libraries. Mentor: Michael A. Heroux

2004–2009 **Engineer Diploma**, *POLITEHNICA University, Faculty of Electrical Engineering*, Bucharest, Romania.

- 5 year studies, equivalent BSc. + MSc.
- Diploma project in Computer Aided Electrical Engineering: *Numerical modelling and simulation of a system for detection of buried objects based on GPR technology*

2000–2004 **High School Diploma**, *"Grigore Moisil" National College of Computer Science*, Brasov, Romania.

Experience

2010–2013 **Doctoral assistant**, *Ecole Polytechnique Federale de Lausanne, Chair of Modelling and Scientific Computing*, Lausanne, Switzerland.

Research in parallel finite element modelling, software development

- Investigation of efficient implementations of parallel finite element software on modern high perfomance computing architectures (IBM BlueGene Q and P, Cray XE/XK)
- Software development of a C++ parallel finite element modelling library, LifeV (www.lifev.org), member of a larger team of developers
- Proposed and implemented improvements to the software engineering process for the LifeV project: Git version control, peer reviewed coding, issue tracking, autoconfiguration with CMake

2009–2010 **Junior Researcher**, *POLITEHNICA University of Bucharest, Centre for Scientific Computing in Electrical Engineering*, Bucharest, Romania.

Research in high performance computing for computational electromagnetics

- Investigation of possible strategies for porting a computational electromagnetics library, written in Matlab, to parallel distributed memory architectures.

July, 2008– Jan, 2009 **Junior software developer**, *SC InterNET SRL*, Bucharest, Romania.

Software development in testing and measurements, signal acquisition and processing

Courses

Sept 2009	Sun Microsystems High Performance Computing Workshop, Regensburg, Germany
Sept 2009	COMSON Autumn School on Model Order Reduction, Terschelling, Netherlands
Aug 2010	Parallel Programming Workshop at the Swiss Centre for Scientific Computing (CSCS), Manno, Switzerland

Conferences and workshops

February 2012	SIAM Parallel Processing 2012, Contributed talk: *A hybrid algorithm for a finite element solver based on two-level parallelism*. Savannah, USA
June 2013	European Trilinos User Group Meeting, Contributed talk: *Hardware aware implementation strategies for finite element modeling with Trilinos and LifeV*. Munchen, Germany

Computer skills

Languages:	C, C++, Lua, Python, Matlab
Parallel programming:	MPI, OpenMP, POSIX threads, some GPGPU programming
Software engineering:	Version control systems (Git, Mercurial etc.), CMake, Autotools, automated testing, performance analysis
HPC arch:	IBM Bluegene/P, Bluegene/Q, Cray XE and XK
Performance analysis:	Craypat, Tau, Intel VTune
Operating systems:	Extensive experience with UNIX and Linux
Projects:	Part of the development team of the LifeV parallel FEM library (cmcs-forge.epfl.ch/projects/lifev); collaborated with developers of the Trilinos project at Sandia National Laboratories and contributed code (trilinos.sandia.gov)

Languages

English	**Excellent**
French	**Intermediate**
Romanian	**Native speaker**