KTH Computer Science
and Communication

High Performance Adaptive Finite Element Methods

With Applications in Aerodynamics

NICLAS JANSSON

Doctoral Thesis
Stockholm, Sweden 2013

TRITA-CSC-A 2013:07
ISSN 1653-5723
ISRN KTH/CSC/A–13/07-SE
ISBN 978-91-7501-814-0

Skolan för datavetenskap och kommunikation
Kungliga Tekniska högskolan
SE–100 44 Stockholm
Sverige / Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i numerisk analys onsdagen den 11 september 2013 klockan 10.15 i sal F3, Sing–Sing, Kungl Tekniska högskolan, Lindstedtsvägen 26, Stockholm.

Tryck: Universitetsservice US AB

Abstract

The massive computational cost for resolving all scales in a turbulent flow makes a direct numerical simulation of the underlying Navier-Stokes equations impossible in most engineering applications. Recent advances in adaptive finite element methods offer a new powerful tool in Computational Fluid Dynamics (CFD). The computational cost for simulating turbulent flow can be minimized by adaptively resolution of the mesh, based on a posteriori error estimation. Such adaptive methods have previously been implemented for efficient serial computations, but the extension to an efficient parallel solver is a challenging task.

This work concerns the development of an adaptive finite element method that enables efficient computation of time resolved approximations of turbulent flow for complex geometries with a posteriori error control. We present efficient data structures and data decomposition methods for distributed unstructured tetrahedral meshes. Our work also concerns an efficient parallelization of local mesh refinement methods such as recursive longest edge bisection, and the development of an a priori predictive dynamic load balancing method, based on a weighted dual graph.

We also address the challenges of emerging supercomputer architectures with the development of new hybrid parallel programming models, combining traditional message passing with lightweight one-sided communication. Our implementation has proven to be both general and efficient, scaling up to more than twelve thousands cores.

Sammanfattning

Den höga beräkningskostnaden för att lösa upp alla turbulenta skalor för ett realistiskt problem gör en direkt numerisk simulering av Navier-Stokes ekvationer omöjlig. De senaste framstegen inom adaptiva finita element metoder ger ett nytt kraftfullt verktyg inom Computational Fluid Dynamics (CFD). Beräkningskostnaden för en simulering av turbulent flöde kan minimeras genom att beräkningsnätet adaptivt förfinas baserat på en a posteriori feluppskattning. Dessa adaptiva metoder har tidigare implementerats för seriella beräkningar, medan en effektiv parallellisering av metoden inte är trivial.

I denna avhandling presenterar vi vår utveckling av en adaptiv finita element lösare, anpassad för att effektivt beräkna tidsupplösta approximationer i komplicerade geometrier med a posteriori felkontroll. Effektiva datastrukturer och metoder för ostrukturerade beräkningsnät av tetrahedrar presenteras. Avhandlingen behandlar även effektiv parallellisering av lokala nätförfiningsmetoder, exempelvis recursive longest edge bisection. Även lastbalanseringsproblematiken behandlas, där problemet lösts genom utvecklandet av en prediktiv dynamisk lastbalanseringsmetod, baserad på en viktad dualgraf av beräkningsnätet.

Slutligen avhandlas även problematiken med att effektivt utnyttja nytillkomna superdatorarkitekturer, genom utvecklandet av en hybrid parallelliserings modell som kombinerar traditionell meddelande baserad parallellisering med envägskommunikation. Detta har resulterat i en generell samt effektiv implementation med god skalning upp till fler än tolv tusen processorkärnor.

Preface

This thesis consists of an introduction and nine papers.

Paper I

N. Jansson, J. Hoffman and J. Jansson. Framework for Massively Parallel Adaptive Finite Element Computational Fluid Dynamics on Tetrahedral Meshes. *SIAM Journal on Scientific Computing 34(1):C24-C41, 2012.*

Paper II

J. Hoffman and N. Jansson. A computational study of turbulent flow separation for a circular cylinder using skin friction boundary conditions. In *Quality and Reliability of Large-Eddy Simulations II, ERCOFTAC Series 17, Springer, 2011.*

Paper III

N. Jansson, J. Hoffman, and M. Nazarov. Adaptive Simulation of Turbulent Flow Past a Full Car Model. *In Proceedings of the 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, State of the Practice Reports, SC '11, 2011.*

Paper IV

J. Hoffman et. al. Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for deforming domains and complex geometry. *Computers & Fluids 2012, Volume 80, pp. 310-319 , 2013.*

Paper V

N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication. *VECPAR 2012, Lecture Notes in Computer Science 7851:128-139, 2013.*

Paper VI

N. Jansson. Towards a Parallel Algebraic Multigrid Solver Using Partitioned Global Address Space. *To be submitted, 2013.*

Paper VII

N. Jansson and J. Hoffman. Improving Parallel Performance of FEniCS Finite Element Computations by Hybrid MPI/PGAS. *In review.*

Paper VIII

R. Vilela De Abreu, N. Jansson and J. Hoffman. Adaptive Computation of Aeroacoustic Sources for a Rudimentary Landing Gear. *In review.*

Paper IX

R. Vilela De Abreu, N. Jansson and J. Hoffman. Computation of Aeroacoustic Sources for a Gulfstream G550 Nose Landing Gear Model Using Adaptive FEM. *To be submitted, 2013.*

Paper I, V-VII: The author of this thesis developed the ideas, wrote the implementations and prepared the manuscripts.

Paper II: The author of this thesis performed the computations, and contributed to the implementation of the method.

Paper III: The author of this thesis performed the computations and prepared the manuscript.

Paper IV: The author prepared parts of the manuscript and contributed to the development of the methods.

Paper VIII-IX: The computations and development of new methods were carried out in close cooperation between the first two authors.

Acknowledgments

First I would like to thank my supervisor Johan Hoffman and co-supervisor Johan Jansson for giving me the opportunity to work on such a challenging subject, and their trust in letting me shape my research according to my interest. Thank you for introducing me to the field of computational fluid dynamics and to the world of adaptive finite element methods.

I thank Jesper Oppelstrup for interesting discussions, encouragement and for reading through this thesis, contributing with many suggestions for improvements.

I would also like to thank my colleagues at the Dept. of Numerical Analysis and the Dept. of High Performance Computing and Visualization for a friendly and creative work environment. I am particularly grateful to my colleague Rodrigo Vilela de Abreu for a fruitful collaboration and interesting discussions. Furthermore, I would like to express my gratitude to the people at PDC, for letting me use their supercomputers, providing support and vast amount of computer time in times of need.

I thank Jeannette Spühler, who had to endure my endless rants about computer architectures, constant complaints about single node jobs, late night monologues about various aspects of algebraic multigrid and for reading through my thesis. Finally I would like to thank my family for their continuous support, regardless the constantly increasing amount of computer hardware in their basement.

Financial support from the Swedish Foundation for Strategic Research is gratefully acknowledged.

Contents

Part I

Introductory chapters

Chapter 1

Introduction

Understanding the irregular motion of turbulent flow is one of the key challenges facing engineers in several fields, for example aerodynamics, and aero-acoustics. In order to optimize a given design one has to either rely on expensive wind tunnel experiments or derive a mathematical model that can be used to simulate turbulent flow with a computer. We model the motion of incompressible Newtonian fluid flow by the Navier-Stokes equations with constant kinematic viscosity $\nu > 0$, in $\Omega \subset \mathbb{R}^3$ over a time interval $I = (0, T]$:

$$\begin{aligned} \dot{u} + (u \cdot \nabla)u + \nabla p - \nu\Delta u &= f, && (x,t) \in \Omega \times I, \\ \nabla \cdot u &= 0, && (x,t) \in \Omega \times I, \\ u(x,0) &= u^0(x), && x \in \Omega, \end{aligned} \tag{1.1}$$

with $u(x,t)$ the velocity vector, $p(x,t)$ the pressure, $u^0(x)$ initial data and a body force $f(x,t)$. The stress tensor is defined by $\sigma_{ij} = -\mu\epsilon_{ij}(u) + p\delta_{ij}$, with constant dynamic viscosity $\mu > 0$, the strain rate tensor $\epsilon_{ij} = 1/2(\partial u_i/\partial x_j + \partial u_j/\partial x_i)$ and the Kronecker delta function δ_{ij}.

A tremendous amount of computing power has been expended to solve these equations, often for simple geometries, far from engineering applications. Therefore, computing a turbulent solution for a complex geometry is often referred to as a *grand challenge* problem.

The ability to compute a time resolved solution to (1.1) for a realistic geometry, within a reasonable amount of time, would open up a new world of possibilities for flow simulation. In the future an engineer might be able to perform such a simulation on a workstation. Until then we have to rely on large parallel computers.

However, even with cutting edge high performance hardware such a simulation still takes too long. Adaptive finite element methods provide powerful and cost effective techniques for simulating turbulent flow on a serial computer by optimization of the computational mesh. If it was possible to implement these methods efficiently on a parallel computer, it would enable time resolved simulation of turbulent flow past complex geometries, within a reasonable amount of time. Furthermore, we also strive towards a general method that can be applied to a wide class of problems, unlike today's scientific software that is often tailored for a certain application.

Based on recent advances in adaptive computation of turbulent flow based on General Galerkin (G2) finite element methods, the main part of this work concerns the efficient simulation of fluid flow using adaptive methods on modern distributed memory machines. We develop suitable data structures and algorithms that enable us to derive an efficient solver which shows near linear strong scaling to thousands of cores. This new work enables us to accurately simulate turbulent flow past a complex geometry, for example a landing gear.

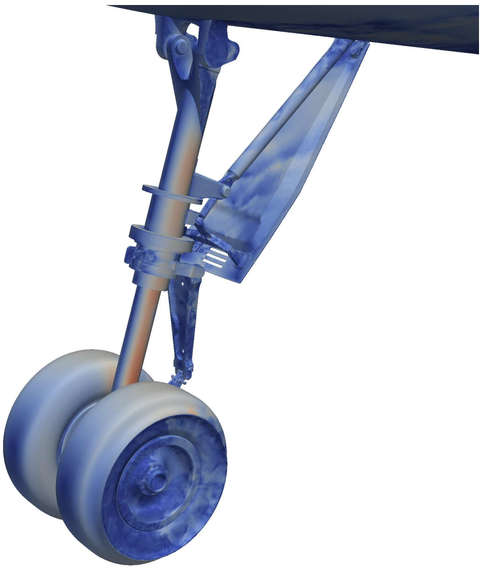


Figure 1.1: An example of a large scale problem. Turbulent flow computed around a Gulfstream G550 nose landing gear. For details see [44] and Paper IX.

The remainder of this thesis is organized as follows. First we give an introduction to high performance computing in chapter 2, and define the setting for our parallel computations. In chapter 3 we present the cG(1)cG(1) method for simulation of turbulent fluid flow. In chapter 4 we present the foundation for a high performance adaptive finite element solver, describe in detail which components are needed and different parallelization strategies. In chapter 5 we discuss issues in formulating efficient parallel linear solvers. Chapter 6 describes our implementation in the open source project FEniCS. Finally in chapter 7 we present examples of grand challenge problems which we were able to solve due to the contribution of this thesis.

Chapter 2

High performance computing

High performance computing is a rapidly evolving field where architectures and programming paradigms are likely to change from one year to the next. A good example is the recent shift towards parallel computing in commodity hardware with chip multicore processors (cmp). Parallel computing is not new, it has been used to tackle grand challenge problems almost as long as supercomputers have been around but the recent introduction of cmp hardware has changed the meaning of high performance computing to always involve some kind of massive parallelism.

2.1 Parallel architectures

In recent years the state of the art in parallel computer architectures has changed rapidly from systems with a few processors, symmetric multiprocessors (smp), to massively parallel systems with thousands of multicore processors. Up until recently the unit often used to describe the size of a system was the number of processors. In order to make it clear we from now on use the term processing element (PE) to describe the number of units that can perform useful computational work in the system, processors or most likely the number of cores.

A coarse grained classification of parallel architectures can be made depending on how PEs are able to access the memory. In shared memory architectures all the processing elements can access the same global memory via a bus or a high speed crossbar switch. These systems can scale to around one thousand PEs with a common global memory address space. However, for large number of cores the cost to access memory is seldom uniform. Hence, on these systems much work has to be invested in optimizing how PEs are accessing memory such that cache misses and memory contentions are reduced.

The distributed memory architectures have one or several PEs with local memory contained inside each node. These nodes are then connected together with a high speed network. These systems can scale to an extremely large number of PEs, more than one million. However, the price to pay for such an extreme scalability is the interconnect latency, which is often several magnitudes higher than the time to access local memory. Since distributed memory systems are more common to come across, this work mostly focuses on high performance finite element computations on distributed memory architectures.

2.2 Parallel programming models

Traditionally, applications running on distributed memory architectures are implemented using the message passing model. With the problem's data distributed across all the memories, PEs exchange (local) data by exchanging messages with each others. Messages can be sent either synchronously or asynchronously, but the model is two-sided: a message sent must be received by a corresponding receive request.

Being the dominating parallel programming model for decades, among several implementations, the Message Passing Interface (MPI) [29] has over the years come out as the de facto standard. MPI has a good abstraction for distributing data across the nodes and for exchanging information in a structured way. However, its two-sided communication semantics makes handling fine-grained parallelism challenging. For example, an unstructured mesh is excellent for an accurate approximation of a complex geometry. However, the lack of underlying structure implies an unstructured exchange of messages, which make matching of send and receive requests a challenge. The message matching also introduce latency due to the necessary hand shaking between communicating PEs, increasing as the number of PEs grows. For exascale computing, this raises concern about the message passing model's applicability.

Recent advances in interconnect technology, e.g. efficient remote memory copy, has given a push for the family of Partitioned Global Address Space (PGAS) languages. Based on the abstraction of a global address space built on top of the distributed global memory, any PE may directly read or write at any remote memory location without explicit commands for communicating with any other PE (see Figure 2.1). The global memory is also partitioned such that each PE has affinity to one part of the memory, an important aspect since there is a varying cost when accessing the memory, depending on its location in the machine. The PGAS concept is a simple and elegant model, especially for algorithms with challenging data dependencies. With its one-sided communication abstraction it is also an efficient model for fine-grained parallelism.

2.3 Computational speedup

Another important factor connected to different architectures in high performance computing is the scalability and speedup of a code running on thousands of PEs.

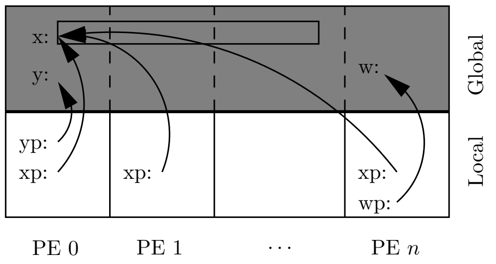


Figure 2.1: An illustration of a partitioned global address space. Here, xp, yp and wp are (per PE, private) local pointers to the global variables x,y and w.

Here we define the parallel speedup S_p of using p processing elements as:

$$S_p = \frac{T_1}{T_p} \tag{2.1}$$

where T_1 is the best sequential time and T_p is the time for running on p PEs. Now it is natural to ask ourself, what is the maximum speedup one can obtain from using p PEs? Suppose our computation depends on N independent tasks. If we use p processing elements it is obvious that the computation can be computed p times faster, since all tasks can be solved concurrently. This is an upper bound [11] which leads to the following theorem:

Theorem 1 *On a given parallel machine and for a fixed problem size, the best p processing element computation is at most p times faster than the best sequential computation*

$$T_p \geq \frac{T_1}{p}$$

In theory we can solve any large scale problem p times faster. However, there are some limiting factors that has a negative impact on the actual performance of a code. In the reasoning above we stated that all tasks can be solved concurrently. Suppose that a fraction α of the serial running time T_1 has to be executed in serial. By using p PEs we can only eliminate $(1-\alpha)T_1$ of the serial running time, hence a total parallel runtime of $T_p = \alpha T_1 + (1-\alpha)T_1/p$. Inserted into (2.1) this leads to Amdahl's law [1]:

$$S_p = \frac{p}{1 + (p-1)\alpha} \tag{2.2}$$

So even for an infinite number of PEs, the maximum speedup will still be limited by the serial fraction $S_p \leq 1/\alpha$. It should also be taken into consideration that this is a rather pessimistic upper bound. However, it illustrates the importance of eliminating serial parts in order to efficiently solve a large scale problem, the core part of this work.

Chapter 3

Simulation of turbulent flow

For high Reynolds numbers $Re = UL/\nu$, with U and L characteristic velocity and length scales and ν the viscosity, inertial effects dominate and turbulent flow develop. Turbulent flow is described by the Navier-Stokes equations (1.1), but solutions are inherently expensive to compute. State of the art numerical methods can be divided into three different categories: For the most detailed simulation one has to rely on a direct numerical simulation (DNS), which solves the Navier-Stokes equations (1.1) without any turbulence model. Hence, the method needs to resolve all scales of turbulent flow in order to be accurate, which is too expensive for the high Reynolds numbers and complex domains which we are interested in.

The industry standard today is to rely on (stationary) Reynolds averaged Navier-Stokes (RANS) simulations. These equations are time-averaged variant of Navier-Stokes equations (1.1) combined with turbulence models to represent turbulent quantities derived from the averaging process. RANS is less expensive than a full DNS simulation, but turbulence models must be constructed and RANS will not resolve certain properties of time dependent flow.

Large eddy simulation (LES) can be used to capture time dependent flow, and deliver an increased detail compared to RANS, modeling unresolved turbulent scales in a subgrid model. It is less computationally demanding than DNS but is significantly more demanding than RANS. Yet, for a complex problem such a simulation can still take several days to complete, which is too long for industry design needs.

3.1 General Galerkin (G2) for turbulent flow

The simulation methodology we use for turbulent flow computations uses a finite element method with piecewise linear approximation in space and time, and with numerical stabilization in the form of a weighted least squares method based on the residual of the equations. We refer to this method as General Galerkin (G2) [20]. It is a time resolved method, and it shares features with an “implicit” LES, i.e. LES without an explicit subgrid model, relying on the numerical method to dissipate turbulent kinetic energy. Since G2 is a finite element method with tetrahedral elements it can easily handle complex domains in contrast to DNS/LES methods based on structured meshes, which for efficiency reasons often are restricted to domains of geometric primitives. Furthermore, since full resolution of turbulent boundary layers is impossible for industry applications, we use a skin friction boundary condition suitable for very high Reynolds number flow simulations, in the spirit of basic boundary layer models [39].

3.1.1 The cG(1)cG(1) method

The cG(1)cG(1) method we use is based on the continuous Galerkin method cG(1) in space and time. In space both test and trial functions are continuous piecewise linear, but in time test functions are piecewise constant while trial functions are continuous piecewise linear. Let $0 = t_0 < t_1 < ... < t_N = T$ be a sequence of discrete time steps with associated time intervals $I_n = (t_{n-1}, t_n)$ of length $k_n = t_n - t_{n-1}$ and space-time slabs $S_n = \Omega \times I_n$, and let $W^n \subset H^1(\Omega)$ be a finite element space consisting of continuous piecewise linear functions on a tetrahedral mesh $\mathcal{T}_n = \{K\}$ of mesh size $h_n(x)$ with W^n_w the functions $v \in W^n$ satisfying the Dirichlet boundary condition $v|_\Gamma = w$, with $\Gamma \equiv \partial\Omega$.

We seek $\hat{U} = (U, P)$, continuous piecewise linear in space and time, and the cG(1)cG(1) method for the Navier-Stokes equations (1.1) with homogeneous Dirichlet boundary conditions reads: For $n = 1, ..., N$, find $(U^n, P^n) \equiv (U(t_n), P(t_n))$ with $U^n \in V^n_0 \equiv [W^n_0]^3$ and $P^n \in W^n$, such that

$$\begin{aligned} &((U^n - U^{n-1})k_n^{-1} + \bar{U}^n \cdot \nabla \bar{U}^n, v) + (2\nu\epsilon(\bar{U}^n), \epsilon(v)) - (P^n, \nabla \cdot v) \\ &+ (\nabla \cdot \bar{U}^n, q) + SD^n_\delta(\bar{U}^n, P^n; v, q) = (f, v) \quad \forall \hat{v} = (v, q) \in V^n_0 \times W^n, \end{aligned} \tag{3.1}$$

where $\bar{U}^n = \frac{1}{2}(U^n + U^{n-1})$ and P^n are piecewise constant in time over I_n, with the stabilizing term

$$\begin{aligned} &SD^n_\delta(\bar{U}^n, P^n; v, q) \equiv \\ &\qquad (\delta_1(\bar{U}^n \cdot \nabla \bar{U}^n + \nabla P^n - f), \bar{U}^n \cdot \nabla v + \nabla q) + (\delta_2 \nabla \cdot \bar{U}^n, \nabla \cdot v), \end{aligned}$$

and

$$(v, w) = \sum_{K \in \mathcal{T}_n} \int_K v \cdot w \, dx,$$

$$(\epsilon(v), \epsilon(w)) = \sum_{i,j=1}^{3} (\epsilon_{ij}(v), \epsilon_{ij}(w)),$$

with the stabilization parameters:

$$\delta_1 = \kappa_1 (k_n^{-2} + |U^{n-1}|^2 h_n^{-2})^{-1/2} \text{ and } \delta_2 = \kappa_2 |U^{n-1}| h_n,$$

where κ_1 and κ_2 are positive constants. Within an adaptive algorithm, G2 is less computational demanding than standard DNS/LES based on ad hoc construction of the mesh [17, 18].

3.1.2 Skin friction boundary layer model

For an accurate simulation of turbulent flow a key question is how to model the effect of the boundary layer. As mentioned earlier, full resolution of turbulent boundary layers is not feasible, therefore numerical methods need to be complemented by a wall layer model. This model needs not only to be accurate, it also has to be efficient and simple in order to handle large and complex domains.

For high Reynolds number turbulent flow we have chosen to apply a skin friction stress as wall layer model. That is, we append the Navier-Stokes equations (1.1) with the following boundary conditions:

$$\begin{aligned} &u \cdot n = 0, \\ &\beta u \cdot \tau_k + (\sigma n) \cdot \tau_k = 0, \quad k = 1, 2, \end{aligned}$$

for $(x,t) \in \Gamma_{\text{solid}} \times I$, with $n = n(x)$ an outward unit normal vector, and $\tau_k = \tau_k(x)$ orthogonal unit tangent vectors of the solid boundary Γ_{solid}. This boundary condition is an efficient and simple model with only one parameter β related to an estimate of the skin friction. The model takes the form of the simplest wall layer models developed already in the 1970s [39]. Furthermore it is also general, it can handle any type of geometry without any manual fine tuning, except the skin friction parameter.

3.2 cG(1)cG(1) as adaptive DNS/LES

Turbulent solutions fluctuate rapidly in wakes and boundary layers. These regions require a higher degree of resolution compared to the free stream region. But an increased resolution will also raise the computational cost. Therefore it is natural to increase the resolution only in these restricted regions. However, how should one choose which elements to refine?

We base the adaptive mesh refinement algorithm on the a posteriori error estimate for cG(1)cG(1) derived in [20]. For $\hat{u} = (u, p)$ a weak solution to the Navier-Stokes equations, and $\hat{\varphi} = (\varphi, \theta)$ a solution to a linearized dual problem with a mean value output $M(\hat{U}) = ((\hat{U}, \hat{\psi}))$, with $\hat{\psi}$ a weight function, we define an error estimate:

$$\begin{aligned} |M(\hat{u}) - M(\hat{U})| \leq \sum_{n=1}^{N} \Big[&\int_{I_n} \sum_{K \in \mathcal{T}_n} |R_1(\hat{U})|_K \cdot \omega_1 \, dt + \\ &\int_{I_n} \sum_{K \in \mathcal{T}_n} |R_2(U)|_K \, \omega_2 \, dt + \\ &\int_{I_n} \sum_{K \in \mathcal{T}_n} |SD_\delta^n(\hat{U}; \hat{\varphi})_K| \, dt \Big] \end{aligned} \tag{3.2}$$

for each element K in the mesh $\mathcal{T}_n$ where R_i are residuals of the Navier-Stokes equations and ω_i are stability weights in the form of powers of the local mesh size and derivatives of the dual solution $\hat{\varphi}$, for details see [20]. An adaptive DNS/LES algorithm can then be stated as in algorithm 3.1.

Algorithm 3.1: Adaptive DNS/LES

```
Given an initial coarse mesh $\mathcal{T}^0$, start at $k = 0$

    1. For the mesh $\mathcal{T}^k$: compute the primal problem and the dual problem.
    2. If the error estimate (3.2) is less than a given tolerance then stop, else:
    3. Mark a fraction of the elements with highest error for refinement.
    4. Generate the refined mesh $\mathcal{T}^{k+1}$, and goto 1.
```

Another key part for efficiency in the adaptive framework of cG(1)cG(1) is the ability to optimize the mesh according to some chosen flow quantity, represented by the data $M(\cdot)$ in the dual problem.

We now have all the components we need to derive an adaptive finite element solver. G2 with skin friction boundary conditions gives us the ability to simulate turbulent flows with a simple wall layer model, avoiding resolution of the boundary layer. A posteriori error estimation together with adaptivity is a powerful tool that we can use to derive an efficient solver, that can be used to compute a solution to a large scale problem, on workstations as well as on massively parallel supercomputers.

Chapter 4

An adaptive finite element solver

For the adaptive DNS/LES algorithm 3.1 we wish to construct an adaptive solver following the framework illustrated in Figure 4.1. The goal here is to run the adaptive iteration several times as illustrated in Figure 4.2.

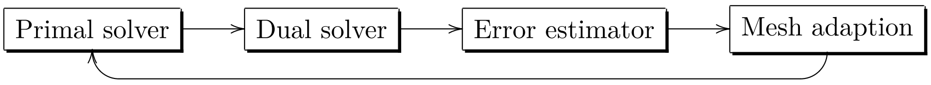


Figure 4.1: Overview of an adaptive finite element framework.

Therefore each component needs to run efficiently and in parallel, otherwise the framework would have a serial section which would restrict the potential speedup as shown by Amdahl's law (2.2). In this chapter we derive the foundation of a high performance adaptive finite element solver, discuss various parallelization strategies of the key components in Figure 4.1, and outline how these can be parallelized with a minimal serial fraction.

4.1 Finite element assembly

As for most problems the parallelization strategy is often restricted by the problem formulation. For a finite element solver this restriction comes from the assembly of the linear system, derived from the variational formulation.

For given bilinear and linear forms a, L and a function space V, we formulate the abstract problem as, find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in V \tag{4.1}$$

Given a certain discretization $\mathcal{T}$ of the domain Ω we make an ansatz for the discrete

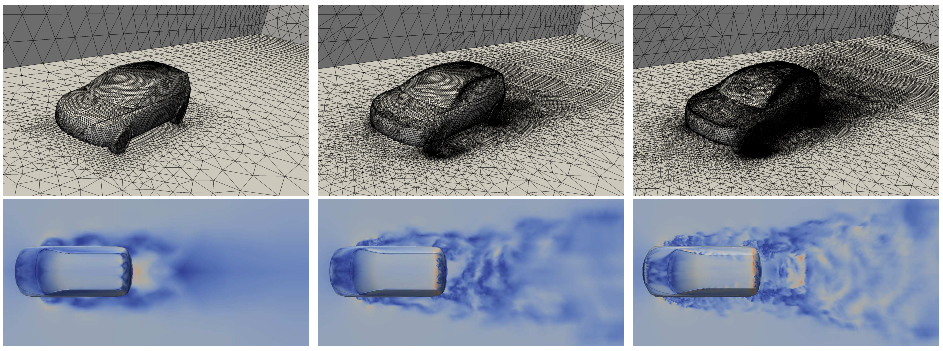


Figure 4.2: Solution and mesh at time t for different adaptive iterations.

solution U as:

$$U = \sum_{j=1}^{N} \xi_j \varphi_j \tag{4.2}$$

where $\xi_j = U(N_j)$ and N is the number of nodes N_j in our discretization, and φ_j are finite element basis functions. For simplicity, we here make the assumption that φ_j is a nodal basis for the finite element space $V_h \subset V$, and we identify the nodes N_j with the vertices of the mesh, corresponding e.g. to a piecewise linear approximation over a tetrahedral mesh as in the cG(1)cG(1) method. Substituting (4.2) in (4.1) we get the following analogy of a discrete abstract problem in V_h,

$$\sum_{j=1}^{N} \xi_j \, a(\varphi_j, \varphi_i) = L(\varphi_i), \quad i = 1, \ldots, N \tag{4.3}$$

which leads to the discrete system $A\xi = b$ where

$$\begin{aligned} A_{ij} &= a(\varphi_j, \varphi_i) \\ b_i &= L(\varphi_i) \end{aligned}$$

A can be constructed by adding the contribution from each element $K \in \mathcal{T}$. Where the element matrix $A_{ij}^K = a(\lambda_j, \lambda_i)$, with element basis functions λ_i, must be computed and A_{ij}^K added to the global matrix A. This leads to the formulation of the general assembly algorithm 4.1.

In algorithm 4.1 the elements are processed one at a time, computing the element matrices. Therefore, a natural parallelization would be to consider each element as one task and then distribute the tasks among all the PEs. But is this parallelization the most efficient? Recall from §2.3, the maximum speedup one can obtain is limited by the serial fraction of a computation. To answer the question one has to deduce how much of algorithm 4.1 needs to be executed in serial. Clearly each element

Algorithm 4.1: Finite element assembly

```
$\mathcal{I}$ (local index on element) = global index, $\mathcal{T}$ mesh
foreach element $K \in \mathcal{T}$ do
    construct local stiffness matrix $A^K$
    foreach $i \in K$ do
        foreach $j \in K$ do
            Add contribution to the global matrix $A$
            $A_{\mathcal{I}(i)\,\mathcal{I}(j)} = A_{\mathcal{I}(i)\,\mathcal{I}(j)} + A^K_{ij}$
        end
    end
end
```

can be computed independent of all the others, but the update of A can only be executed in serial due to the data dependency.

At first this might be seen as a rather pessimistic result for the potential speedup. However, assume the matrix has m rows, and each PE out of a total of P is assigned a fraction m/P of the rows. The update in algorithm 4.1 can then be executed in parallel if the row is owned by one and only one PE. For all non local updates, that means when vertices corresponding to i, j is not on the local PE, a serial synchronization point is needed. Hence, for good efficiency the number of non local updates has to be minimized. This can be achieved by a good initial data decomposition.

4.2 Data decomposition

In parallel computing the idea of data decomposition or static load balancing is simple, namely divide the job equally across all PEs. Poor load balancing can have a significant effect on performance when several PEs are sitting idle instead of doing useful work. In the finite element setting there is a close relationship between work and the number of elements assigned to a PE.

Furthermore, for distributed memory machines as in most high performance architectures used today one also has to consider the communication. Even for a fairly evenly divided workload, a data decomposition that does not take into account the communication cost can have a significant effect on performance. We now leave the communication issue for a moment and focus solely on the data decomposition problem, which can be formulated as the partitioning problem.

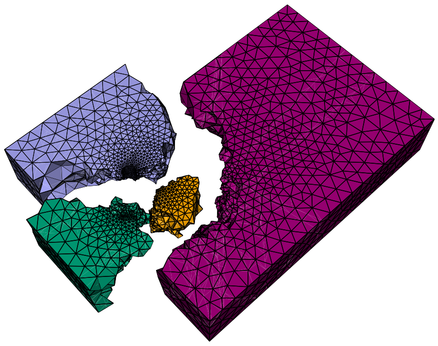


Figure 4.3: An example of data decomposition on 4 PEs.

4.2.1 The partitioning problem

Given a set of elements $\mathcal{C}$ from a mesh $\mathcal{T}$, the partitioning problem for p workers can be expressed as to find p subsets $\{\mathcal{T}^i\}_{i=1}^p$ such that:

$$\mathcal{T} = \bigcup_{i=1}^{p} \mathcal{T}^i \quad \text{and} \quad \mathcal{T}^i \cap \mathcal{T}^j = \emptyset \,, \quad i \neq j \tag{4.4}$$

with the constraint that the workload $W(\mathcal{T}^i) = |\{\mathcal{C} \in \mathcal{T} \mid \mathcal{C} \in \mathcal{T}^i\}|$ should be equal for all subsets. As mentioned earlier for finite element computation a key for good performance is to minimize data dependencies between PEs. If we consider an element to be a set of vertices and edges, it is natural to model data dependencies as the edges between vertices. In order to minimize the data dependency and communication the number of edges cut between partitions must be minimized. This is then natural to formulate as a graph partitioning problem.

4.2.2 Graph partitioning

Given a weighted undirected graph $G = (V, E)$, with nodes V and edges E, the k-way partitioning problem is to split V into k subsets $\{S_j\}_{j=1}^k$ with the constraint that the sum of the weights should be roughly equal in each subset and the number of edges cut should be minimized. The problem is known to be NP-complete, but there exist several good approximation algorithms.

The least expensive *geometric* methods are solely based on the geometry of the graph. For example linearize the graph with a space filling curve and try to divide V into k equally sized subsets [31]. The method is fast, but doesn't take into consideration the topology of the graph, hence it does not minimize the number of edges that are cut [45].

Spectral methods approximate the NP-complete problem into a pure linear algebra problem. These methods partitions the graph based on the eigenvector corresponding to the second smallest eigenvalue of the graph's Laplacian matrix L, defined as:

$$l_{i,j} = \begin{cases} -1 & \text{if}\, i \neq j \,\text{and}\, (v_i, v_j) \in E \\ |D_{v_i}| & \text{if}\, i = j \,\text{where}\, D_{v_i} = \{e \in E \,|\, v_i \in e\} \\ 0 & \text{otherwise} \end{cases}$$

This eigenvector, called the Fiedler vector, only contains values ± 1, hence it can be used to divide a graph in half, forming a base for a recursive partitioning technique [5]. However, computing eigenvalues from L quickly becomes too expensive for larger graphs [26].

The current state of the art techniques in graph partitioning are multilevel methods. Based on the idea of divide and conquer, a multilevel method computes an approximation of the graph by contracting neighboring nodes. On the contracted graph the new partitions are computed and the result is projected onto the full graph [26]. The method is moderately cheap, it all depends on which method that is used on the coarse approximation. A key feature here is that one can afford to use an expensive method on the coarse approximation, one that also tries to minimize the number of edge cuts.

4.2.3 Mesh partitioning

Graph partitioning extends naturally to mesh partitioning. However, choosing the correct graph representation is crucial for a good parallel performance. The naive way is to directly interpret the mesh as a graph, with the vertices as the graph nodes V and the element edges as the graphs edges E. Using the naive representation the finite element solver does not have any control over how the elements are distributed. Elements might also be split into several partitions since the partition function is defined on vertices.

In the finite element assembly algorithm 4.1, the elements are processed one at a time. For each element, the local tensor is computed and its contribution is added to the global tensor. As mentioned earlier a natural way to parallelize this operation is to assign different elements to different PEs. However if the elements are split across different PEs, the question is to which PE an element belongs? And which PE should perform the assembly? A better solution is to assign whole elements instead. This can be achieved if we instead partition the dual graph of the mesh. Recall that the dual graph is the graph where the nodes are the mesh elements, and edges are placed between nodes if two elements share a common facet.

Once partitioned, the mesh needs to be represented in some way on all the PEs. The naive way is to store the entire mesh on each PE. The partition algorithm is then used for static load balancing, assigning elements to PEs for assembly. This representation would of course lower the assembly time but due to the fixed memory footprint it would not scale well for larger problems. A more natural and memory conservative way is to use a distributed mesh representation. The entire mesh is divided into smaller parts by the partitioning algorithm. Each PE will then have a smaller part of the entire mesh, and the question is how to handle the overlap of possibly shared entities. From algorithm 4.1 it is clear that a PE needs to be able to access all entities of an element in order to compute the element stiffness matrix. Hence, when a mesh is divided into smaller partitions we introduce an important data dependency between PEs. How to represent this overlap is crucial in order to achieve good scaling. If we take a too memory conservative approach, and let each partition be a disjoint subset of the mesh, each PE would need to communicate with its neighbour for each element it assembles on the interface between PEs. Instead we represent the overlap with ghost entities, replicating all the needed data on each PE. It requires more memory, but it will drastically reduce communication during assembly.

4.3 Mesh refinement

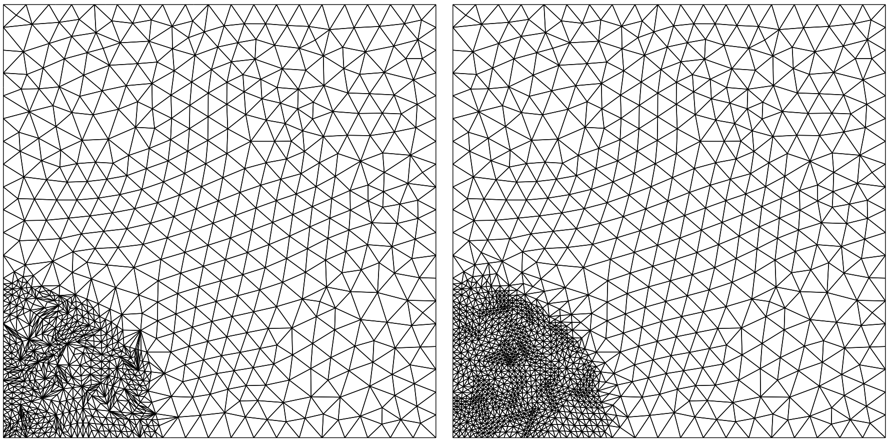
In order to harvest the powerful potential of the adaptive cG(1)cG(1) method described in §3.2, a solver needs an efficient mesh adaption routine. For this the traditional way is to rely on local mesh refinement, in particular edge bisection.

The general idea is to bisect elements into new elements, with the constraint that the mesh must be free of *hanging nodes* for the mesh to be conforming, that means no vertices should be on another element's facet or edge. A common method is to bisect all edges in the element, creating several new elements for each bisection (see for example [7]). Other methods take a more conservative approach. Instead of bisecting all edges in an element, they select one edge that would produce two new elements with the best quality possible. For example, compare the mesh quality for the simple bisection in Figure 4.4a with the longest edge bisection algorithm in Figure 4.4b. Both methods bisect elements across one edge but use two completely different approaches in selecting the most appropriate candidate.

The simple edge bisection method bisects elements in pairs across one edge (see for example Paper I and [22]). Another method bisects the edge opposite to its newest vertex. This method is often referred to as the newest vertex approach, described in [4]. Both of these methods have their drawbacks. The simple methods produce meshes with inadequate quality and the newest vertex approach is rather complex to implement with many special cases to consider, something that one tries to avoid for code that has to perform on thousands of PEs. A more elegant and simpler method is the recursive longest edge algorithm [33], where one always selects the longest edge for refinement and recursively continues to bisect elements until the mesh is free of *hanging nodes*.

4.3.1 Parallel mesh refinement

Mesh refinement in parallel introduces an additional layer of complexity in the process. In the parallel setting, due to the data decomposition each PE has a



(a) Simple edge bisection. (b) Recursive longest edge.

Figure 4.4: An illustration of how drastically mesh quality can differ between different mesh refinement methods.

small part of the global mesh. If a new vertex is created on the shared boundary between PEs the refinement must ensure that the information propagates onto all the neighboring PEs and that the *hanging node* is removed.

A positive effect of the decomposition is that one divides the refinement processes into two distinct stages. First a bisection stage and secondly a clean up stage where *hanging nodes* are removed. Furthermore, it is easy to realize that the first stage is a local operation without any communication, hence perfect for deriving a parallel algorithm with a natural synchronization point, algorithm 4.2.

Algorithm 4.2: General parallel refinement algorithm

```
Let $\mathcal{R}$ be the set of elements marked for refinement, $\mathcal{T}$ be the initial mesh,
partitioned into $k$ subsets $\mathcal{T}^i$

    1. Bisect all elements from $\mathcal{R} \in \mathcal{T}^i$

    2. Propagate hanging nodes to neighboring PEs

    3. Mark elements with hanging nodes for bisection

    4. If $\mathcal{R} \in \emptyset$ end, else goto 1.
```

The complexity of algorithm 4.2 can change significantly depending on which bisection method is used. For example consider the simple bisection method. Since the method bisects elements in pairs, after step one there will be several hanging nodes on the shared boundary between processors. But after the propagation step all these nodes will be part of newly bisected elements on the receiving PE. Thus, the shared boundary will be free of hanging nodes and algorithm 4.2 will terminate after one exchange step.

Consider instead a refinement algorithm which does not bisect element in pairs. After the first step the situation will be the same as for the simple bisection with several hanging nodes on the shared boundary. However when the algorithm wraps around on the next bisection round things will be different. Now it is not guaranteed that the element will be bisected across the same edge as the hanging node. Hence, in each iteration additional hanging nodes can be produced and it is not clear that the algorithm will terminate. To make things even worse, in the parallel setting each PE can arbitrarily change state from being active, refining, or to being idle with no elements to bisect. So a condition for termination of the algorithm is that all PEs are idle at the same time. It can therefore be reduced to a global termination detection problem.

4.3.2 Distributed termination detection

An efficient termination detection method is needed in order to realize the general refinement algorithm 4.2. A naive approach is the ring detection algorithm [47] where PEs are grouped in a virtual ring topology, see Figure 4.5. One assigned master PE initiates termination detection by passing a vote for termination to one of its neighbours. The neighbour then also votes for termination or no termination and passes along the vote to the next PE. If the vote returns positive to the initiator the algorithm has terminated. Ring termination is straightforward to implement but since it is a serial communication pattern it will have a negative impact on the parallel efficiency.

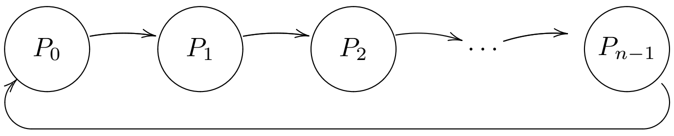


Figure 4.5: Ring termination detection algorithm.

A more efficient method is Dijkstra's general distributed termination algorithm [12, 6]. Once again one of the PEs is assigned to be the master. But instead of initiating termination detection by passing along a token, it detects termination by counting messages. The master PE will initiate a task by sending *work* messages to all the other PEs. Depending on the task to solve, the receiving PE can also send more *work* message to other PEs, constructing an activity graph illustrated in Figure 4.6. When a PE finishes its task it sends back a *done* message to the PE from which it got its *work* message. The master PE then signals for termination when it has received a *done* message for all the *work* messages sent. For example in refinement, each processor would send a *work* message to the PEs to which refinement has propagated.

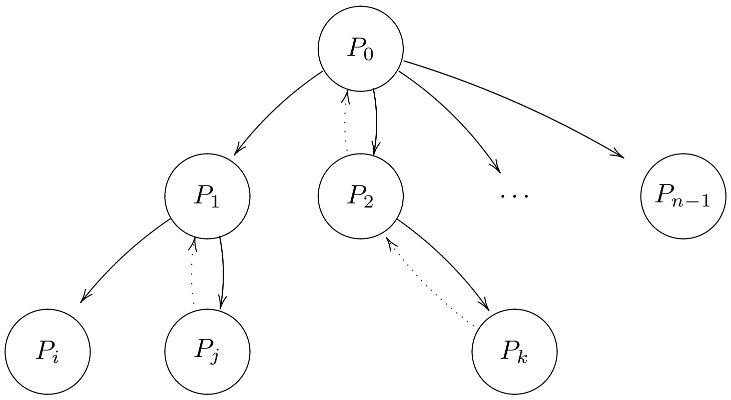


Figure 4.6: An illustration of the activity graph during mesh refinement. Solid lines (−) represent *work* messages, and dotted lines (· · ·) represent *done* messages.

Using Dijkstra's algorithm in algorithm 4.2, termination detection is more distributed and hence more efficient than ring detection. However, since global termination only can be decided by the master PE, the algorithm might suffer from a high message contention when a large number of PEs send *done* messages to the master. A more efficient termination detection strategy for parallel mesh refinement is to include termination detection in the propagation step. This can be realized by exchanging messages between PEs in a hypercube pattern (Figure 4.7). With such a communication pattern, termination can be detected in a truly distributed fashion without any central controlling PE. In Paper I we present our hypercube based distributed termination detection algorithm.

4.4 Dynamic load balancing

A key component in any parallel computation is the balance of computational work between all the workers. Data decomposition using graph partitioning gives a good initial static load balancing, but if a mesh is refined the workload balance is changed. Therefore any adaptive parallel computation needs dynamic load balancing, that means given a workload W a program should be able to adjust the workload distribution while it is running.

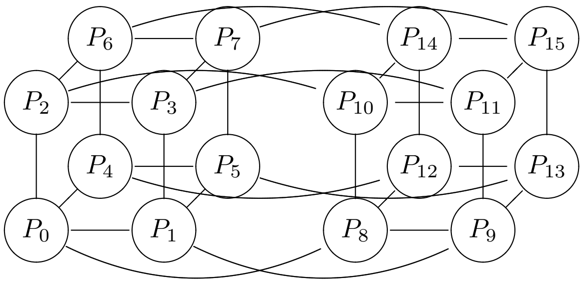


Figure 4.7: An illustration of the 4D hypercube communication pattern.

We model the workload by a weighted dual graph of the finite element mesh. Let $G = (V, E)$ be the dual graph of the mesh, q be one of the partitions and let w_i be the computational work (weights) assigned to each graph vertex. The processor workload is then defined as:

$$W(q) = \sum_{w_i \in q} w_i$$

where communication costs are neglected. Let W_{avg} be the average workload and W_{max} be the maximum, then the graph is considered imbalanced if:

$$W_{\text{max}}/W_{\text{avg}} > \kappa$$

where κ is the threshold value, often determined depending on the current problem and/or machine characteristics.

4.4.1 Diffusive schemes

In diffusive methods, a heavily loaded PE's vertices would move to another PE and in that way smear out the imbalance, described e.g. in [21, 37]. For example, consider the boundary vertices between two partitions $\mathcal{T}^i$ and $\mathcal{T}^j$. If $\mathcal{T}^j$ has fewer vertices, boundary vertices are moved from $\mathcal{T}^i$ to $\mathcal{T}^j$.

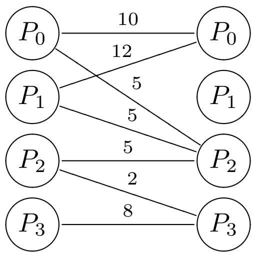
A drawback of diffusive schemes is that they are limited to a local view of the imbalance. When a diffusive scheme computes a new load balance, it does not consider the global imbalance it only consider the imbalance between adjacent partitions. Therefore if the imbalance is concentrated in a local region of the mesh, diffusive schemes can perform poorly by only smearing out the imbalance between adjacent partitions. Propagation to regions far away takes long, a characteristic of diffusion processes.

4.4.2 Intelligent remapping

In order to consider the load imbalance problem globally one has to approach the problem in a different way. Consider the graph partitioning problem where we try to divide a graph into equal sized subsets. We know that these methods produce well balanced partitions, and we can add weights to the dual graph to model workload. But how to add this constraint to the dual graph such that the data movement is minimized? This is a fairly difficult modelling problem that can be avoided.

Assume that a graph partitioner has produced new partitions $\mathcal{T}'$ from an already partitioned mesh $\mathcal{T}$. If the new partitions are assigned in such way that a minimal amount of data is moved from the original partitions we consider the load balancing problem solved. The idea behind intelligent remapping schemes, is to assign the new partitions to workers in an optimal way. Since this method uses a graph partitioning method it has a global view of the imbalance problem, and performs better than diffusive schemes if the imbalance appears in local regions of the mesh [38].

Start with an imbalanced workload. Repartition the dual graph with a graph partitioner and place the result in a matrix S, where each entry S_{ij} is the number of vertices on PE i which would be placed in the new partition j. The goal is to keep as much data local as possible, hence to keep the maximum row entry in S local. If the matrix is transformed into a bipartite graph (Figure 4.8) where each edge e_{ij} is weighted by S_{ij}, the problem can be reduced to the maximally weighted bipartite graph problem [30].



$$S = \begin{pmatrix} 10 & 0 & 5 & 0 \\ 12 & 0 & 5 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix}$$

Figure 4.8: Example of a weighted bipartite graph and its corresponding S matrix.

Solving this graph problem is known to be expensive $O(V^2 \log(V) + VE)$, and since the number of nodes V in a graph with P PEs is $O(P)$ it quickly becomes expensive if a large number of PEs is used. In [30] it was shown that the graph problem can be solved in $O(E)$ time using a heuristic algorithm. Each row in the matrix S is placed in a sorted list, a greedy algorithm is then used to pick the largest value which belongs to an unassigned partition. In the worst case all PEs need to send data to all the other PEs, hence $E \sim P^2$. Therefore, even the linear heuristic becomes too expensive to solve as the number of PEs grows. In Paper I we improved the heuristic algorithm by performing the sorting in parallel using a binary radix sort with a runtime of $O(P)$.

4.4.3 A priori workload estimation

As we have seen the imbalance problem caused by mesh refinement can be efficiently solved. But at what cost? Imagine if all refined elements are located at one PE. In that case load balancing will be fairly expensive due to the excessive data movement that is needed. But this is not the only problem. How efficient is the mesh refinement if executed by only one PE out of thousands? What is needed is something that will load balance the load balancer and mesh refinement.

Remember that the workload was modelled by a weighted dual graph of the mesh. So far we have not discussed how the graph should be weighted. Since the entire data decomposition is based around balancing the work of the assembly processes a natural choice of weights is the amount of “assembly work” each element contains. There are of course numerous ways to define this, due to different boundary conditions, connectivity and such, but a good and simple approximation is one element equals one unit of work.

Now consider the case of an element that is marked for refinement. We know that after the bisection the element’s unit of work will at least be two (since it will at least be bisected once). If we weight the dual graph according to the workload after a refinement and load balance before the refinement, then a more balanced number of elements will be bisected on each PE and less data will be moved [30]. In Paper I we present our a priori predictive dynamic load balancer. The method is based on a priori workload estimation. A dry run of the refinement processes is used to compute the workload, which is then used as input data for the load balancer. Once the estimated workload has been evened out the refinement process starts to bisect elements.

Chapter 5

Scalable linear algebra

One of the key components in a finite element solver is the linear algebra kernels. In order to scale to thousands of PEs a high performance solver must have efficient and scalable linear solvers in order to solve the seemingly trivial problem,

$$A\xi = b \tag{5.1}$$

which arises from the abstract discrete problem (4.3). In practice, solving (5.1) is far from trivial in serial and the parallel setting does not make the situation any easier. The matrix A in (5.1) is in general sparse which adds additional constraint on both solvers and matrix representation.

5.1 Sparse matrices

A matrix where most of the entries are zero is defined to be sparse. To avoid saving redundant information, special data structures are used to represent only the non–zero entries of the matrix. These compressed representations have different costs and trade-off associated with them. Since a FEM solver can give rise to large sparse matrices that may need to be assembled and solved in every time step, it is important that the format is chosen with care.

5.1.1 Storage formats

Sparse matrices can be represented in numerous formats, depending on the application in mind. For scientific computing the Compressed Row Storage (CRS) is the most common for general usage. In CRS the non–zero entries of a matrix are encoded by three arrays, $value, col$ and row. The $value$ array stores the non–zero elements of A, $col(k)$ holds the column indices for the entry in $value(k)$ and the row array keep tracks of where different rows begin and ends in the other two arrays. For example, let $value(k) = a_{i,j}$ be the non-zero matrix entry at position (i, j) then $col(k) = j$ and $row(i) \leq k < row(i + 1)$. Hence in order to access an element one has to use the *row* array to find the offset into the *value* array. For instance, the matrix

$$S = \begin{pmatrix} 10 & 2 & 5 & 0 \\ 12 & 0 & 5 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 0 & 0 & 8 \end{pmatrix} \tag{5.2}$$

would be encoded as illustrated in Figure 5.1.

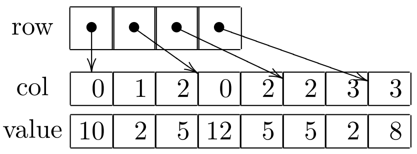


Figure 5.1: An illustration of the Compressed Row Storage representation of (5.2).

Since each row is stored in consecutive memory locations, the CRS format is very efficient for problems with a regular access pattern such as matrix vector multiplication, but less efficient for random access operations such as inserting entries at random locations. As presented in chapter 4, a frequent operation in our solver is the finite element assembly. In algorithm 4.1 the elements are processed one at a time, and its contributions are added to the matrix. Since the local-to-global mapping function $\mathcal{I}$ can map an element's local index to an arbitrary global index, the access pattern with which the indices are inserted are highly likely to be irregular.

For static encoding schemes such as CRS, if an element needs to be inserted in the middle of the *value* array it will lead to a costly reallocation and data movement due to the growth of the entire array. Therefore it is crucial that one preallocates enough space for CRS like formats. However, it is often difficult to know well in advance how many entries that are going to be needed. Hence, naive insertion into a matrix can be costly.

To overcome this problem most implementations, for example MTL4 [15], preallocate s entries for each row in the CRS data structure [14], enabling efficient on the fly insertion, given an adequate preallocation factor. Another strategy is to use an intermediate format for assembly and then convert it into CRS. In [2] the sparse matrix is represented as a long hash-table. The implementation shows impressive performance number but since hash-tables have poor memory locality it has to be converted into CRS before any Sparse Matrix Vector Multiplication (SpMV) operation can be performed. A conversion that can be inherently costly. The general vector-of-vector format in uBLAS [46] stores the values as a vector of sparse vectors, but its insertion rate is also poor [14] since it needs to be converted to CRS before it can be used efficiently. Instead we propose a new way of using a stack-based representation. It is similar to the linked-list data structure where each row in the matrix is represented by a linked list.

5.1.2 Stack-based representation

A stack-based representation of a sparse matrix is based around a long array A, with the same length as the number of rows in the matrix. For each entry in the array $A(i)$ we have a stack $A(i).rs$ holding tuples (c, v) representing the column index c and element value v, hereby referred to as a row-stack, illustrated in Figure 5.2. Inserting an element into the matrix is now straightforward. Namely, find the

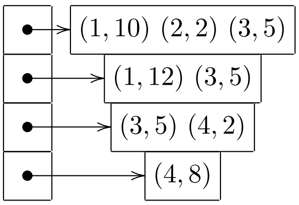


Figure 5.2: An illustration of the stack-based representation of the matrix S (5.2).

corresponding row-stack and push the new (c, v) tuple on the top. Matrix updates, such as adding a value to an already inserted element, is also straightforward: Find the corresponding row-stack, perform a linear search until the correct (c, v) tuple is found and add the value to v, as illustrated in algorithm 5.1.

Algorithm 5.1: Pseudo-code for matrix update ($A_{i,c} += v$).

```
for $j = 1 : length(A(i).rs)$ do
    if $A(i).rs(j).c == c$ then
        $A(i).rs(j).v += v$ ;
        return;
    end
end
push $(c, v)$ onto the row-stack $A(i).rs$
```

Compared to CRS the stack based format removes the indirect addressing needed to find the start of a row, and has in general a lower reallocation cost since each stack can be grown independently. Also these stacks do not need to be ordered. Thus we could push new elements regardless of the column index. In the case of a matrix update, the linear search will still be efficient since each stack has a short length equal to the number of non-zeros in the corresponding row.

However, the representation's SpMV performance is lower than CRS mostly due to the lack of locality when iterating over all the rows. Therefore, for problems with a static sparsity pattern we rearrange the row-stacks such that they are placed in consecutive memory locations. Hence, we convert the format into something similar to CRS.

Since the compressed row storage has become more or less the de facto standard for general scientific computing, we pause for a while and present serial performance numbers for different storage formats, before we continue with a discussion on the issues with parallelizing sparse matrices efficiently.

We benchmarked the stack-based representation for two different scenarios, ascending row order and fully random insertion, and compare it to two state of the art implementations, PETSc [3] and MTL4 [15]. Finally we measured the SpMV GFlop/s rate for all implementations. For all insertion benchmarks we let A be an $N \times N$ matrix with $nz = 64$ preallocated nonzeros per row. We have repeated each experiment several times to get an accurate measurement, and also varied the matrix size as $N = 10^i$ for $i = 2, \dots, m$ (with an m depending on the experiment). The experiments were tested on four different architectures, using four different compilers. In all the experiments JANPACK refers to a library implementing the stack-based representation. For more information about the library please refer to §6.2.

Ascending row order insertion

In the first benchmark entries are inserted in an ascending row order. For each row the column index of the non-zero value is random. This should correspond to an almost perfect insertion, except that formats like CRS need to sort the inserted entries. In Figure 5.3 we see that the stack-based format performs well compared to the other state of the art implementations, achieving almost a factor of two times faster on all architectures, despite the additional conversion step.

Fully random insertion

In the second benchmark entries are inserted in a fully random fashion, both row and column index are for each element chosen at random:

$$A(mod(rand(),N)\ ,\ mod(rand(),N)) \mathrel{+}= value$$

The fully random benchmark is the most difficult one since rows are visited completely random and there is no guarantee that there will only be nz elements per row. From the results in Figure 5.4 we see how challenging this benchmark is for static sparsity formats, due to the many needed reallocations and costly data movement, while the dynamic stack-based format handles these issues with ease.

Sparse matrix vector multiplication

Good assembly performance is of course important for a FEM solver, but it is not everything. As we will see later in §5.2 good SpMV performance is as important. Therefore as a final serial benchmark, several matrix vector products were computed for a set of test matrices [48], representing a wide range of applications. All matrices were obtained from the university of Florida sparse matrix collection [9]. In Figure 5.5 we see that the stack-based representation performs on a par with other state of

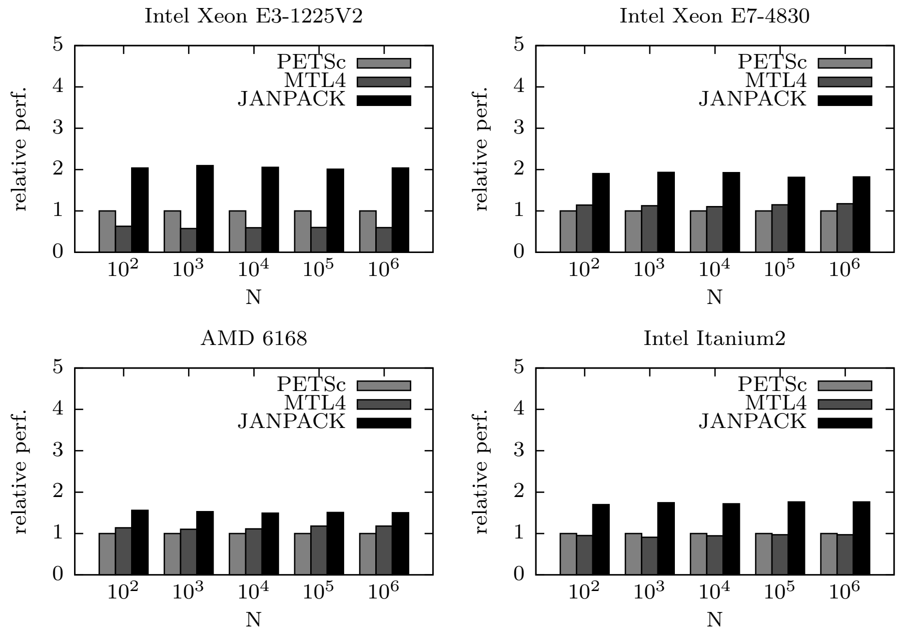


Figure 5.3: Ascending row order insertion, relative performance against PETSc.

the art implementations. These good results substantiate the stack-based format's suitability for large scale scientific computing.

5.1.3 Parallel representation

With an efficient serial sparse matrix representation the question is how to also make it efficient for thousands of PEs. As with all kinds of parallelizations the fundamental question is data decomposition: How should the matrix be distributed between the PEs in order to perform well in all parts of an FEM solver?

A natural decomposition distributes the matrix either by columns or rows. Using a column based decomposition, the matrix vector product (with a row distributed vector) will perform well since all necessary data is local. Considering matrix assembly the performance will however be poor. Recall from §4.1 that each element corresponds to a row in the matrix, hence with a column based distribution there will be more communication needed during FEM assembly.

For our finite element needs a row based distribution, where the first $0, \ldots, n$ rows are assigned to the first PE, the $(n + 1), \ldots, m$ to the next and so forth, it is much better suited for our needs. During assembly larger part of the elements will now be local. The only downside is the need to fetch dependencies in the vector during matrix vector multiplication. However, if the degrees of freedom are

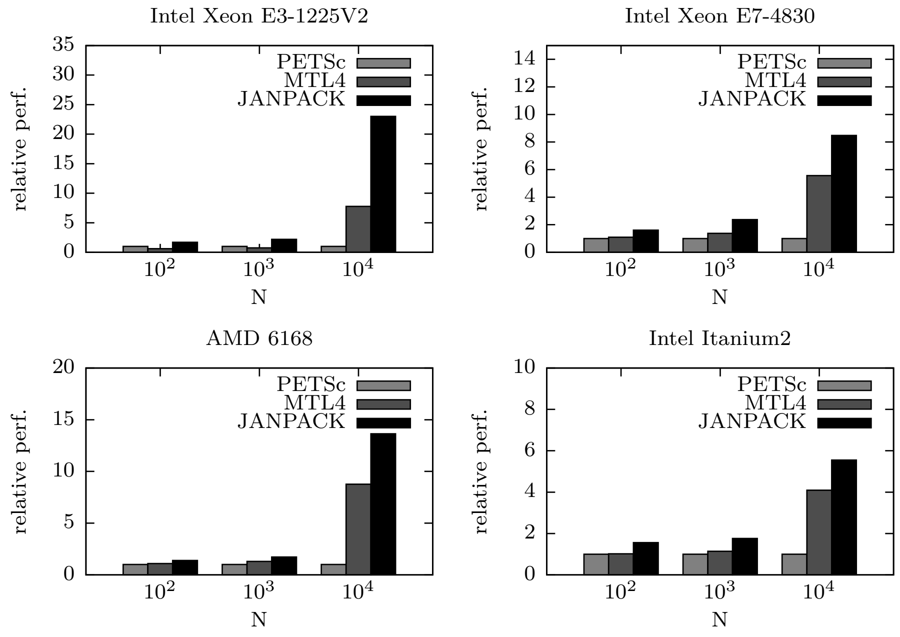


Figure 5.4: Fully random insertion, relative performance against PETSc.

renumbered in a certain way, the amount of dependencies can be reduced.

A good data decomposition is not everything. To scale for example sparse matrix assembly, with its random access pattern on thousands of PEs, the code needs to be implemented using a parallel programming model which allows for low latency point to point communication to support the required fine grained parallelism. Since the communication pattern is also highly unstructured, the two-sided communication abstraction of traditional message passing will enforce certain synchronization points which introduce latency and slow down the code.

To overcome this problem, one can use the one-sided communication abstraction of the PGAS programming model (discussed in §2.2). With one-sided communication and a matrix placed in global memory, each PE can fetch or update any matrix element without needing to explicitly inform the owner of the matrix element. In other words, PGAS provides an ideal sparse matrix implementation for large scale FEM assembly.

5.2 Linear solvers

The efficient parallel sparse matrix representation solves only half the problem. We also need to solve the linear systems. Now the situation is slightly more complex

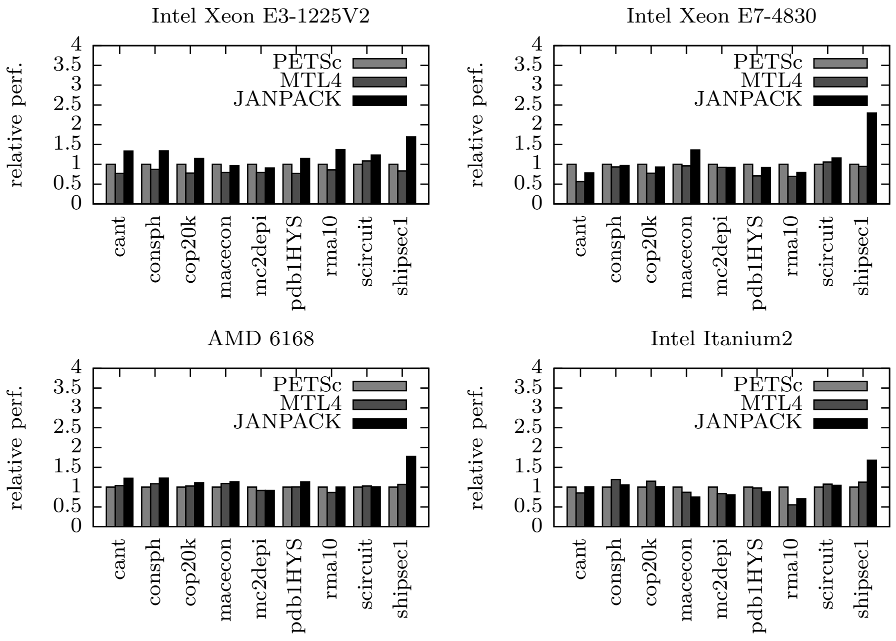


Figure 5.5: Sparse matrix vector multiplication performance for the benchmark suite [48], relative performance against PETSc.

than before. Several years of numerical linear algebra research have produce excellent serial solvers, but far from all of them are applicable in the parallel setting.

The solvers fall into different categories. The *direct* methods are derived from Gaussian elimination and the *iterative* methods are based on generating sequences of improving approximate solutions, e.g. Krylov subspace methods. Finally, *multilevel* methods are based on a hierarchy of different problem sizes, where the iterations damp different error frequencies. Despite being exact, the direct methods suffer from the problem of fill–in during the elimination process. This can lead to large memory consumption and increased solution cost, see for example [42] for compelling argument against direct methods for large problems. Also, based on Gaussian elimination the cost of solving a general $m \times m$ matrix is $\mathcal{O}(m^3)$, which for a large m quickly becomes too expensive.

Therefore we will focus our work on iterative and multilevel methods in the form of Krylov and algebraic multigrid solvers. Both have proved excellent in serial, but they require some extra work in order to be used efficiently in parallel.

5.2.1 Iterative methods

For an iterative method we seek a sequence of vectors, $\xi_1, \xi_2, \dots, \xi_n, \dots, \rightarrow \xi^*$, where $A\xi^* = b$ solves (5.1). Krylov subspace methods seek for this vector in the Krylov subspace $\mathcal{K}$ spanned by the vectors obtained by successively multiplying a starting vector ξ_0 with the matrix A,

$$\mathcal{K}_k = (A, \xi_0) = \{\xi_0, A\xi_0, \dots, A^{k-1}\xi_0\}. \tag{5.3}$$

For a more complete overview of Krylov subspace methods, see [35, 42]. Based on matrix vector multiplication the operation count for solving a $m \times m$ system goes down to $\mathcal{O}(m^2)$. Since the matrices are sparse, let us assume that each row has on an average of nz non-zero entries, the cost can be reduced further down to $\mathcal{O}(nz \cdot m)$. This makes these methods more competitive than direct methods for large systems. One of the biggest challenges in parallel is to have an efficient sparse matrix vector multiplication kernel and a Krylov method suitable for parallelization.

In our setting, the corresponding linear system of (3.1) is most often nonsymmetric. One popular Krylov method for general nonsymmetric systems is the Generalized Minimal Residual method (GMRES) [36], which generates an orthonormal basis for the Krylov subspace, by successively orthogonalizing the basis. This is costly and consumes a large amount of memory since the entire basis has to be stored. One way to avoid this is to restart the method after k steps (only storing k basis vectors at a time), which leads to the GMRES(k) method. The problem now lies in choosing a proper value of k, such that poor convergence is avoided.

Another way around this problem is to instead use methods based on two biorthogonal subspaces,

$$\mathcal{K}_k = (A, v) = \{v, Av, \dots, A^{k-1}v\}$$

and

$$\mathcal{L}_k = (A^T, w) = \{v, A^T w, \dots, (A^T)^{k-1}w\}.$$

A popular method in this class of Krylov subspace iterations is the Bi-CGSTAB method [41], which avoids computing the transpose of A in one of the subspaces. Since the method has to construct two subspaces, Bi-CGSTAB requires two matrix vector products but does not require the additional storage and costly orthogonalization of the basis vectors. Since matrix vector products are relatively inexpensive, Bi-CGSTAB is a good candidate for a scalable iterative solver.

5.2.2 Preconditioners

The convergence rate of Krylov subspace methods depends on the distribution of A's eigenvalues. Often it is possible to accelerate the convergence by applying a preconditioner M. This solves the transformed system $M^{-1}A\xi = M^{-1}b$, which has the same solution as (5.1), with M chosen to make the spectral properties of $M^{-1}A$ more favorable. Describing the entire field of preconditioners is out of the scope of this work (for an overview see [35]), instead we focus on the properties which make them suitable in parallel.

One of the simplest preconditioners is the Jacobi preconditioner, defined as,

$$m_{ij} = \begin{cases} a_{ij} & \text{if}\, i = j \\ 0 & \text{otherwise.} \end{cases}$$

since it is a diagonal matrix, its inverse can easily be computed. The Jacobi precon-ditoner may not be the best in terms of convergence rate, but due to its inexpensive applicability (a single matrix-vector product) and excellent scalability it often performs better than more complicated preconditioners in time to solution, albeit with many more Krylov iterations. Unfortunately, not all systems converge with a Jacobi preconditioner. To make these systems converge we need to use more complicated preconditioners, but we don't want to sacrifice scalability. Remember that the preconditioner needs to be applied in every Krylov iteration, hence, if the forming of M incurs large amount of communication, scalability will be lost.

In these kinds of situations one often resorts to block Jacobi methods, where the variables are partitioned into mutually disjoint sets. With the goal of avoiding communication these sets can be chosen as the matrix entries that a PE owns, resulting in the preconditioner M defined as,

$$m_{ij} = \begin{cases} a_{ij} & \text{if}\, i \,\text{and}\, j \,\text{are owned by the same PE} \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

For a row distributed matrix, this results in a preconditioner with square blocks on the diagonal. Since these blocks are disjoint, we can compute a more expensive preconditioner in each block, without communication. Such expensive preconditioner are the ones based on an incomplete factorization of M. A regular LU factorization of M will have the same problem as discussed for direct methods (see §5.2). To avoid excessive amount of fill in, we limit the amount of new matrix entries to n, and get the incomplete factorization ILU(n) with a factorization level of n. Often, n is chosen to be zero such that the sparsity pattern of M is a subset of that of A, hence reducing memory requirements. Finally if the factorization step tends to be too expensive one can use an even simpler version called D-ILU [32]. In this diagonal ILU we alter only the diagonal elements, let $A = D_A + L_A + U_A$ the preconditioner becomes $M = (D + L_A)D^{-1}(D + U_A)$, where D contains the pivots generated by the factorization and L_A, U_A is the lower and upper part of A. In parallel these are the local sub blocks defined by (5.4). Despite its simple definition, the D-ILU preconditioner performs excellently in parallel, foremost since the preconditioner's factorization and solve step is inexpensive and can be performed in parallel, without communication.

5.2.3 Multilevel methods

Multigrid, or more general multilevel, algorithms are fast and efficient methods for solving linear systems. The idea is to eliminate "smooth errors" by solving the residual equation $Ae = r$ on a coarser representation of the problem, and interpolate back the result in order to correct the solution of the real problem. This coarse grid correction can of course be performed using several layers of coarser representations, which results in the classic multigrid cycling algorithms, see e.g. [8]. For certain problems it can also be proved that the execution time is asymptotically optimal [8, 34].

There exist two different multilevel approaches, *geometric* and *algebraic*. Geometric multigrid reduces the different error frequencies on a set of fine and coarse grids constructed based on the geometry of the problem. For structured grids choosing an interpolation between the different grids that reduces the error can be formulated easily. Often a simple linear interpolation is sufficient [8]. However, for the unstructured grids we are aiming for how should the different grids be selected in order to minimize the different error frequencies. In the algebraic multigrid method (AMG) [34] one forgets about the underlying discretization and only considers the matrix entries a_{ij} when selecting the different ("grid") levels A^k.

The coarsening process is based on a heuristic that smooth errors varies differently depending on how strongly coefficient in the matrix depends on each other. But how strong is "strongly"? One measure we can use is to say that an unknown i strongly depends on j if,

$$-a_{ij} \geq \theta \max_{k \neq i}\{-a_{ik}\} \tag{5.5}$$

given a threshold $0 < \theta \leq 1$. Using the strength measure (5.5) the problems unknowns (denoted by Ω) are partitioned into two disjoint sets $\Omega = C \cup F$, where C is the set of coarse points and F is the set of fine points which will be interpolated by the C points. In order to derive a partitioning that leads to a good interpolation formula, classic AMG coarsening [34] uses two criteria for choosing C and F points.

> **C1:** For each point j that strongly influences an F–point i, j is either a C–point or it strongly depends on a C–point k that also strongly influences i.
>
> **C2:** C should be a maximal subset of all points with the property that no two C points are strongly connected to each other.

where C1 relates to the quality of the interpolation between levels and C2 limits the size of the coarser level.

Using the two disjoint sets we can then define the interpolation and restriction operators I^k_{k+1} and I^{k+1}_k needed to transfer A between level k and $k+1$ in a Galerkin type projection,

$$A^{k+1} = I^{k+1}_k A^k I^k_{k+1}.$$

Most of the components in an AMG solver parallelize easily, for example the solve and interpolation step are composed mainly of matrix-matrix and matrix-vector products. A more difficult component is the coarse grid selection, which in its classical form is of a fundamentally sequentially nature. We refer to the classical coarsening algorithm as Ruge-Stüben (RS) [34], which selects C and F points using a two pass algorithm. The main problem when parallelizing RS is how the algorithm sequentially tests all points in order to make sure that they adhere to C1 and C2. Various parallelizations of RS has been proposed [16]. Most of them are based on adding a third pass to the algorithm, which tries to fix any violation of the criterias.

A less costly approach is to base the coarsening on the parallel maximal independent set algorithm [28], in order to ensure C2. Popular methods in this category is the Parallel Modified Independent Set (PMIS) [10] and the Hybrid Modified Independent Set (HMIS) [10]. These algorithms are inherently parallel, where each PE builds an independent set for the unknowns it owns. The PMIS algorithm coarsens only using the set based algorithm while HMIS combines the first pass of RS (for the local unknowns) and runs PMIS to cleanup the shared boundary unknowns. The parallelism in these methods comes with a price, namely a violation of criterion C1. Instead both methods enforce a less stringent requirement [10],

C1': Each F-point needs to strongly depend on at least one C-point.

The penalty of not enforcing C1 comes in the form of vanishing terms in the interpolation formula. Therefore, for PMIS and HMIS coarsening schemes the interpolation formula needs to be modified in order to handle these cases, which in turn could lead to a slower convergence time but with an improved initial coarsening time.

Despite all the successful work, most of the issues with formulating a parallel AMG method is due to the programming model used. If implemented using message passing, the large amount of fine grained parallelism and global dependencies in an algorithm like RS makes an efficient implementation challenging. Hence, if implemented using a PGAS language, the C and F points could be put into the global address space, which allows for efficient and novel parallel coarsening algorithms.

Chapter 6

Software

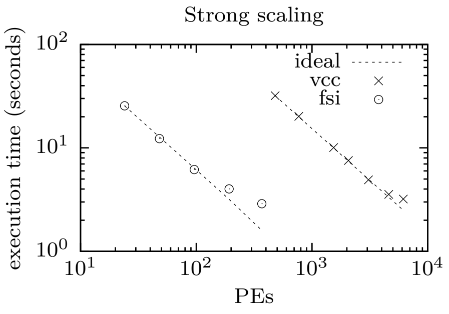
This thesis has contributed to many software packages. In this chapter we give a brief overview of the different codes and highlight the impact made by this work. This chapter also serves as an introduction for the more software oriented Paper I, IV, V, VI and VII.

6.1 The FEniCS project

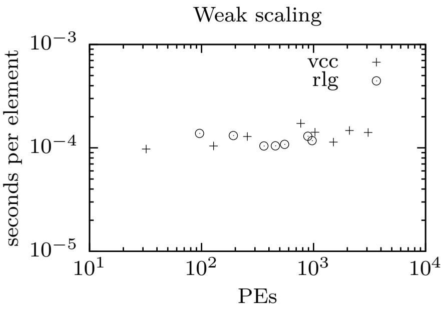
Writing high performance software for scientific computations is a delicate task. These codes are often developed on an application to application basis, highly optimized to solve a certain problem. The FEniCS project [13] seeks to automate the scientific software process. Instead of developing one code per application one should have one general code that solves instances of problems from a large family in an automated fashion. In FEniCS, a solver takes the equation and discretization method as input in a high level language close to mathematical notation. Low-level assembly functions are then generated by a FEniCS compiler. This means that the development of physical models and discretization methods can be done on a high level, implying robustness and enabling high speed of development. The contribution of this thesis is to bring high performance computing capability into the project.

6.1.1 DOLFIN HPC

The core part of FEniCS is the Object-Oriented finite element library DOLFIN [27], from which we have developed a high performance branch optimized for distributed memory architectures. DOLFIN handles mesh representation and finite element assembly but relies on external libraries for solving the linear systems. Our high performance branch also extends DOLFIN with parallel mesh refinement and dynamic load balancing capabilities as presented in this thesis.



(a) Strong scalabilty test on a Cray XE6.



(b) Weak scaling of the adaptive process.

Figure 6.1: Strong and weak scalability of the flow solvers in Unicorn. Here, fsi refers to a flexible mixer plate, vcc refers to the flow past a car and rlg refers to a landing gear configuration. The dashed lines in the left figure display the ideal scaling for each configuration.

6.1.2 Unicorn

On top of DOLFIN we have developed Unicorn [19], a unified continuum mechanics solver with adaptive mesh algorithms based on a posteriori error estimation. The incompressible flow solver in Unicorn has been the basis for most large scale simulation in this thesis such as [25, 44], and their needs have been the major driving force for optimization and new development within FEniCS. For example, the development of checkpoint restart capability and scalable parallel I/O in DOLFIN HPC.

6.1.3 Parallel performance

The performance and scalability of DOLFIN and Unicorn has improved quite a lot during the time of this work. With its first parallel assembly and solve on 4 PEs in 2008, the software has been verified to perform well on as much as 12288 PEs, and has been tested on up to 24576 PEs during late 2012. Unicorn's good strong scalability (fixed total problem size) has also been shown for several industrial applications (Figure 6.1a). Weak scaling (fixed problem size per PE) is also an interesting metric for evaluating the solver's performance throughout several adaptive iterations, which has been measured for various applications (Figure 6.1b). For more details please refer to [25, 19].

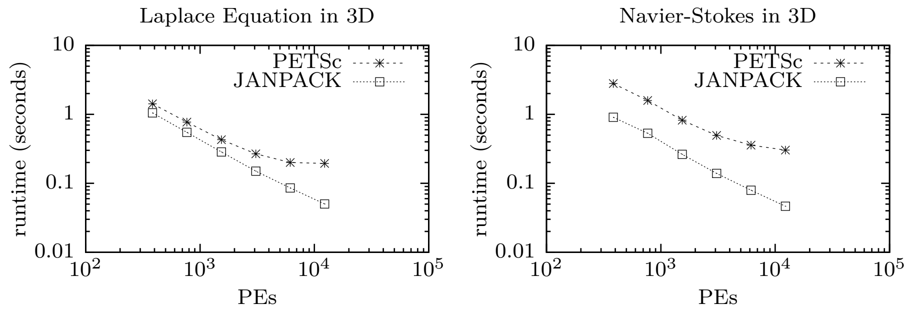


Figure 6.2: Sparse matrix assembly benchmark. Measuring mean reassembly time for three dimensional Laplace equation (left) and Navier-Stokes equation (momentum part) (right).

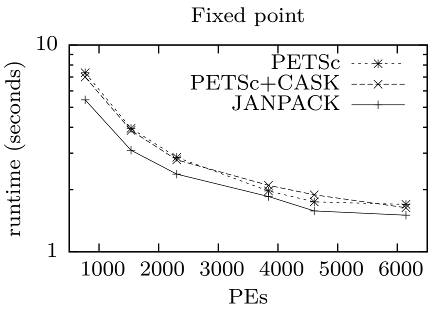


Figure 6.3: Average time for computing a time-step in Unicorn's incompressible flow solver for three different linear algebra backends.

6.2 JANPACK

The work on scalable linear algebra has resulted in a software package called JANPACK [23], written in Unified Parallel C (UPC) [40] and released as open source under the BSD license. JANPACK uses the alternative stack-based representation of a sparse matrix presented in §5.1.2, which has proven (see for example Figure 6.2) to outperform other well known MPI based linear algebra packages [24] (Paper V). The package also contains linear solvers, such as conjugate gradient (CG) and biconjugate gradient stabilized (Bi-CGSTAB) Krylov solvers. A set of inexpensive preconditioners has been implemented, all based on a block Jacobi approach. More precisely, the package contains a Jacobi preconditioner and two different incomplete factorization methods, ILU and D-ILU where the factorization is performed on the local blocks of the matrix, without overlap. The performance of these linear solvers has proven to be good, on a par with the solvers from the well known PETSc library and even Cray's optimized version of PETSc called CASK. As in illustration, in Figure 6.3 the average time for computing a time-step in Unicorn's incompressible flow solver is presented for a 3D flow problem. Where, JANPACK is successfully compared against two other linear algebra backends. For a more detail please refer to Paper VII.

JANPACK is also equipped with an algebraic multigrid solver and preconditioner. The entire AMG framework runs in parallel and supports several different coarsening strategies, a classic Ruge-Stüben, parallelized using a novel PGAS based approach, and regular PMIS and HMIS schemes. This work is summarized in Paper VI.

6.2.1 Hybrid interface

Another key feature of JANPACK is its hybrid interface. Despite all the appealing features of UPC, a PGAS based languages is seldom used in production codes due to the cost involved in rewriting or replacing legacy software. The hybrid interface allows developers to prepare old legacy codes for exascale computing by replacing bits and pieces of their old code and mix it freely with UPC, thus creating hybrid MPI/PGAS applications which is a quite unusual combination.

Chapter 7

Applications

Complex industrial flow problems are notoriously expensive to compute. In this chapter we demonstrate the impact of this work by presenting a set of simulations, which had not been possible to perform without the contribution of this thesis. This chapter serves as an introduction to the more application oriented Paper II, III, VIII and IX.

7.1 Aeroacoustic sources for a rudimentary landing gear

Stated as a benchmark problem in connection with the 16th AIAA/CEAS Aeroacoustics Conference in 2010. The goal was to assess the accuracy of different simulation techniques in the field of aeroacoustics by comparing numerical results with detailed experimental data. The problem we addressed consisted of the computation of aeroacoustic sources due to turbulent flow around a simplified geometry of a four-wheel landing gear (Figure 7.1).

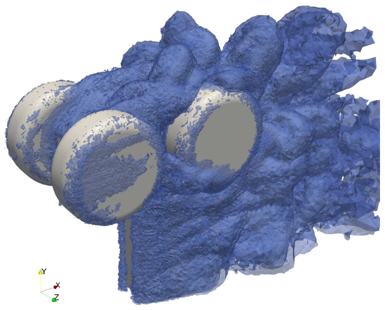


Figure 7.1: Vorticity computed around the rudimentary landing gear.

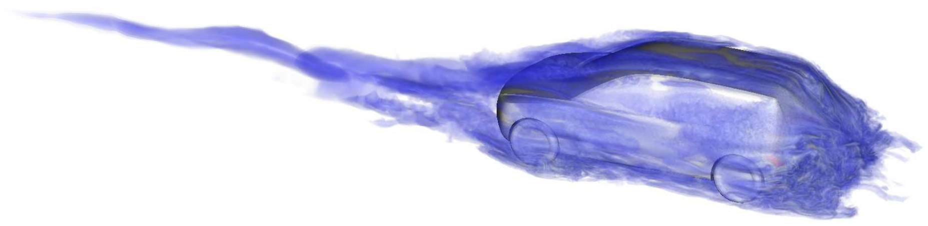


Figure 7.2: Volume rendering of the solution to the adjoint problem. The solution characterizes sensitivity in the output of interest, in this case drag.

This benchmark problem was one of our first attempts to solve an industrial problem using adaptivity. With an initial mesh of around 70k vertices we obtained after 7 iterations of adaptive refinement with respect to drag a mesh of ca 1M vertices. The average turn around time for an adaptive iteration was around 12 hours, in total the entire problem (adaptive iterations + final simulation) consumed around 71000 core hours. For a more detailed description of the simulation and results please refer to [43] and Paper VIII.

7.2 Turbulent flow past a full car model

Aerodynamic forces are of key importance in the design of a car, directly connected to fuel consumption. We have investigated the ability of Unicorn to accurately and efficiently compute the forces on a full car model called Volvo Research into Automotive Knowledge (VRAK). A principal goal of this work was to demonstrate that LES in the form of G2 is mature enough to offer an alternative to RANS in industrial applications. Using wind tunnel data for the Volvo car we have achieved this goal with convergence in aerodynamic drag force after some adaptive iterations (see Figure 7.2 and 4.2). This simulation demonstrated our solver's capability to efficiently solve large scale flow problems and the results (Paper III) were presented at Supercomputing 2011. The initial mesh consisted of only 58k vertices and 280k elements. After 13 adaptive iterations the number had increased to 4.5M vertices and almost 24M elements. During the simulation we balanced the number of PEs such that one time step took roughly $1-2$ seconds to compute, which gave a turn around time of 48 hours for 150k time-steps, a performance almost within the time-frame for important industrial simulations.

7.3 Aeroacoustic sources for a nose landing gear

For the 18th AIAA/CEAS Aeroacoustics Conference in June 2012, we ran one of our largest simulations up to date: we used over 1.5M core hours on a Cray XE6 to compute the flow past a Gulfstream G550 nose landing gear, [44] and Paper IX.

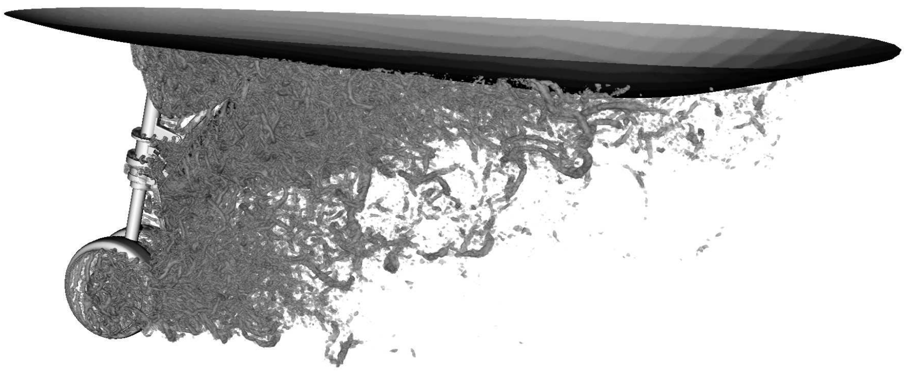


Figure 7.3: Vortices generated around a Gulfstream G550 nose landing gear, illustrated as a volume rendering of the λ_2 criterion.

The highly complex geometry (see Figure 7.3), including details such as bolts and wheel rims, was certainly challenging for Unicorn, and only a few CFD frameworks could currently produce reliable unsteady flow solutions for such a configuration. During the meeting our results were anonymously compared to other computational results and to experiments and no significant differences were found.

This project pushed our limits both in terms of simulation and postprocessing capabilities. Regarding the computational resources, 900k core hours out of the 1.5M were used for the adaptive process using various number of PEs. The final simulation was executed on 2304 PEs, consuming 600k core hours. Since one of the required deliveries for the meeting was power spectral density (PSD) data for a set of microphones, we had to sample the simulation at 32kHz, which put pressure on our parallel I/O routines. Finally, this project is an excellent illustration of the kind of FEniCS applications enabled by the contribution of this thesis.

Chapter 8

Summary and outline of future work

In this thesis we have presented our work on adaptive finite element methods suitable for high performance parallel computer architectures. Our adaptive framework is developed for efficient simulations on unstructured meshes, with fully distributed algorithms for assembly, error estimation, mesh refinement and load balancing. We have shown that a general finite element framework can be parallelized efficiently for unstructured meshes.

This work has lead us into the development of new parallel local mesh refinement methods and dynamic load balancing techniques based on a priori workload estimation. The core part of this work is presented in Paper I. Using the adaptive framework we have performed several large scale simulations of turbulent flow problems on realistic geometries. The results has been validated with good agreement at several conferences, for example the AIAA BANC-I workshop (Paper VIII) and the AIAA BANC-II meeting (Paper IX). Our solver has also been an important tool in the development of new numerical methods for high Reynolds number flow (Paper II and IV). The efficiency of our solver has been a high priority when addressing these large scale problems, therefore a non-negligible amount of time has been spent in tuning and optimization of our solver for scalability up to thousands of PEs (Paper V,VI and VII).

Recently the focus has shifted towards new parallel programming models and their applicability to finite element solvers, in particular the family of Partitioned Global Address Space languages. This work has resulted in the development of a PGAS based linear algebra library and hybrid programming models, combining traditional message passing with PGAS (Paper V, VI and VII).

There are several opportunities to enhance our framework further. It is interesting to investigate the impact of extensive use of global address space languages, e.g. in mesh adaption. With such continued work, DOLFIN and Unicorn would certainly be ready to meet the challenges of exascale computing.

Bibliography

[1] 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), pages 483–485, New York, NY, USA, 1967. ACM.

[2] M. Aspnäs, A. Signell, and J. Westerholm. Efficient Assembly of Sparse Matrices Using Hashing. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 900–907. Springer Berlin / Heidelberg, 2006.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.

[4] E. Bänsch. An adaptive finite-element strategy for the three-dimensional time-dependent navier-stokes equations. *J. Comput. Appl. Math.*, 36(1):3–28, 1991.

[5] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. Technical Report RNR-92-033, NAS Systems Division, NASA Ames Research Center, 1992.

[6] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Athena Scientific, 1997.

[7] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[8] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial.* SIAM, second edition, 2000.

[9] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[10] H. De Sterck, U. Yang, and J. Heys. Reducing Complexity in Parallel Algebraic Multigrid Preconditioners. *SIAM J. Matrix. Anal. A.*, 27(4):1019–1039, 2006.

[11] E. F. V. de Velde. *Concurrent Scientific Computing*, volume 16 of *Texts in Applied Mathematics.* Springer-Verlag, New York, NY, USA, 1994.

[12] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Inform. Process. Lett.*, 11(1):1 – 4, 1980.

[13] FEniCS. FEniCS project. http://www.fenicsproject.org, 2003.

[14] P. Gottschling and D. Lindbo. Generic compressed sparse matrix insertion: algorithms and implementations in mtl4 and fenics. In *Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 2:1–2:8, 2009.

[15] P. Gottschling and A. Lumsdaine. MTL4 Web page, 2010. http://mtl4.org.

[16] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155 – 177, 2002.

[17] J. Hoffman. Adaptive simulation of the subcritical flow past a sphere. *J. Fluid Mech.*, 568:77–88, 2006.

[18] J. Hoffman. Efficient computation of mean drag for the subcritical flow past a circular cylinder using general Galerkin G2. *Int. J. Numer. Meth. Fl.*, 59(11):1241–1258, APR 20 2009.

[19] J. Hoffman, J. Jansson, R. V. de Abreu, N. C. Degirmenci, N. Jansson, K. Müller, M. Nazarov, and J. H. Spühler. Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid-structure interaction for deforming domains and complex geometry. *Comput. Fluids*, 80(0):310 – 319, 2013.

[20] J. Hoffman and C. Johnson. *Computational Turbulent Incompressible Flow*, volume 4 of *Applied Mathematics: Body and Soul*. Springer, 2007.

[21] Y. Hu and R. Blake. An optimal dynamic load balancing algorithm. Technical Report DL-P95-011, Daresbury Laboratory, Warrington, UK, 1995.

[22] N. Jansson. Adaptive Mesh Refinement for Large Scale Parallel Computing with DOLFIN. Master's thesis, Royal Institute of Technology, School of Computer Science and Engineering, 2008. TRITA-CSC-E 2008:051.

[23] N. Jansson. JANPACK, 2012. http://www.csc.kth.se/~njansson/janpack.

[24] N. Jansson. Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-sided Communication. In *High Performance Computing for Computational Science – VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 128–139. Springer Berlin Heidelberg, 2013.

[25] N. Jansson, J. Hoffman, and M. Nazarov. Adaptive Simulation of Turbulent Flow Past a Full Car Model. In *State of the Practice Reports*, SC '11, pages 20:1–20:8, New York, NY, USA, 2011. ACM.

[26] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *ICPP (3)*, pages 113–122, 1995.

[27] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[28] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[29] MPI Forum. Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/.

[30] L. Oliker. PLUM Parallel Load Balancing for Unstructured Adaptive Meshes. Technical Report RIACS-TR-98-01, RIACS, NASA Ames Research Center, 1998.

[31] J. R. Pilkington and S. B. Baden. Partitioning with Spacefilling curves. Technical Report CS94-349, Department of Computer Science and Engineering, University of California, San Diego, 1994.

[32] C. Pommerell. *Solution of large unsymmetric systems of linear equations.* PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1992.

[33] M. Rivara. Mesh Refinement Processes Based on the Generalized Bisection of Simplices. *SIAM J. Numer. Anal.*, 21(3):604–613, 1984.

[34] J. W. Ruge and K. Stüben. *Algebraic Multigrid*, volume 3 of *Frontiers Appl. Math.*, chapter 4, pages 73–130. SIAM, Philadelphia, PA, USA, 1987.

[35] Y. Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA, USA, 2nd edition, 2003.

[36] Y. Saad and M. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 7(3):856–869, 1986.

[37] K. Schloegel, G. Karypis, and V. Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *J. Parallel. Distr. Com.*, 47(2):109–124, 1997.

[38] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker. A Performance Study of Diffusive vs. Remapped Load-Balancing Schemes. In *11th Intl. Conference on Parallel and Distributed Computing Systems*, 1998.

[39] U. Schumann. Subgrid scale model for finite difference simulations of turbulent flows in plane channels and annuli. *J. Comput. Phys.*, 18(4):376 – 404, 1975.

[40] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[41] H. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comp.*, 13(2):631–644, 1992.

[42] H. A. van der Vorst. *Iterative Krylov Methods for Large Linear Systems.* Cambridge University Press, 2009.

[43] R. Vilela de Abreu, N. Jansson, and J. Hoffman. Adaptive Computation of Aeroacoustic Sources for Rudimentary Landing gear. In *Proceedings of the First Workshop on Benchmark problems for Airframe Noise Computations (BANC-I)*, Stockholm, 2010.

[44] R. Vilela de Abreu, N. Jansson, and J. Hoffman. Computation of aeroacoustic sources for a complex nose landing gear geometry using adaptivity. In *Proceedings of the Second Workshop on Benchmark problems for Airframe Noise Computations (BANC-II)*, Colorado Springs, 2012.

[45] C. Walshaw and M. Cross. Parallel Mesh Partitioning on Distributed Memory Systems. In *Computational Mechanics Using High Performance Computing*, pages 59–78. Saxe-Coburg Publications, Stirling, 2002.

[46] J. Walter and M. Koch. uBLAS – BOOST Basic Linear Algebra Library, 2002.

[47] B. Wilkinson and M. Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, second edition, 2005.

[48] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Comput.*, 35(3):178 – 194, 2009.

Part II

Included papers