Coarsening of Simplicial Meshes for Large Scale Parallel FEM Computations with DOLFIN HPC

A parallel implementation of the edge collapse algorithm

B A L T H A S A R R E U T E R

Master of Science Thesis
Stockholm, Sweden 2013

Coarsening of Simplicial Meshes for Large Scale Parallel FEM Computations with DOLFIN HPC

A parallel implementation of the edge collapse algorithm

B A L T H A S A R R E U T E R

Master's Thesis in Scientific Computing (30 ECTS credits)
Master Programme in Scientific Computing (120 credits)
Royal Institute of Technology year 2013
Supervisors at KTH were Johan Hoffman and Niclas Jansson
Examiner was Michael Hanke

TRITA-MAT-E 2013:37
ISRN-KTH/MAT/E--13/37--SE

Royal Institute of Technology
School of Engineering Sciences

KTH SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci

Abstract

Adaptive mesh refinement and coarsening methods are effective techniques to reduce the computation time of finite element based solvers. Parallel implementations of such adaption routines, suitable for large scale computations on distributed memory machines, need additional care. In this thesis, a coarsening technique based on edge collapses is presented, its implementation and optimization for parallel computations explained and it is analyzed with respect to coarsening efficiency and performance. As a possible application the use of mesh coarsening in adaptive flow simulations is demonstrated.

Sammanfattning

Utglesning av simplex-nät för storskaliga parallella FEM-beräkningar med DOLFIN HPC

Adaptiv förfining och utglesning av element-nät är effektiva tekniker för att minska beräkningstiden för finita-element-lösare. Implementering av sådana adaptions-rutiner, passande för stora beräkningar på maskiner med distribuerat minne, kräver stor omsorg. I detta arbete presenteras en utglesnings-metod baserad på kant-sammanslagningar. Dess implementering och optimering för parallell-beräkningar förklaras och analyseras med avseende på glesnings-effektivitet och tidsåtgång. Som tillämpning visas nätutglesning i adaptiv strömningssimulering.

Contents

Chapter 1

Introduction

> *While I am describing to you* how *nature works, you won't understand* why *nature works that way. But you see, nobody understands that.*
>
> – Richard Feynman [1]

Even if we don't know *why* nature works in a certain way, or maybe will never know, we know in many cases physical and mathematical descriptions of ways *how* nature works. Often in the form of partial differential equations, we know laws that describe nature's processes, like the Navier-Stokes equations for fluids or Maxwell's equations for electromagnetic waves. And, following the spirit of Richard Feynman, by studying those laws, we gain more knowledge about the *how*, which might eventually even lead to some understanding of the *why*.

Over the last decades, computer simulations became one of the most powerful tools in investigating those laws. Many different numerical methods arose to solve the equations of such laws, allowing to study problems and scenarios that mankind wasn't able to investigate before.

The finite element method (FEM) is one of the most popular techniques used to solve partial differential equations. Due to its general nature it is widely applied to problems arising in science and engineering, but, to obtain a sufficiently accurate solution, it depends on adequate mesh resolutions. However, a priori knowledge about the required resolution is rarely available. By preferring too fine meshes over too coarse meshes, one can make sure to maintain stability of the numerical method and obtain the desired accuracy for the solution. But this comes at the cost of long computation times, since a larger number of elements increases the workload and reduced element sizes demand smaller time steps. Adaptive mesh refinement and coarsening methods are effective techniques to reduce the computation time of finite element based solvers.

By evaluating the error, mesh regions with a large error are refined by adding more elements, while regions with small error indicators are coarsened by removing some elements. That way, a mesh with specific local regions emerges from an initial coarse mesh, allowing to obtain a solution with reasonable computation time and high accuracy.

In this thesis, the edge collapse algorithm for mesh coarsening and its implementation in DOLFIN HPC is described. This FEM library was designed for large scale parallel computations, hence a key aspect of this project was to create an efficient implementation for massively parallel environments. As the name of the algorithm already reveals, it coarsens a mesh by collapsing edges. The basic idea is to pull one endpoint vertex of an edge onto the other, hence removing one vertex from the mesh. Elements adjacent to the edge end up with zero area or volume and are removed as well, which leaves the mesh with a smaller number of cells and vertices. Since DOLFIN HPC is limited to simplicial meshes in two or three dimensions, i. e., triangles or tetrahedrons, this thesis focuses on such mesh types, although many of the presented concepts are also true for other types.

This report is structured in three parts: The first part will cover the used methodology, including a short introduction into the finite element method, its variant *General Galerkin* and adaptive simulations, before explaining the edge collapse scheme. The second part is focused on aspects of the implementation. It gives a short overview over DOLFIN HPC, and discusses the implementation and optimization of the coarsening algorithm. In the final part the developed algorithm is analyzed and applied. These evaluations include the efficiency of the method both in terms of mesh coarsening and in terms of parallel performance and scalability. It concludes with an application of coarsening in two flow simulation scenarios. Suggestions for further improvements and future work close this thesis.

Part I

Methodology

Chapter 2

The finite element method

This first part will explain the methodology used or developed during this project, in order to lay the necessary background and introduce involved theory for later parts. Since the development of mesh coarsening was done in the context of the finite element method, this technique will be outlined first, followed by the details about mesh coarsening itself.

The finite element method (FEM) is a standard tool for the numerical solution of differential equations and thanks to its general nature, has been applied to a variety of different problem settings. It is particularly popular in mechanical engineering, for example for structural and fluid computations. Its basic idea is to break the computation domain into an arbitrary number of elements of finite size. Defining ansatz functions on each element and inserting those into the differential equation allows, combined with initial and boundary conditions, to obtain systems of equations, which can then be solved.

Due to the long existence and broad usage of FEM, there is no lack of good literature on the method and its many different types. Therefore, only a short introduction into the major steps of solving a differential equation using this technique will be given by means of a simple example. It is followed by a summary of the General Galerkin (G2) method, a variant of FEM, and a short introduction into adaptive finite element computations.

2.1 Introduction to FEM

This section follows the introduction given by Kirby and Logg [2], which was chosen for its comprehensible nature and good applicability to the rest of this thesis.

As a simple example, the chosen differential equation to be solved is Poisson's equation on a domain $\Omega \subset \mathbb{R}^d$ with a set of Dirichlet and Neumann boundary conditions on Γ_D and Γ_N respectively:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= u_0 && \text{on } \Gamma_D \subset \partial\Omega, \\ -\partial_n u &= g && \text{on } \Gamma_n \subset \partial\Omega, \end{aligned} \tag{2.1}$$

where ∂_n denotes the derivate in normal direction.

Transforming 2.1 into weak form and integration by parts is the first step towards a finite element discretization:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \partial_n u v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in \hat{V}. \tag{2.2}$$

Let the test space

$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D\} \tag{2.3}$$

and trial space

$$V = \{v \in H^1(\Omega) : v = u_0 \text{ on } \Gamma_D\}, \tag{2.4}$$

then the boundary term vanishes on the Dirichlet boundary and the variational problem to solve reads as: *Find $u \in V$ such that*

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_N} g v \,\mathrm{d}s \quad \forall v \in \hat{V}. \tag{2.5}$$

Restricting the variational problem 2.5 to discrete spaces $V_h \subset V$ and $\hat{V}_h \subset \hat{V}$ gives the discrete problem: *Find $u_h \in V_h$ such that*

$$\int_\Omega \nabla u_h \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x - \int_{\Gamma_N} g v \,\mathrm{d}s \quad \forall v \in \hat{V}_h. \tag{2.6}$$

By choosing suitable basis functions $\{\phi_j\}_{j=1}^N$ for V_h and $\{\hat{\phi}_j\}_{i=1}^N$ for $\hat{V}_h$ (both spaces are of dimensionality N), the solution can be expressed in terms of these basis functions:

$$u_h(x) = \sum_{j=1}^N U_j \phi_j(x). \tag{2.7}$$

This leads to a system of N equations:

$$\sum_{j=1}^N U_j \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x = \int_\Omega f \hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_N} g \hat{\phi}_i \,\mathrm{d}s, \quad i = 1, 2, \ldots, N. \tag{2.8}$$

Let $U = (U_1, U_2, \ldots, U_N)$ be the vector of coefficients, then the finite element solution $u_h = \sum_{j=1}^N U_j \phi_j$ can be computed by solving the linear system

$$AU = b, \tag{2.9}$$

with

$$\begin{aligned} A_{ij} &= \int_\Omega \nabla \phi_j \cdot \nabla \hat{\phi}_i \,\mathrm{d}x, \\ b_i &= \int_\Omega f \hat{\phi}_i \,\mathrm{d}x - \int_{\Gamma_N} g \hat{\phi}_i \,\mathrm{d}s. \end{aligned} \tag{2.10}$$

In the FEM context the system matrix A is called *stiffness matrix* and the right hand side vector b is called *load vector*. For more complex equations than Poisson's equation the entries of the stiffness matrix and load vector are of course more involved. However, the basic steps persist: weak formulation and integration by parts, followed by space discretization and inserting the basis functions to obtain a system of equations that can then be solved.

2.2 General Galerkin (G2)

General Galerkin is a variant of the finite element method with piecewise linear approximation in space and time combined with numerical stabilization. It was used before to solve the Navier-Stokes equations to simulate turbulent flow and fluid-structure interactions in multiple scenarios [3–5]. A comprehensive description of the method with many details and examples was done by Hoffman and Johnson [6]. The following description of the G2 method, as it was applied for the showcase in chapter 8, follows the introductions by Jansson [3] and Hoffman et al. [5].

An incompressible Newtonian fluid flow is described by the Navier-Stokes equations with constant kinematic viscosity $\nu > 0$ in $\Omega \subset \mathbb{R}^3$ over a time interval $I = (0, T]$:

$$\begin{aligned} \dot{u} + (u \cdot \nabla)u + \nabla p - \nu \Delta u &= f, && (x,t) \in \Omega \times I, \\ \nabla \cdot u &= 0, && (x,t) \in \Omega \times I, \\ u(x,0) &= u^0(x), && x \in \Omega, \end{aligned} \tag{2.11}$$

with velocity $u(x,t)$, pressure $p(x,t)$, initial data $u^0(x)$ and force $f(x,t)$. The stress tensor is defined by $\sigma_{ij} = -\nu\epsilon_{ij}(u) + p\delta_{ij}$, with strain rate tensor $\epsilon_{ij} = 1/2(\partial u_i/\partial x_j + \partial u_j/\partial x_i)$ and Kronecker delta δ_{ij}. An example for a suitable choice of boundary conditions is given in section 8.1.1.

This system is discretized by a continuous Galerkin method with piecewise linear functions, i. e., the so called cG(1), in space and time, hence referred to as cG(1)cG(1)-method. It uses continuous piecewise linear test and trial functions in space and piecewise constant test functions and piecewise linear trial functions in time. The time interval is split into a sequence of discrete time steps $0 = t_0 < t_1 < \ldots < t_n = T$ with associated time intervals $I_n = (t_{n-1}, t_n)$ of length $k_n = t_n - t_{n-1}$. Space discretization is done with respect to a tetrahedral mesh $\mathcal{T}_n = \{K\}$ with mesh size $h_n(x)$ and K naming elements in $\mathcal{T}_n$.

Let $W^n \subset H^1(\Omega)$ be a finite element space with piecewise linear functions on $\mathcal{T}_n$ and let $W^n_\omega = \{v \in W^n | v = \omega \text{ on } \Gamma\}$ the set of functions satisfying the Dirichlet boundary condition $v|_\Gamma = \omega$. Furthermore, let

$$\Big(u, v\Big) := \sum_{K \in \mathcal{T}_n} \int_K u \cdot v \, \mathrm{d}x \qquad \text{and} \qquad \Big(\epsilon(u), \epsilon(v)\Big) = \sum_{i,j=1}^{3} \Big(\epsilon_{ij}(u), \epsilon_{ij}(v)\Big).$$

Then the cG(1)cG(1)-method for the Navier-Stokes equations with homogeneous Dirichlet boundary conditions seeks an approximate solution $\hat{U} = (U, P)$ and takes

the form: *For* $n = 1, \ldots, N$ *find* $(U^n, P^n) \equiv (U(t_n), P(t_n))$ *with* $U^n \in V_0^n \equiv [W_0^n]^3$ *and* $P^n \subset W^n$ *such that*

$$\begin{aligned}\Big((U^n - U^{n-1})k_n^{-1} + \bar{U}^n \cdot \nabla \bar{U}^n, v\Big) + \Big(2\nu\epsilon(\bar{U}^n), \epsilon(v)\Big) - \Big(P^n, \nabla \cdot v\Big) \\ + \Big(\nabla \cdot \bar{U}^n, q\Big) + SD_\delta^n\,(\bar{U}^n, P^n; v, q) = \Big(f, v\Big) \quad \forall \hat{v} = (v, q) \in V_0^n \times W^n,\end{aligned} \tag{2.12}$$

with $\bar{U}^n = \frac{1}{2}(U^n + U^{n-1})$ *and* P^n *piecewise constant over time.* Numerical stabilization is done by a weighted least squares method based on the residual and enters in the stabilizing term

$$\begin{aligned}SD_\delta^n\,(\bar{U}^n, P^n; v, q) \equiv &\Big(\delta_1(\bar{U}^n \cdot \nabla \bar{U}^n + \nabla P^n - f), \bar{U}^n \cdot \nabla v + \nabla q\Big) \\ &+ \Big(\delta_2 \nabla \cdot \bar{U}^n, \nabla \cdot v\Big)\end{aligned} \tag{2.13}$$

with the stabilizing parameters $\delta_1 = \kappa_1(k_n^{-2} + |U^{n-1}|^2 h_n^{-2})^{-1/2}$ and $\delta_2 = \kappa_2 h_n$. κ_1 and κ_2 are positive constants without any units.

2.3 Adaptive finite elements

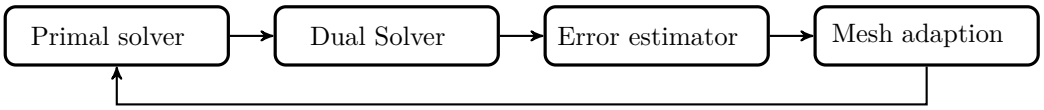


Figure 2.1. Schematic representation of a typical adaptive finite element simulation.

A complex flow problem, like the turbulent flow around an object, introduces local regions of interest, e. g., the wake downstream behind the object, and leaves other regions of less interest. These restricted regions one is interested in require a high spatial resolution, but to keep the computational cost to a minimum the resolution should only be increased in these regions. However, there is often no or insufficient a priori knowledge about shape and location of these regions of interest, making it impossible to create an initial mesh that accounts for those with the appropriate local resolution. Consequently, one has two options to tackle this problem: either use a mesh with uniformly high resolution, giving the desired accuracy of the solution but at the cost of a very high computation time. Or, use a coarse mesh initially and pair it with an adaptive solver.

An adaptive finite element scheme determines the location of regions that require a higher or lower spatial resolution and adapts the mesh accordingly. It uses knowledge from a computed (but not necessarily sufficiently accurate) solution to derive these information before continuing or re-starting to solve the problem. A necessary assumption for that is that the error is localized. For G2-simulations the mesh adaption can be based on the a posteriori error estimate for cG(1)cG(1) derived by Hoffman and Johnson [6] and applied by Hoffman et al. [5]:

Let $\hat{u} = (u, p)$ be a weak solution to the Navier-Stokes equations 2.11, $\hat{U}$ a finite element approximation and $\hat{\varphi} = (\varphi, \theta)$ a solution to a linearized dual problem. An error estimate is defined with a mean value output $M(\hat{U}) = ((\hat{U}, \hat{\psi}))$, with $((\cdot, \cdot))$ the inner product over the space-time domain, and a weight function $\hat{\psi}$:

$$|M(\hat{u}) - M(\hat{U})| \leq \sum_{K \in \mathcal{T}_n} \mathcal{E}_K, \tag{2.14}$$

with the element-wise error indicators

$$\mathcal{E}_K \equiv \sum_{n=1}^{N} \Big[\int_{I_n} \int_K \sum_i |R_i(\hat{U})|_K \cdot \omega_i \, \mathrm{d}x \, \mathrm{d}t + \int_{I_n} | \, SD_\delta^n \, (\hat{U}; \hat{\varphi})_K | \, \mathrm{d}t \Big], \tag{2.15}$$

where R_i are residuals of the Navier-Stokes equations and ω_i are stability weights in the form of derivatives of the dual solution $\hat{\varphi}$ multiplied by the local mesh size. The error indicators incorporate both discretization error and modeling error, with the latter coming from the stabilization.

The quality of this error estimate depends on the approximation of the dual problem, which typically suffers from errors due to linearization and numerical approximation. Using the error estimator a simple adaptive algorithm using only mesh refinement looks like algorithm 2.1 (see also figure 2.1), as described by Hoffman et al. [5].

Algorithm 2.1 Simple adaptive algorithm

Start with initial coarse mesh $\mathcal{T}_0$. Let $n = 0$:

1: For mesh $\mathcal{T}_n$: compute primal and dual problem.
2: Compute error estimates $\mathcal{E}_K \, \forall K \in \mathcal{T}_n$.
3: If $\sum_{K \in \mathcal{T}_n} \mathcal{E}_K <$ TOL then stop, else:
4: Mark some chosen percentage of the elements with highest $\mathcal{E}_K$ for refinement.
5: Generate refined mesh $\mathcal{T}_{n+1}$, set $n = n + 1$, and goto 1.

One should note that an iteration of the adaptive algorithm does not necessarily correspond to a time step. On the contrary, each iteration can include solving primal and dual problem over the whole time interval, if the location of the regions of interest does not change significantly over the simulation time but might need some time to develop.

An associated issue is that many physical problems have some initial phase before the system enters a stable configuration. For example, in a flow simulation it might take some time before a stationary solution of the flow exists around an object. To avoid recomputing this initial phase in the next adaptive iteration, the solution could be projected into the adapted mesh and used as better initial guess, shortening the initial phase significantly.

More involved problems might even require mesh adaption after each time step, e. g., when regions of interest move during simulation time as it is the case in shock-wave simulations or some fluid-structure interaction problems.

Chapter 3

Mesh coarsening

In most applications of adaptive algorithms, as described in chapter 2.3, the initial mesh is chosen to be relatively coarse to keep the initial degrees of freedom and hence computation time minimal. But the accuracy of numerical solutions to differential equations always depends on the mesh size, therefore it is obvious that mesh refinement improves the accuracy of the computed solution. With this relation in mind, it is not obvious on first sight why one would apply mesh coarsening in an adaptive algorithm.

Especially in the context of time-dependent problems, situations might occur in which form and location of the regions requiring a high spatial resolution change over time. An example is a shock wave that travels through the simulation domain, resulting in a steep pressure gradient close to the shock front, which needs a high spatial resolution to be resolved accurately. Over simulation time the position of the wave front changes, making refinement necessary in new regions of the mesh, while previously refined regions retain a much higher resolution than necessary. The application of a coarsening scheme in those regions allows to reduce the computation time without a significant impact on the accuracy of the solution. A general description of such cases was given by Li et al. [7] and a specific application for supersonic flows has been done by Darwish et al. [8].

Mesh coarsening is also popular in the field of computer graphics, since rendering of surfaces is more expensive for a larger number of elements. If a certain object doesn't require a high resolution, for example because it is displayed very small, a coarser surface mesh suffices. In such cases it can be beneficial to coarsen the surface mesh prior to rendering, which has been examined in a number of publications [9–12].

An application of the techniques from computer graphics is also imaginable in the context of scientific computing: Meshes generated from CAD-models (like cars or airplanes) often contain many small details, leading to a surface mesh with a very high resolution. But especially very small details have neglectable effect on the solution and hence could be ignored in favor of a coarser surface mesh. Mesh coarsening could offer an automatic adaption of the surface resolution.

3.1 General techniques

Cougny and Shephard [13] distinguish between two different approaches of mesh adaption to obtain a certain distribution of the mesh resolution: To regenerate the entire mesh or to refine and coarsen locally. When using sufficiently good algorithms, the first approach promises to have optimal mesh quality and a distribution of element sizes close to the desired resolution in the resulting mesh. But this is computationally expensive and requires to map entire solution fields from one mesh to another. Local changes to the mesh can be done much more efficiently and are easier to parallelize.

One technique to implement such local mesh coarsening is to maintain a mesh hierarchy that keeps information about the relation between coarse and refined meshes. That way previously refined elements are coarsened by simply replacing all fine elements with the coarse element they originated from. This is used for example by the open source framework libMesh [14] and Biswas and Strawn [15]. Although this approach is computationally fast, it requires a large amount of extra information to be stored and coarsening is not possible beyond the initial mesh. Furthermore, sticking to a hierarchical mesh limits the usage of mesh optimization techniques to such methods that don't destroy the hierarchy [13, 16].

These drawbacks can be overcome when creating the coarser mesh directly from the fine mesh. The most popular algorithm for coarsening of simplicial meshes is "edge collapse" (sometimes also referred to as "QSlim" or "edge contraction"), other methods are mostly variations of this algorithm or discard a local portion of the mesh and retriangulate it using meshing algorithms. The edge collapse was applied in many scenarios [10–13, 16] and due to its simplicity and since most other methods are only variations or other forms of it, this was also the method of choice in this thesis.

3.2 Coarsening simplicial meshes by edge collapse

The upcoming section will describe the edge collapse in more detail, including details about all required components, followed by the resulting coarsening algorithm. Necessary considerations for parallelizing the algorithm conclude the chapter.

3.2.1 Edge collapse

Algorithm 3.1 Edge collapse

Given an edge e in a mesh $\mathcal{T}$, that is marked for collapse.
Let v_D, v_R be the endpoint vertices of e.

1: Pull vertex v_D onto vertex v_R.
2: $\forall K \in \{K \in \mathcal{T} | e \in K\}$: remove K from $\mathcal{T}$.

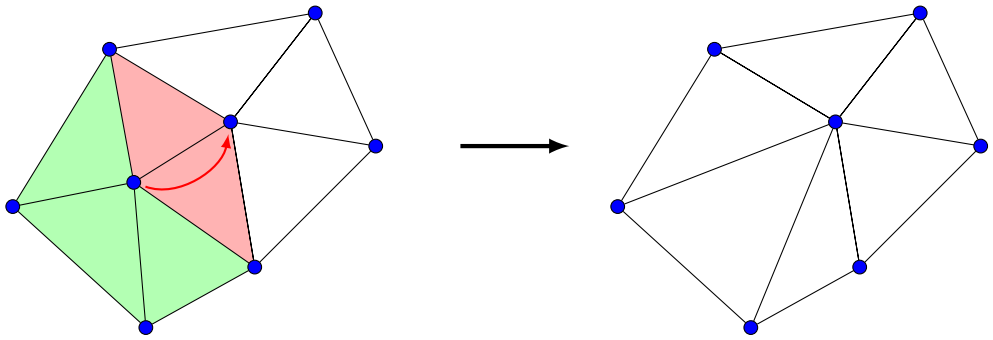


Figure 3.1. Edge collapse in a triangular mesh: One endpoint of the edge is pulled onto the other, causing the red elements to vanish and the green elements to stretch.

The edge collapse algorithm used (see algorithm 3.1) is similar to the algorithm described by Cougny and Shephard [13]. It always acts locally on a single edge that has been selected for coarsening and pulls one of the two endpoint vertices onto the other. All elements sharing this edge are reduced to zero volume and hence removed, while all cells containing the pulled vertex are increased in volume due to the new location of the vertex. This is equivalent to a retriangulation of the polyhedron formed by all mesh entities connected to the pulled vertex. An illustration of this procedure for a triangular mesh is given in figure 3.1. For tetrahedral meshes the method is the same, only the number of both deleted and modified cells is usually larger.

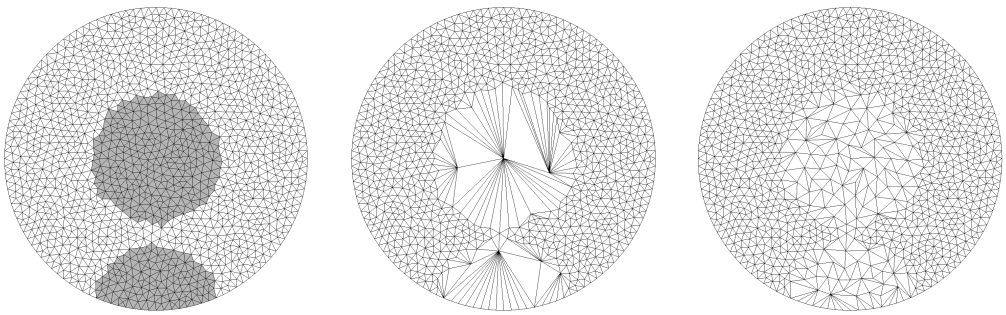


Figure 3.2. Coarsening with and without enforcing a vertex distribution. Two regions in a unit circle are selected for coarsening (left). Only very few vertices remain inside the regions when coarsening without controlling the vertex distribution (center). Coarsening when using the independent set algorithm generates a coarse mesh with evenly distributed vertices (right).

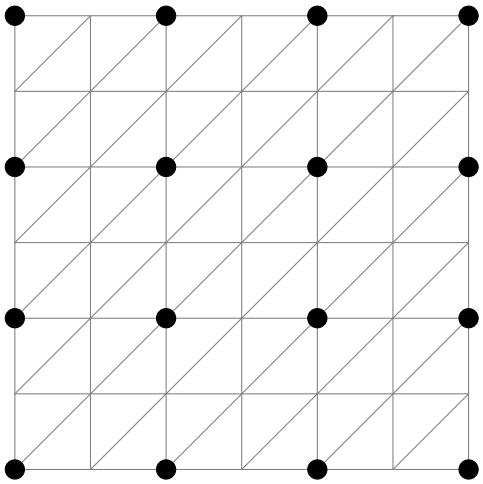


Figure 3.3. Example for an independent set of vertices in a structured triangular mesh. The vertices in the independent set are not connected directly to each other, however all other vertices are connected to one or more members of the set.

3.2.2 Maintaining a proper vertex distribution

In an adaptive simulation (see chapter 2.3) the regions for refinement and coarsening are for example determined from an error indicator by choosing a certain percentage of cells with highest or lowest indicators for refinement or coarsening respectively. Though solutions can vary rapidly, in many cases there will be at least some neighboring elements with error indicators of a similar magnitude, yielding that they are commonly selected for coarsening. If edge collapse is performed on all marked cells, this will leave only very few vertices in this marked region (see figure 3.2). Therefore the vertex distribution in the resulting coarse mesh has to be controlled. This can be achieved by the independent set approach described by Cougny and Shephard [13].

An independent set is a collection of vertices which are not connected to each other directly (see figure 3.3). The more restrictive variant is the maximal independent set.

Definition: maximal independent set (MIS) [17] The MIS of a graph $G = (V, E)$, where E is the edge set and V is the node set, is a set $U \subseteq V$ such that the following is true:

1. **independence:** if $u_1, u_2 \in U$ then $(u_1, u_2) \notin E$, and

2. **maximality:** no node $v \in V$ can be added to U without violating independence, i. e., all nodes $v \in V \backslash U$ are incident to some edge (v, u) such that $u \in U$.

For the application of mesh coarsening an initial independent set of vertices is computed and all vertices contained in this set are marked as forbidden for collapse and hence ensured to remain in the mesh. Unfortunately the problem to find a maximal independent set is NP-hard [18]. Luckily, for a proper vertex distribution it turned out non necessary to find the maximal independent set. Though an independent set smaller than the MIS, i. e., a set that only fulfills the first condition of above definition, is a less restrictive constraint on the vertex distribution, a successful edge collapse is bound by many more constraints than just this set, which will become evident in the remainder of this thesis.

Therefore, finding an independent set can be done by a greedy algorithm that starts with an empty set, simply iterates over all vertices in the mesh and adds them to the set if neither of the adjacent vertices belongs to the set. Such an approach is outlined in algorithm 3.2.

Algorithm 3.2 Finding an independent set

```
Start with empty set S and mesh T.
for all vertex v in T do
    if ∀ adjacent vertices w: w ∉ S then
        add v to S.
    end if
end for
```

The resulting independent set depends on the order in which one iterates over the vertices in the mesh, which is usually related to the numbering of vertices. The first vertex will always be included and later vertices are less likely to be included due to the larger number of elements in the set. However, the impact on the even distribution of vertices is small and therefore this does not cause any problems.

As visible from figure 3.2 coarsening of domain boundaries, although possible, is usually not desired since it can heavily affect the shape of the boundaries. Moreover, boundary conditions are often defined in terms of geometrical positions and hence would no longer be applied correctly if vertices from the boundary were deleted. To avoid this, algorithm 3.2 was modified such that it does not start with an empty set but an initial set $\mathcal{S}$ that already contains all boundary vertices. Then $\mathcal{S}$ is no longer an independent set following the classical definition, since it contains adjacent vertices on the boundary. Yet, the condition will be fulfilled for all interior nodes, which is why it will also be referred to as "independent set" in the remainder of this thesis.

3.2.3 Selecting edges for coarsening

As mentioned in the previous section, coarsening indicators are defined cell-wise, but the edge collapse works on edges. So some criterion is necessary to select the most suitable edge of a cell for coarsening.

Since edge collapse causes the remaining cells to be stretched along the collapsed edge, this can lead to skinny and elongated elements and yield a poor mesh quality. To overcome this problem the best and fastest choice is the shortest edge of the element.

But the independent set puts additional constraints on the selected edge: Especially when boundary vertices are included in the set (see chapter 3.2.2) there will be edges with both endpoints belonging to $\mathcal{S}$. Incorporating all these conditions gives the resulting edge-selection algorithm 3.3.

Algorithm 3.3 Edge selection

Given an element $K \in \mathcal{T}$ and and independent set $\mathcal{S}$.
Let $l_{\min} = \infty$.
for all edges $e \in K$ **do**
  Let v_1, v_2 be the endpoints of e.
  if ($v_1 \notin \mathcal{S}$ or $v_2 \notin \mathcal{S}$) and length$(e) < l_{\min}$ **then**
    $l_{\min} = \text{length}(e)$.
    $e_{\min} = e$.
  end if
end for
if no $e_{\min}$ found **then**
  K cannot be coarsened.
else
  Choose $e_{\min}$ for collapse.
end if

3.2.4 Selecting vertex for collapse

Algorithm 3.4 Select vertex for collapse

Let $\mathcal{S}$ be the independent set and e the edge to be collapsed.
Let v_1, v_2 be the endpoints of e.
if $v_1 \in \mathcal{S}$ **then**
  $v_D = v_2$ and $v_R = v_1$.
else if $v_2 \in \mathcal{S}$ **then**
  $v_D = v_1$ and $v_R = v_2$.
else
  $v_D = \max(v_1, v_2)$.
  $v_R = \min(v_1, v_2)$.
end if

Once an edge is chosen for collapse the question arises, which of the two endpoints should be pulled onto the other. If one of the two endpoints is in the independent set $\mathcal{S}$, this question is fairly easy to answer since then only the other

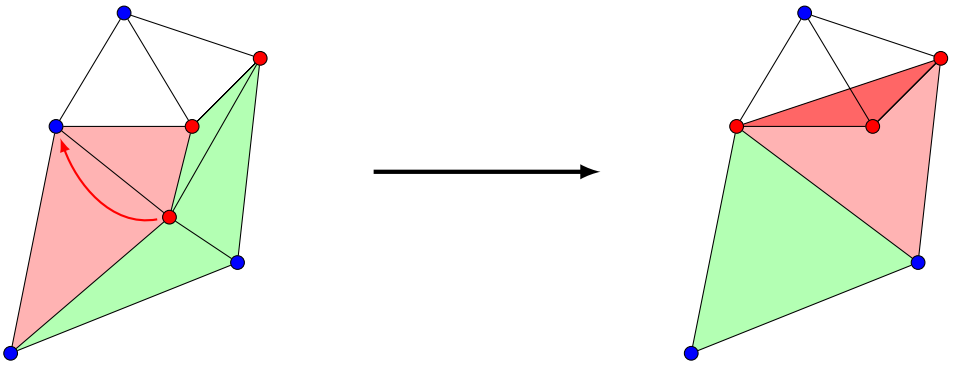


Figure 3.4. Example for a possible problem during an edge collapse. If the lower left cell is selected for coarsening, the above scenario is plausible as a possible collapse. Red cells again vanish after the edge collapse and green cells are modified. However due to the position and shape of the mesh entities, the collapse will lead to an invalid mesh structure. This can be detected by checking the orientation of the cells, since it will flip for the cell with the red vertices.

endpoint is allowed to be removed. But if both endpoints are not in the set, each of them could be deleted. This was resolved by simply collapsing the vertex with higher index onto the vertex with lower index (see algorithm 3.4). It is impossible for both vertices to belong to $\mathcal{S}$, since such an edge would not have been selected in the first place (see algorithm 3.3).

3.2.5 Mesh quality and validity after edge collapse

Since the edge collapse stretches the remaining elements along the collapsed edge, they can happen to become very long and thin, resulting in poor mesh quality. This can occur frequently for complex geometries in tetrahedral meshes and in the worst case a poor mesh can drastically increase the condition number of the linear system of the finite element method, causing the solver no longer to converge.

To avoid such problems, an additional check of the mesh quality was added. Since edge collapse is a local operation, only involving the elements adjacent to the deleted vertex, it is sufficient to check the quality of those. As a criterion for the quality of a cell the ratio of volume or area over diameter is chosen (for tetrahedral or triangle meshes respectively) with the diameter defined as twice the circumradius. This indicator becomes large for elements close to a regular simplex and smaller for stretched elements. Since the computation of this criterion is fast and easy and gave satisfying results it was chosen over other indicators.

Even worse than a bad mesh quality is an invalid mesh. A two-dimensional example for such a situation is illustrated in figure 3.4. Luckily, such a problem can easily be detected by comparing the cell orientation before and after the collapse because it flips in such cases. For tetrahedral cells another problem similar to this might occur in form of negative cell volumes, even though the orientation of the cell

might not be affected in those cases. This is why the before mentioned quality check is not done using the absolute value of the ratio, hence exposing negative volumes in this check.

The resulting mesh quality check is summarized in algorithm 3.5.

Algorithm 3.5 Check mesh quality

Let v_D be the deleted vertex and $\mathcal{T}_D \subset \mathcal{T}$ the set of all elements adjacent to v_D.
for all $K \in \mathcal{T}_D$ **do**
  if orientation of K changed **then**
    Reject coarsening.
  else if volume(K)/diameter$(K) <$ TOL **then**
    Reject coarsening.
  end if
end for
Accept coarsening.

Whenever this check reveals too poor a mesh quality, the previously applied edge collapse is revoked and the coarsening continues with the next cell (see next section).

3.2.6 Complete coarsening algorithm

In the previous section it was explained how to select edges and vertices for coarsening and perform the edge collapse, allowing to put together the whole algorithm 3.6.

The outer loop repeats as long as changes to the mesh happen. Although the inner loop alone might suffice to obtain some coarsening, elements are only removed from $\mathcal{C}$ when the quality criterion described in chapter 3.2.5 is fulfilled. However, due to mesh changes when coarsening neighboring cells, the edge collapse of previously failed attempts can be successful in new tries, being the reason for the additional outer loop.

Algorithm 3.6 Complete coarsening algorithm

Given a mesh $\mathcal{T}$ and a set $\mathcal{C} \subset \mathcal{T}$ containing elements marked for coarsening.
Find the independent set $\mathcal{S}$. ▷ Algorithm 3.2
repeat
  for all $K \in \mathcal{C}$ **do**
    Select edge e for collapse. ▷ Algorithm 3.3
    Select $v_D, v_R \in e$. ▷ Algorithm 3.4
    Edge collapse. ▷ Algorithm 3.1
    Check quality of changed elements. ▷ Algorithm 3.5
  end for
until number of elements in $\mathcal{T}$ did not change.

3.3 Parallel edge collapse

When running mesh coarsening in parallel, problems can occur in cases where the mesh entities necessary for the coarsening of a cell belong to more than one process. In chapter 3.2.1 it was mentioned that edge collapse is equivalent to a retriangulation of the polyhedron formed by all mesh entities connected to the pulled vertex. Therefore, edge collapse is possible whenever the complete polyhedron belongs to a single process, whereas it is not possible if it is distributed among multiple processing elements.

This means mesh coarsening inside the process boundaries is possible without any changes to algorithm 3.6. A special treatment is only necessary if the vertex selected in algorithm 3.4 is on a process boundary and hence shared by two or more processes.

Two different principles to tackle this problem are imaginable: either to distribute the coarsening of a cell itself among all involved processes, for example with a voting scheme similar to the one used for refinement by Jansson [19], or to perform coarsening locally in a single process and move the necessary mesh entities between processing elements. The first approach could turn out quite complex, with many different cases for 2D or 3D and possible conflicts to resolve. The second approach, however, has the advantage that it allows to use the same coarsening technique as in the serial case and simply add special treatment for above mentioned cases. Among others, this method was chosen by Cougny and Shephard [13] and Alauzet et al. [16] and the method used here is inspired by those implementations.

Since coarsening inside the process boundaries is possible without any modifications, this is done for all selected cells in $\mathcal{C}$ (see algorithm 3.6). The only thing to be modified for that is algorithm 3.4: Here it is necessary to check if the selected vertex lies on the process boundary. In such a case an edge collapse is not possible, the vertex is marked for migration and coarsening of this cell is postponed, leading to the modified algorithm 3.7.

Once all cells inside the boundary are coarsened, i. e., a full execution of the inner loop of algorithm 3.6 is finished, a migration step is executed to exchange the required mesh entities for the previously postponed cells. Afterwards, the set $\mathcal{C}$ of cells marked for coarsening is updated with the newly received cells and the inner loop is executed again. These modifications lead to algorithm 3.8.

3.3.1 Migration of mesh entities

For the migration of mesh entities that are required for an edge collapse, all neighboring processes that own some of those entities have to be instructed to move those to the requesting process. Each mesh entity is uniquely owned by a single processing element but may be shared among more than one process. For example, let two vertices v_A, v_B be on the boundary between processes A and B, with v_A belonging to process A and v_B belonging to process B (see figure 3.5). Although both vertices are uniquely affiliated with one process, they are shared among both

Algorithm 3.7 Select vertex for collapse in parallel

Let $\mathcal{S}$ be the independent set and e the edge to be collapsed.
Let v_1, v_2 be the endpoints of e.
if $v_1 \in \mathcal{S}$ **then**
    $v_D = v_2$ and $v_R = v_1$.
else if $v_2 \in \mathcal{S}$ **then**
    $v_D = v_1$ and $v_R = v_2$.
else
    $v_D = \max(v_1, v_2)$.
    $v_R = \min(v_1, v_2)$.
end if
if v_D is on a process boundary **then**
    Mark v_D for migration.
    Postpone coarsening of the cell.
end if

Algorithm 3.8 Parallel coarsening algorithm

Given a mesh $\mathcal{T}$ and a set $\mathcal{C} \subset \mathcal{T}$ containing elements marked for coarsening.
Let $\mathcal{M}_v, \mathcal{M}_c$ be the empty sets of vertices and cells marked for migration.
Find the independent set $\mathcal{S}$. ▷ Algorithm 3.2
repeat
    for all $K \in \mathcal{C}$ **do**
        Select edge e for collapse. ▷ Algorithm 3.3
        Select $v_D, v_R \in e$. ▷ Algorithm 3.7
        Edge collapse. ▷ Algorithm 3.1
        Check quality of changed elements. ▷ Algorithm 3.5
    end for
    Migrate mesh entities according to $\mathcal{M}_v$ and $\mathcal{M}_c$. ▷ Algorithm 3.9
until number of elements in $\mathcal{T}$ did not change and no migration happened.

of them since both vertices are also entities of the elements K_A and K_B. Now let process A attempt to coarsen K_A by collapsing v_B onto v_A. This is not possible yet, since neither v_B nor all the elements adjacent to v_B belong to A. However, migrating v_B and all elements adjacent to v_B to A shifts the process boundary and lets v_B appear as an inner node, making coarsening as simple as before.

Due to implementational details, the requests for the elements adjacent to v_B cannot be sent directly to all sharing processes. Only the owner of a vertex knows all sharing processes, making a 3-step communication necessary: First each process sends all vertices, that have been postponed during the vertex selection, as requests to the respective owners. These pass each request on to all processes that share the requested vertex, together with the rank of the requesting process. Finally the requested entities can be migrated to the requesting vertices. The complete migration scheme is outlined in algorithm 3.9.

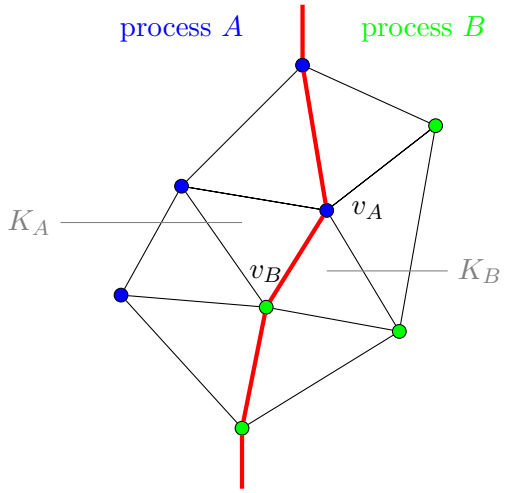


Figure 3.5. Example for process affiliation of vertices at process boundaries: each vertex is owned by a single process (indicated by coloring), but is shared amongst neighboring processes to form elements. The red line shows the process boundary.

Algorithm 3.9 Migration of mesh entities

Let $\mathcal{M}_v, \mathcal{M}_c \subset \mathcal{T}$ be the sets of vertices and cells marked for migration.

1: Send requests for all $v \in \mathcal{M}_v$ to their respective owners.
2: Let $\mathcal{R}_v$ be the set of all received vertex requests.
3: Send requests for all $v \in \mathcal{R}_v$ to each process that shares v.
4: Let $\mathcal{R}_c$ be the set of all cells adjacent to received vertex requests.
5: Migrate all $K \in \mathcal{R}_c$ to the requesting processes.

Resolving migration conflicts

During the migration procedure conflicts will appear between requests. For example a third process C might also share vertex v_B in figure 3.5 and might attempt to collapse v_B onto a different vertex, also resulting in requests for the elements adjacent to v_B. In this case an explicit criterion is necessary to decide which process should receive the requested elements.

In principle, the conflict can arise at two stages during algorithm 3.9: Either in the form of multiple requests for the same vertex, as described in the example, or in the form of overlapping polyhedrons from different requests. Either way, migration is always done in favor of the process with lower rank, leading to elements on process boundaries being shifted towards lower rank processes (see figure 3.6).

In the case of multiple requests for the same vertex the conflict gets resolved fairly easy: the owning process selects from the requesting processes the one with lowest rank and passes the requests on to the sharing processing elements with respect to this process.

If polyhedrons overlap, this will only become apparent after processing elements

received the requests from the vertex owners. Then cells are marked multiple times for migration to different processes. In this case all elements of the polyhedron are marked for migration to the process with lowest rank from the conflicting cell.

Conserving coarsening markers and independent set

After the migration of mesh entities, algorithm 3.8 still needs the independent set $\mathcal{S}$ and the set $\mathcal{C}$ of cells marked for coarsening. Hence these must also be migrated together with the requested entities.

3.3.2 Global termination detection

The amount of cells selected for coarsening will differ between processes, making it necessary to determine the termination of the coarsening algorithm globally. In extreme cases, one process might not have a single cell marked for coarsening, while another process has thousands. Therefore, some synchronization has to ensure that all processes are informed when coarsening is finished on all other processes. For this, the algorithm of Cougny and Shephard [13] added the absence of migration as second termination criterion to the outermost loop. How this global termination detection was implemented is outlined in chapter 5.2.5.

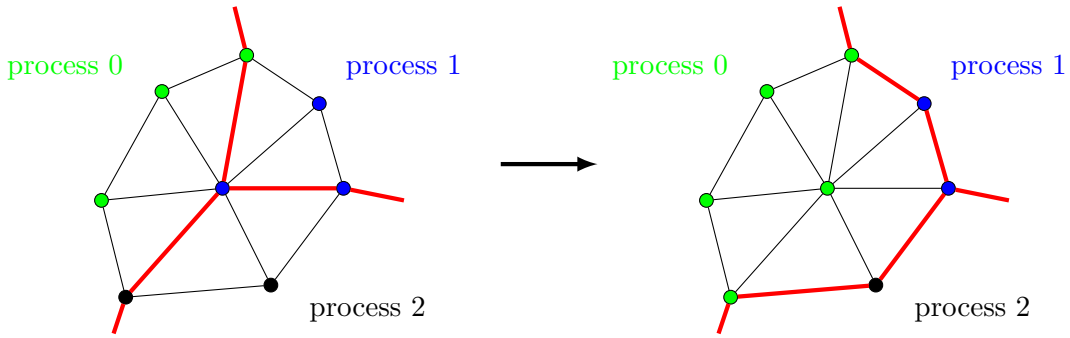


Figure 3.6. Example for the shift of elements on process boundaries (indicated by the red line) towards processing elements with lower ranks.

Part II

Implementation

Chapter 4

FEniCS

4.1 The FEniCS project

FEniCS [20] is an open source software project that attempts to automate the solution of problems in scientific computing. Often applications in scientific computing are developed specific for a certain problem, making it necessary to start from scratch every time. FEniCS, however, provides functionalities that allow to solve a broad range of problems based on the finite element method. It allows to specify the variational form, boundary conditions and discretization method on a high level close to the mathematical notation. A compiler then generates low-level code from those statements, allowing to solve the problem in an efficient way.

This section gives a short introduction into the core components included in the project and an overview over the different flavors of FEniCS.

4.1.1 Components

The FEniCS project is a collection of different software components. The main interface is DOLFIN [21], a C++ library that contains data structures, mesh representations, assembly routines and provides interfaces to linear algebra backends.

FFC is the **F**EniCS **F**orm **C**ompiler [22] and responsible for the transformation of the high level mathematical formulation into C++ code. The generation of arbitrary Lagrangian elements is handled by FIAT (**Fi**nite element **A**utomatic **T**abulator) [23] and a framework for finite element assembly is provided by UFC [24]. Further components are a just-in-time compiler (Instant) for the usage with Python and the Unified Form Language UFL.

Closely associated but strictly speaking not a part of FEniCS is the unified continuum mechanics solver Unicorn [5].

4.1.2 Branches

The main branch[1] of FEniCS is targeted on average users, providing precompiled packages for all major operating systems. As of this writing, the most recent released version was 1.2.0.

Based on DOLFIN 0.8.0, a branch specialized for computing on distributed memory machines was developed, called DOLFIN HPC[2] [3]. Amongst others it is supplemented with mesh partitioning, parallel mesh refinement, checkpointing and load balancing to suit the needs when computing on large scale distributed memory machines.

The implementatory work in this project was done as a part of DOLFIN HPC, with a code basis close to the upcoming release 0.8.4-hpc.

4.2 DOLFIN HPC

The library is implemented in C++ with an object oriented design. Parallelization is done using the Message Passing Interface (MPI), hence it follows the Single Program Multiple Data (SPMD) paradigm.

As mentioned before, DOLFIN is responsible for providing most of the necessary functionalities for finite element computations and includes many different parts, for example the mesh library, assembly routines for matrices, interfaces to linear algebra backends (like PETSc[3] or JANPACK[4]), input/output-routines, load balancer (using ParMETIS[5]) and checkpointing-functionalities. However, since the mesh library is the only part of major interest in this thesis, details will only be given about this.

4.2.1 Mesh-library

The mesh representation in DOLFIN is based on vertices and cells, generating all other mesh entities from those two. It is always fully distributed among the processing elements, meaning each process has only its local part of the mesh together with surrounding ghost layers. Mesh objects and derived objects (like functions) are always defined in terms of the local mesh. The global connectivity is given as an additional object, that contributes ownership information, a mapping from local to global entity indices and other meta information.

Modifications to the mesh are not directly possible, instead a new mesh object has to be created and filled with both kept and changed mesh entities. This constraint has serious impact on the performance of mesh adaption algorithms like refinement or coarsening. Due to this fact a dynamic mesh was implemented by Jansson [3] that allows to convert an existing mesh into a dynamic mesh on which

[1] `http://www.fenicsproject.org`
[2] `https://www.launchpad.net/unicorn`
[3] `http://www.mcs.anl.gov/petsc/`
[4] `http://www.csc.kth.se/~njansson/janpack/`
[5] `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

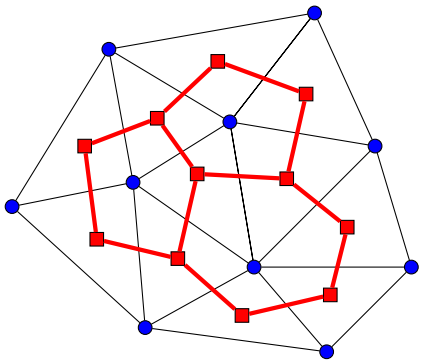


Figure 4.1. A triangular mesh and its corresponding dual graph (red). Each cell is represented by a node in the graph.

adaption routines can be executed much more efficiently. But other operations, like assembly of FEM matrices, are not possible on the dynamic mesh. Therefore, after completion of the mesh adaption algorithms, the dynamic mesh is converted back to a regular mesh object. This existing dynamic mesh implementation had only been available locally in the mesh refinement and was extended and made usable to the whole library as part of this project.

Further details about the mesh library and its parallelization aspects can be found in the thesis of Jansson [3, 19].

4.2.2 Load-balancing

A key aspect to scalability in parallel computing is an equal distribution of the workload among the processing elements. Work-imbalance causes processors to wait idle while others are completing tasks and thus the total runtime increases in two ways: Assuming that the runtime depends (linearly) on the number of work items, then (i) the completion of the overall task takes longer due to waiting for completion of the larger workload on one process and (ii) the additional work items could have been completed by an idle process simultaneously. Hence load balancing is crucial for efficient computing.

Load balancing in DOLFIN HPC is carried out by partitioning the domain among the available processes. To find those partitions the mesh is converted into its dual graph (see fig. 4.1), in which every node of the graph represents a cell of the mesh and graph edges correspond to shared facets. This graph is divided into partitions using the multilevel k-way partitioning algorithm [25] of ParMETIS[6]: Every node and edge are assigned a certain weight corresponding to the associated workload and the overall goal is to divide the graph into k partitions with same workload such that the number of cut edges is minimal.

[6] `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`

For the assembly procedures of finite element computations the minimal number of cut edges is particularly relevant since each edge requires communication with the neighboring process to determine the neighbors contribution. The workload, however, is the same for every cell in this case. In contrast, mesh refinement only causes workload on cells that are to be refined and the neighboring cells, which are also refined to remove hanging nodes. To apply load balancing prior to the refinement procedure, a dry-run is executed to determine the weights for the load balancer and subsequently obtain the desired partitions. Further insights into DOLFIN HPC's load balancer and its application for mesh refinement have been described by Jansson [19].

4.3 Unicorn

Unicorn[2] is a collection of continuum mechanics solvers built on top of DOLFIN HPC. It offers adaptive solvers for the incompressible Navier-Stokes equations (2.11) and fluid-structure interaction problems and was described in detail by Hoffman et al. [5]. It is not a part of the FEniCS-project itself but strongly connected to it.

The incompressible Navier-Stokes solver of Unicorn has been used in the application of chapter 8.

Chapter 5

Mesh coarsening in DOLFIN HPC

The implementation of the parallel mesh coarsening algorithm 3.8 in DOLFIN HPC had to be done with respect to the specialties in its mesh library. A straightforward but not very mature serial implementation of a mesh coarsening scheme was already available in DOLFIN. This existing algorithm and the changes made to it are discussed in the upcoming section before the remainder of this chapter lists optimization approaches together with their individual impact on the performance of the algorithm.

5.1 From serial to parallel coarsening

5.1.1 The existing implementation

DOLFIN HPC had already a serial implementation of a mesh coarsening scheme based on edge collapse. The algorithm was only serial but already quite close to the approach described in section 3.2, only without the additional independent set constraint for proper vertex distribution. To make coarsening also available to large scale problems with a reasonable total execution time, this parallel implementation was done.

The serial algorithm consisted also of an iteration over the cells marked for coarsening plus an additional outer loop that terminated when the total number of cells didn't decrease anymore (identical to the loops in algorithm 3.6). For each cell from the coarsening list the following steps were executed: (i) select shortest edge for collapse, (ii) select vertex with lower rank for deletion, (iii) move all cells that are not adjacent to the deleted vertex to a new mesh, (iv) replace the deleted vertex by the remaining vertex in all remaining cells adjacent to the deleted vertex, (v) check the quality of the regenerated cells and (vi) dismiss the old mesh if the quality is ok, otherwise dismiss the new mesh.

5.1.2 Changes for parallelization

Besides cleaning up the existing code, the first modification dealt with the problem of the vertex distribution (mentioned in chapter 3.2.2) for which the independent set algorithm was added. After having implemented this modification and after adding all vertices on process boundaries to the independent set, such that coarsening was restricted to the interior of processes, execution was already possible in parallel.

The next step was to add the migration procedures outlined in chapter 3.3. To logically split between the coarsening algorithm itself, bookkeeping and communication functionalities, an additional class `CoarseningManager` was introduced. It manages all sets ($\mathcal{C}, \mathcal{M}_v, \mathcal{M}_c, \mathcal{S}$, see algorithm 3.8) and provides the migration algorithm 3.9.

Although the changes sound marginal, combined with the optimizations described in the upcoming sections it was equivalent to completely rewriting the coarsening procedures.

5.2 Performance optimization

To increase performance and scalability of the algorithm a few optimization approaches were implemented and tested. They are summarized in the following including short analyses of their impact.

The impact of the optimizations was always tested for three-dimensional testcases, since the computation time for tetrahedral meshes is drastically higher than for triangular meshes. Additionally, in most cases DOLFIN HPC is used, simulations are carried out in 3D and consequently such scenarios are far more relevant. The measurements for each approach were done using one or more of the following scenarios:

- Uniform coarsening of the entire mesh, i. e., all cells are marked for coarsening;
- coarsening of cells within a single restricted region in the domain;
- coarsening of a fixed percentage on each process;
- coarsening with respect to error indicators from a fluid simulation.

Each of them gave insight into different aspects of the algorithm, which is explained in more detail in the upcoming sections. The meshes used are listed in appendix A. All measurements in this section were done on a single node with an Intel Core i7-3770 CPU (4 cores, 8 treads) with 16 GBytes of RAM. The MPI implementation of OpenMPI in version 1.4.3 was used together with g++ 4.6.3. Larger setups and results are discussed in chapter 7.

5.2.1 Dealing with DOLFIN's mesh datastructures

In DOLFIN, meshes are defined in terms of vertices and cells (triangles in 2D and tetrahedrons in 3D) – all remaining topological entities (like edges or faces)

are computed from those two entity types. Mesh objects are optimized such that assembly routines are as fast as possible, resulting in the static implementation already described in 4.2.1 which does not allow any direct modifications to the mesh. Instead, new mesh objects have to be created every time and filled with the modified mesh. Consequently, improving the datastructures was the first thing to do.

Avoiding topology computations

To instantiate a new mesh object leads to a loss of all previously computed entities besides vertices and cells, which subsequently have to be recomputed. Profiling the application revealed that most of the time was spent on this repeated derivation of connectivity information.

Naturally the first optimization approach was to avoid the recomputation of the connectivity. For that, all operations that rely on those connectivity information (like edge lengths) were replaced by own implementations that would not require them. The impact of these changes was tested using uniform coarsening of a small mesh: a sphere that is built up of 9312 cells and 2129 vertices. Uniform coarsening reduced the number of cells by 8-10%, depending on the shape of the cells and the order of iteration.

This optimization mainly tackled the local serial part of the algorithm and since the mesh is rather small in terms of the number of mesh entities, it was possible to test it on a single process without too long waiting times. By simply avoiding the computation of the mesh topology, runtime dropped by approximately 60% compared to the original implementation, as visible in table 5.1. The modification involved some minor changes that influenced the iteration order and consequently the number of actually coarsened cells did also change in a small range. To include this in the time comparison, the runtime per deleted vertex is given as a better indicator of the real performance gain. It was determined by dividing the total execution time by the number of performed collapses. Since every removed vertex corresponds to a successful edge collapse, this is a rough estimate of the average time spent on each successful collapse. This quantity is also used in following runtime comparisons. Though the ratio is a little bit smaller there, it still gives approximately 60% performance gain.

Repeating the same measurements with two processes and hence introducing some communication overhead lowered the percentage of the performance gain.

	total execution time [s] / per collapse [ms]		
# processes	original	no topology	dynamic mesh
1	21.74 / 247.0	8.47 / 99.65	0.57 / 5.00
2	15.17 / 194.5	9.11 / 111.1	1.65 / 13.98

Table 5.1. Runtime comparison for uniform coarsening of a sphere with and without optimizations to the mesh datastructures.

This can easily be explained with the fact that in this case the optimized part of the algorithm had a smaller stake of the total runtime due to the now necessary migration, and thus the overall performance gain was reduced. But the obtained improvement is still in the same range as in the serial case.

Using dynamic meshes

Not recomputing the connectivity of the mesh only widened the actual bottleneck but did not yet eliminate it. Still a lot of time was spent on allocation of memory for the second mesh and copying the entities for each collapse. This was overcome by using the dynamic mesh introduced in 4.2.1. The dynamic mesh only provides the basic entities and stores the fundamental data of the mesh, hence does not allow to use assembly routines or other operations on it. But it provides in-place modification procedures and hence entities can efficiently be added, removed or modified, making the previously needed allocation of a second mesh and copy operations obsolete. Import- and export routines provide conversion to and from conventional mesh objects. These necessary conversion are included in the time measurements.

For the uniform coarsening of the sphere on one processor this resulted in another reduction of the total runtime by more than 93% compared to the optimized version of the previous section and even more than 97% compared to the original implementation. Naturally the gain was again smaller for two processor due to the same reasons as before. The numbers are also included in table 5.1.

5.2.2 Initial load balancing

In applications of mesh coarsening the regions marked for edge collapse can appear in small restricted areas, e. g., only in the wake of an object in a fluid simulation. Subsequently, only processing elements in this local regions are involved in the coarsening process while the other nodes remain idle. As already mentioned in chapter 4.2.2, equal distribution of the workload is usually crucial to scalability and consequently should be applied prior to the coarsening process.

Unfortunately, the greedy algorithm used to find the independent set (see section 3.2.2) will give different results after repartitioning the mesh because the order of vertices might be different, and hence it is not possible to predict exactly which cells will be involved with each edge collapse. Also, depending on the order of execution of the collapses, the selected vertices and involved cells can differ. Therefore, it is not feasible to perform a dry run to obtain the weights for mesh coarsening, as it is done for the refinement. Instead, it turned out to be best for practical cases to weigh marked cells and their neighbors (as they are possibly involved in the edge collapse) against all other cells with weights 2:1.

Another problem is that performing load balancing too aggressively might distribute the marked cells evenly across all processes, resulting in very small tasks on each process. This might be advantageous for other algorithms but, as mentioned before, marked regions are usually restricted but not necessarily distributed

among the whole domain and too aggressive load balancing can put process boundaries inside marked regions. Subsequently extensive communication between the neighboring processes would be necessary, rendering the gain from the equal work distribution useless due to the increased communication efforts. Additionally, the load balancing process does take some time itself, hence the runtime gain from its application should more than outweigh its own cost.

Of course, load balancing can only improve the runtime when an initial imbalance is given, therefore uniform coarsening would not benefit from it. But in typical applications coarsening is applied in restricted areas of the simulation domain which are only in a subset of all processing elements. To verify this, first cells in a spherical region behind the landing gear in the landing gear mesh (see figure 5.1) were marked for coarsening and the runtime was measured for different numbers of processes. Secondly, the coarsening procedure was applied with respect to the error indicators after the sixth adaptive iteration of the fluid simulation around a cylinder, to measure the impact of load balancing in a real application. The simulation setup is explained in more detail in chapter 8. Thirdly, to investigate the additional workload from load balancing in an initially balanced setup, uniform coarsening of the landing gear mesh was done.

Unfortunately, the measurements in table 5.2 state clearly that load balancing did not improve the overall runtime. The very opposite was true: the communication overhead from interprocess communication was by far larger than the performance gain from the reduced workload per process. For the first, rather extreme case of a small, very restricted marked area behind the landing gear, applying load balancing was always 3-6% slower than without. A look on the number of processors involved with coarsening revealed that in fact not more processes were used. Hence it is probable that the increased runtime was due to the additional load balancing step. However, this is not entirely certain since the gap was only marginally when applying uniform coarsening. But this could still be due to the fact that in the latter case almost no migration of cells happened during the load balancing step,

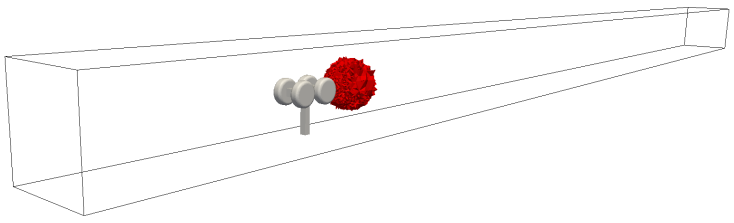


Figure 5.1. The spherical region marked for coarsening behind the landing gear.

	spherical region behind landing gear ($\approx 2.7\%$ cells marked)		
	2 proc.	4 proc.	8 proc.
no load-balancing	90.41 / 118.0 (1)	85.49 / 112.3 (2)	73.86 / 102.0 (3)
initial load-balancing	95.53 / 124.7 (1)	88.28 / 116.0 (2)	76.43 / 108.6 (3)

	cylinder, error indicators, 8 proc.		landing gear, uniform
	10% marked	50% marked	8 proc.
no load-balancing	63.9 / 37.70 (2)	314 / 16.40 (8)	368.96 / 21.13 (8)
initial load-balancing	177 / 183.6 (5)	407 / 20.79 (8)	368.98 / 21.13 (8)

Table 5.2. Measurements for load-balancings runtime impact. Each entry shows: total execution time [s] / per collapse [ms] (# processes involved).

while in the first case more mesh entities changed processes.

However, when looking at the cylinder testcase results were even worse: the total execution time increased drastically with load balancing enabled, especially when a smaller percentage of cells was selected for coarsening, although load balancing was obviously effective due the considerably smaller number of idle processes.

The results were disappointing but not surprising: profiling showed that after incorporating the better datastructures, the local coarsening computations did no longer take a major runtime portion, even for large meshes. For example in the cylinder testcase, 50% of the time was spent in the migration phase. Also, since these measurements have been done on a single node, results might turn out even worse when interprocess communication involves a real network connection.

Experiments with different weights for marked and neighboring cells, different ratios between marked and non-marked cells etc. showed small improvements for some testcases but on average the above described approach for choosing the weights yielded the best results. Therefore, it is improbable that the poor outcome of load balancing is due to badly chosen weights.

With respect to these findings load balancing is not applied by default but subject to the user who can use the prepared functionalities prior to the call of the coarsening algorithm.

5.2.3 Improving migration

The major difference between serial and parallel mesh coarsening was the introduced migration routine. Ideally, mesh entities should be moved between processes such that the coarsening result is the same as if the subsequently coarsened cells were located in the interior of a process. Of course even then some artifacts from process boundaries might be visible in the resulting mesh since these cells are always coarsened after the interior cells.

But the migration procedure should also be as fast as possible and hence move only as few cells as necessary to minimize communication costs. This is why the migration routine only transfers the cells in the polyhedron of the requested vertex.

	uniform coarsening of landing gear mesh			cylinder, error indicators
	2 proc.	4 proc.	8 proc.	10% marked, 8 proc.
original	248.8 / 14.1	245.5 / 14.0	369.0 / 21.13	63.9 / 37.7
improved	60.27 / 3.41	51.99 / 2.97	61.22 / 3.51	29.3 / 16.9

Table 5.3. Measurements for the impact of migration improvements. Each entry shows: total execution time [s] / per collapse [ms].

But the results of the previous section revealed that most of the time was still lost in the migration phase and hence optimization of this phase could be promising.

Once the requests are exchanged and the target process for each migration is known (see algorithm 3.9), the distribution is executed. This step was implemented using the existing functionalities in DOLFIN HPC for two reasons: (i) it saved the effort of reimplementing already available communication patterns and (ii) the existing routines are used widely in applications and hence tested thoroughly. However, the existing distribution relies on regular mesh objects and thus the dynamic mesh has to be exported first, is distributed then and reimported to a dynamic mesh afterwards. Consequently, the focus for the optimization lied primarily on the steps executed before and after the existing distribution routines.

Avoiding topology computations

Detailed profiling revealed once again that a major portion of the time was spent in the computation of topology information. Further analysis of the implementation of involved routines showed that this could be avoided, allowing for the performance gain visible in table 5.3.

Once again, this was tested with uniform coarsening of the landing gear mesh which had a large communication portion due to coarsening at all process boundaries. Additionally, to measure the impact in the more likely scenario of restricted coarsening regions, the cylinder testcase, where coarsening happens w. r. t. error indicators, was tested.

The results showed the expected performance gain considering the large runtime portion the migration step originally had: in the communication rich uniform coarsening more than 75% improvement were achieved. The cylinder testcase with less communicaton still improved by approximately 45%. Once again, it should be noted that the difference would be less on a machine involving a real network interconnection, since the communication patterns themselves, which are kept the same, would then take a larger percentage of the total runtime.

Limiting the number of coarsening attempts

The analysis of the algorithm in many testcases showed that a major problem of the edge collapse scheme was the quality of the resulting mesh, as already discussed in chapter 3.2.5. A very large number of edge collapses had to be undone due to

	uniform coarsening of landing gear mesh			cylinder, error indicators
	2 proc.	4 proc.	8 proc.	10% marked, 8 proc.
unlimited	60.27 / 3.41 (11.8)	51.99 / 2.97 (11.7)	61.22 / 3.51 (11.6)	29.3 / 16.9 (8.74)
limited	54.23 / 3.07 (11.8)	37.78 / 2.16 (11.7)	37.43 / 2.15 (11.6)	28.5 / 17.0 (8.45)

Table 5.4. Measurements for the impact of limiting coarsening attempts to 10. Each entry shows: total execution time [s] / per collapse [ms] (coarsening efficiency in %). The coarsening effectivity is determined as ratio between the number of coarsened cells and initially marked cells.

a flipped orientation or too poor quality of the modified cells. This is a known drawback that one has to pay for the simplicity of the algorithm and that is why most publications combined the edge collapse with some kind of smoothing scheme afterwards or collapsed both vertices onto the edge midpoint instead of one onto another [13, 16].

DOLFIN HPC provides a Lagrangian mesh smoothing routine but its results were not very satisfying. Therefore, the quality criterion in the mesh check (algorithm 3.5) had to be chosen rather conservative resulting in the large number of coarsening failures. The large number of issues that arose during the implementation of this simple edge collapse scheme did not allow for investigation of another edge collapse algorithm.

Instead, a counter was introduced for each marked cell, which increased with every failed attempt. After a number of failures it is improbable that the cell will be coarsened successfully eventually. Therefore it can be unmarked and skipped in following iterations.

The introduced counter allowed for another modification in the vertex selection algorithm 3.4. When both endpoints of the edge are available for collapse, until now always the vertex with larger index was selected. But with the counter it is possible to alternate between points in consecutive coarsening attempts, actually making the coarsening successful in few cases. But these changes did neither result in a measurable performance gain nor in a large increase of the coarsening efficiency. Especially the latter is far more dependent on the quality and shape of the initial mesh than such tweaks.

Further analysis of the migration paths showed that after the first few migration steps cells were only moved to lower rank processes without being able to be coarsened there either. Hence a lot of time was spent in useless communication which also increased the imbalance of cells among processing elements. Consequently, the number of attempts was limited for all cells, now also counting coarsening tries which were postponed due to missing entities as attempts. Experiments showed that picking 10 as the maximum number of attempts and hence migration was both effective in terms of performance improvements and still on the side of conservatism regarding the number of skipped cells. Table 5.4 shows the impact of the limit

	spherical region behind landing gear ($\approx 2.7\%$ cells marked)		
	2 proc.	4 proc.	8 proc.
no load-balancing	10.52 / 13.72 (1)	10.64 / 13.89 (2)	10.16 / 13.94 (3)
initial load-balancing	15.69 / 20.46 (1)	13.79 / 18.00 (2)	12.87 / 17.61 (2)

	cylinder, error indicators, 8 proc.		landing gear, uniform
	10% marked	50% marked	8 proc.
no load-balancing	28.5 / 16.95 (3)	55 / 2.85 (8)	37.43 / 2.15 (8)
initial load-balancing	37.8 / 39.83 (5)	46.9 / 2.42 (8)	37.55 / 2.15 (8)

Table 5.5. Measurements for load-balancings runtime impact after improving the migration phase. Each entry shows: total execution time [s] / per collapse [ms] (# processes involved).

in terms of runtime and coarsening efficiency. Further remarks about coarsening efficiency itself can be found in chapter 6.

5.2.4 Load balancing revised

After improving the migration step the effort for exchanging entities between processes was reduced significantly and hence load balancing was worth another try. Therefore, the same measurements as before were repeated with the outcome showed in table 5.5

The results basically confirmed the previous observation that the serial part of the algorithm, the actual coarsening of cells, doesn't take a portion large enough of the whole to benefit from load balancing.

5.2.5 A good termination strategy

This final optimization approach did not specifically aim on perfomance improvements but is a detail of the way the algorithm is implemented and hence was found worth mentioning.

When comparing algorithm 3.6 and its parallel version (algorithm 3.8) two major differences stick out: (i) the migration and the associated modifications to the vertex selection and (ii) the changed termination criterion of the outer loop. As explained in section 3.2.6 in the serial algorithm, the outer loop is necessary to possibly coarsen cells that previously failed due to inverted orientation or poor mesh quality. But if the number of elements in the mesh did not change anymore after one iteration of the outer loop, elements won't be coarsened in any following iteration either and hence the algorithm terminates.

In a parallel setting, however, coarsening of a cell can also fail due to missing mesh entities from neighboring processes. Hence, even after an iteration of the outer loop that did not coarsen any cells, the next iteration might still be able to perform successful edge collapses on elements that previously didn't belong to the process.

Therefore, the absence of migration was added to the termination criteria because when no cells have been coarsened previously and the process affiliation of elements stays the same nothing will happen in the following iterations either.

In a first, naive implementation the absence of migration was determined by simply checking globally for the existence of requests at all. This check could be done using the already determined maximum number of requested entities to allocate sufficiently large send- and receive-buffers. But due to the collision detection, the actually transferred number of entities in the migration could differ from the amount originally requested, possibly resulting in continuous requests without any migration happening. In the worst case this led to infinite loops.

Hence, it would take an additional synchronization step to detect whether migration really took place. To avoid this extra communication step, another approach was chosen: The number of cells migrated is saved locally and exchanged within the before mentioned global communication step in the next migration. This saves the effort of the extra synchronization effort in all migration steps in exchange for possibly one extra execution of the inner loop in algorithm 3.8.

In small to average sized coarsening examples this solution did not give any measurable improvement compared to an extra synchronization step but it might turn out to be advantageous for very large simulation setups.

Part III

Results and application

Chapter 6

Effectivity and efficiency of mesh coarsening

The most important question when talking about results of a mesh coarsening algorithm is whether the produced mesh is coarser than the initial one, i. e., does it have a lower resolution? To investigate this aspect, first the chosen quantities to measure the effectivity and efficiency of coarsening are introduced before the outcome of the implemented algorithm is discussed with the help of some examples.

6.1 Quantifying effectivity and efficiency

Whether a mesh has a high or low resolution is always specific to the associated problem and the chosen numerical method. If the sought-after solution to the problem has a steep gradient, e. g., a shock wave, even a very fine mesh might be considered to have too low a resolution if it can't resolve the gradient properly. However, the very same mesh could be considered to have a sufficient resolution if the solution does not have such a shock wave or if a different numerical method would be used that gives more accurate results.

Therefore, in most cases some kind of a posteriori error estimate is used to judge whether a gained numerical solution is accurate. If it is not accurate enough, the mesh resolution has to be increased. In adaptive simulation schemes (see section 2.3) the mesh is then refined in regions with large errors until the desired accuracy is reached. But with respect to the general applicability of DOLFIN to finite element computations, it does not make sense to judge the quality of the coarsening method based on some error estimate for a certain problem setup. Rather, it is of interest if the mesh resolution did change in a more general applicable way.

For two geometrical dimension, Miller et al. [17] defined coarsening on the basis of edge lengths in the mesh:

Definition: edge-length function el_M Let M be a mesh over a two-dimensional domain Ω. For each $x \in \Omega$, $el_M(x)$ is defined to be the length of the longest edge

of all the mesh elements that contain x.

It is to be noted that only points that coincide with a mesh vertex or lie on an edge are contained in more than one element. With this edge length function coarsening was then defined as:

Definition: coarsening (Miller et. al) A *coarsening* $M' = (\mathcal{V}', \mathcal{E}', \mathcal{B})$ of a mesh $M = (\mathcal{V}, \mathcal{E}, \mathcal{B})$ is a mesh whose edge-length function $el_{M'}$ is point-wise larger than el_M but still conforms to the same domain. The mesh M is than a *refining* of M'.

$\mathcal{V}$ and $\mathcal{E}$ are the sets of vertices and edges of the mesh and $\mathcal{B}$ its boundary description. To apply this to a general two- or three-dimensional mesh, a different definition is necessary. Since the collapse of boundary vertices is forbidden by default, it is ensured that the coarsened mesh conforms to the same domain as the fine mesh. However, the problem with above definition is that it is defined on a mesh entity which is somewhere between the highest and lowest topological dimension. Topological dimension 0 corresponds to vertices, 1 to edges and 2 to elements in 2D. But when going to three geometrical dimensions, two topological dimensions lie between vertices and elements: edges and faces. Hence it is no longer clear which one is to be used.

Furthermore, following the general comprehension of *pointwise*, above definition requires that the edge-length function is larger in every point of the mesh. But this is not desirable for a coarsening scheme that should act only locally, as described before. This is why henceforth coarsening is defined using the *Lebesgue measure*:

Definition: volume function v_M Let M be a mesh over an n-dimensional domain Ω. For each $x \in \Omega$, $v_M(x)$ is defined to be the largest *Lebesgue measure* of all the mesh elements that contain x.

The *Lebesgue measure* is the generalization of volume to an n-dimensional Euclidian space, i. e., for $n = 1, 2, 3$ it coincides with length, area and volume, respectively. For the sake of simplicity, in the following when talking about the volume it refers to the Lebesgue measure to make the statements true for two- and three-dimensional cases and keep the text readable. Using above volume function, coarsening is henceforth defined as:

Definition: coarsening A coarsening $M' = (\mathcal{V}', \mathcal{E}', \mathcal{B})$ of a mesh $M = (\mathcal{V}, \mathcal{E}, \mathcal{B})$ is a mesh whose volume function $v_{M'}$ is point-wise larger or equal than v_M and $\exists x \in \Omega : v_{M'} > v_M$ but still conforms to the same domain.

With this definition, a mesh is now considered to be coarsened when at least at one point the volume of the corresponding cell is larger than the volume of the corresponding cell in the fine mesh, but in all other points the volume of the corresponding cells did not decrease. This is only possible when the number of

elements decreases, since without modifying the domain Ω an element can't increase its volume while all other elements remain the same. Hence it is sufficient to compare the number of elements in fine and coarse mesh and decide that coarsening was effective when this number decreased.

In contrast, there is no such thing as an optimal amount of coarsening. Too aggressive reduction of mesh entities leads to a poor distribution of vertices in the region, while removing too few entities misses the point of coarsening in the first place. Consequently, all quantities that consider the reduction of mesh entities are only a measurement for the reduction itself, without a statement about quality or optimality. A straightforward approach would be to determine a ratio between mesh characteristics like average cell volumes or numbers of cells or vertices. But such quantities take their maximum in the extreme case of reducing the mesh to a single element, which is usually not a desired behavior. In fact, to avoid such cases the independent set, described in section 3.2.2, was introduced. On the other hand, it is also not clear what can be considered a useful amount of removed vertices. Due to stability problems of simulations, as explained later in chapter 8, when applying too aggressive coarsening, a smaller number might be better than too big an amount, still allowing for multiple successive applications of the coarsening scheme, if needed.

Therefore, due to the lack of a better quantity, the efficiency of the coarsening M' of a mesh M is given as the ratio between the numbers of deleted and marked cells:

$$\text{Efficiency}(M \to M') := \frac{|\mathcal{T}| - |\mathcal{T}'|}{|\mathcal{C}|}, \tag{6.1}$$

with $\mathcal{T}$ and $\mathcal{C}$ being the sets of elements in the mesh and marked elements, respectively. One should note that it is not clear what value for efficiency can be considered useful, since it turned out to to be usually around 10%, but for some meshes dropped down to 0.1%. But it gives at least a value, that relates to the number of cells marked for coarsening and the amount of actually removed cells. This allows to compare results from different runs of the same testcase by means of a single quantity but not to compare results from different setups.

Furthermore, it should be noted that due to the previously established fact, that a decrease in the number of cells indicates effectivity of coarsening, an efficiency larger than zero does also indicate effective coarsening. Hence, in the following only the efficiency of coarsening test cases will be mentioned since effectivity is given by that implicitly.

6.2 Efficiency of parallel coarsening

With respect to the fact that there is no such thing as a good value for coarsening efficiency, this section investigates instead whether coarsening efficiency stays the same for an increasing number of processes. This is also an important quality of the algorithm, since the overall outcome of the mesh adaption should approximately be the same, independent of the number of processes used to achieve this result.

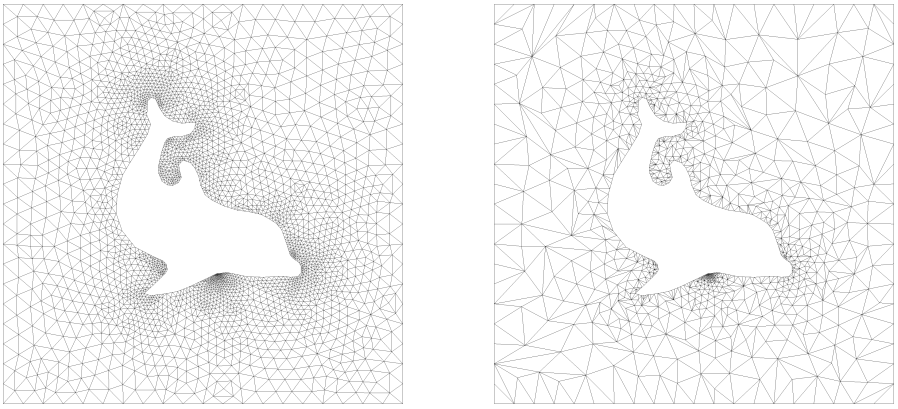


Figure 6.1. Fine and uniformly coarsened version of the triangular dolphin mesh.

This was investigated for both two- and three-dimensional meshes by uniformly coarsening the mesh. Any testcases with coarsening of a smaller amount than the whole domain would only add additional data but not change the outcome of the algorithm since in this context coarsening of the entire domain can also be seen as coarsening of a subdomain of an even bigger domain. For the 3D case, the already known landing gear mesh and the later used airplane mesh (see appendix A) were used and scaled from 1 up to 128 processes, while the 2D case was run with an unstructured triangular mesh of DOLFINs heraldic animal, which is shown in fine and coarsened form in figure 6.1. Due to the small size of the mesh it was only run on up to 8 processes.

Looking at the trend of the coarsening efficiency over the varying number of processes (see figure 6.2) reveals that it indeed varies within some range for all testcases, but does not rise or drop in any significant way with increasing number of processes. The changed order of coarsening attempts is responsible for those variations. In fact, any random change to the local numbering of mesh entities would cause similar variations, even if exactly the same testcase on the same number of processes was run.

But the numbers reveal also the problem of deciding, what reduction of number of cells can be considered a good coarsening efficiency. While in the two-dimensional case uniform coarsening shows an efficiency close to 70%, the landing gear ends up with a little more than 10% and the airplane even with less than 0.4%. There are various reasons for this: first, the difference between two and three dimensions lies in the increased complexity of the coarsening operation. Removing a vertex in a tetrahedral mesh modifies a by far larger number of elements than in two dimensions, making it more likely to run into problems due to the mesh quality criterion or cell orientation. But even more impact has the shape of the mesh itself, more precisely the different scales of elements included in the mesh: Between the areas of largest and smallest cells in the dolphin mesh lies a factor of 1129, while the factor between

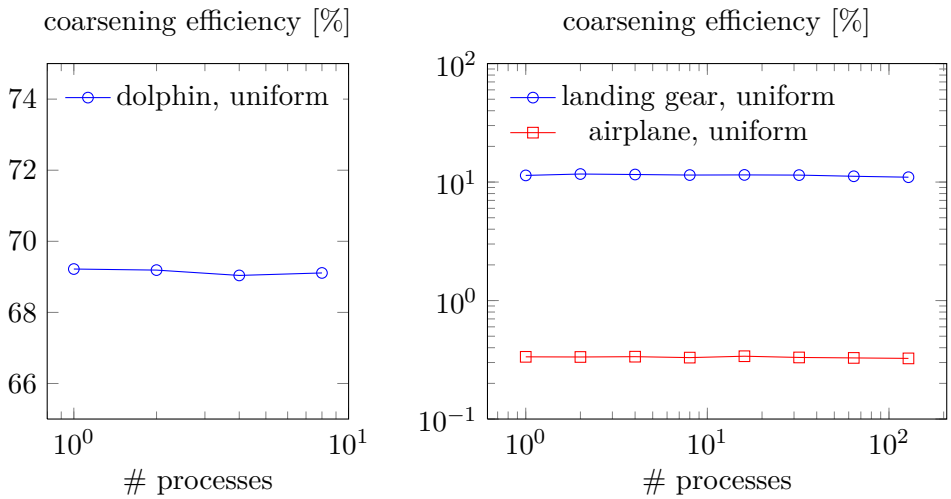


Figure 6.2. Coarsening efficiency of uniform coarsening of the dolphin, landing gear and airplane meshes for different numbers of processes. One should note the different scaling of the ordinate axis between two- and threedimensional cases.

largest and smallest cell volumes in the landing gear mesh is $8.49 \cdot 10^5$ and in the airplane mesh even $5.85 \cdot 10^{11}$. This is due to the fact that object boundaries need a very high resolution to be resolved correctly but the outer parts of the domain can be covered with fewer elements with larger volumes. Although the element sizes change gradually and extremely large elements are not next to very small ones, it still limits the number of successful coarsening attempts.

Concluding, one should note that the efficiency of coarsening is extremely dependent on the mesh but does not depend on the number of involved processes.

Chapter 7

Performance analysis

When performing computations in parallel, it is always of interest how the execution time varies with different numbers of processes and problem sizes. A good scalability is a key criterion for the successful simulation of larger problems.

7.1 Performance model

Prior to discussing the results from scalability experiments, a theoretical complexity estimate is necessary to gain some insight what results one can expect and to be able to discuss the actual outcomes in a reasonable context. For that, all of the algorithms described in chapter 3 will be analyzed with respect to their complexity and a total complexity estimate will be constructed from that.

7.1.1 Finding the independent set

The first step of the coarsening procedure is to find the independent set of vertices, as shown in algorithm 3.2. This is done only once and dominated by the loop over all vertices in the mesh. For each vertex, only the neighboring vertices are checked, which are much less than the total number of vertices in the mesh. Hence this initial computation has complexity $\mathcal{O}(|\mathcal{V}|/p)$, with $\mathcal{V}$ being the set of all vertices and the modulus indicating the number of cells in the set and p being the number of processes.

7.1.2 Edge and vertex selection

The first step towards the coarsening of a cell $K \in \mathcal{C}$ is to choose an edge for the collapse (see algorithm 3.3). In terms of complexity, this step is characterized by the loop over all edges of K. Operations like the computation of the edge length take constant time after incorporating the optimizations described in section 5.2.1. Since the loop has the same number of iterations for each cell, the overall complexity of the edge selection is also constant in $\mathcal{O}(1)$.

Once the edge is known, a decision has to be made which vertex should be deleted and which remains in the mesh (algorithm 3.4). Since this only consists of checking for affiliation to the independent set, it has also constant complexity.

7.1.3 Edge collapse

Once it is known which vertex v_D is pulled onto which v_R, the edge collapse itself is relatively cheap. In the implementation it consists of three steps, hence slightly different than in the more abstract formulation of algorithm 3.1: (i) removing all cells adjacent to the collapsed edge, i. e., removing the respective entries from the cell lists that each vertex carries as well as removing them from the global cell list, and (ii) replacing v_D with v_R in the other cells adjacent to the deleted vertex. This also includes updating the cell list of the remaining vertex. Finally, (iii) deleting the vertex v_D.

Let $\mathcal{T}_D, \mathcal{T}_R$ be the sets of elements that are deleted or modified, respectively. Removing the cells in $\mathcal{T}_D$ from the global list takes constant time due to the used datastructure and removing them from the cell lists in each vertex is in $\mathcal{O}(n_v)$ for each cell, with n_v being the number of vertices that build up a cell. For triangular meshes, $n_v = 3$ and for tetrahedral meshes $n_v = 4$. Since the number of vertices is the same for all cells, it is assumed to be constant and the resulting complexity for the first step is $\mathcal{O}(|\mathcal{T}_D|)$.

Replacing v_D with v_R in all cells in $\mathcal{T}_R$ has complexity $\mathcal{O}(|\mathcal{T}_R|)$, since replacing the vertex in each of these cells and inserting it into the local list of v_R takes constant time for each cell. Trivially, deleting the vertex from the global vertex list is also in $\mathcal{O}(1)$. Hence, the resulting complexity for the edge collapse step is $\mathcal{O}(|\mathcal{T}_D| + |\mathcal{T}_R|)$.

7.1.4 Checking the mesh

The last step executed in each coarsening attempt is the check of mesh quality and validity (algorithm 3.5). It consists of a loop over the modified elements, hence has complexity $\mathcal{T}_D$. Assuming a successful attempt, the inner loop of the parallel coarsening algorithm 3.8 is finished at this point. However, if the check fails all steps from the edge collapse in the previous section have to be undone, with the same complexity $\mathcal{O}(|\mathcal{T}_D| + |\mathcal{T}_R|)$.

7.1.5 The inner and outer loop

Taking the results from the steps together, an estimate of the total complexity of one iteration of the inner loop can be given:

$$\underbrace{\mathcal{O}(1) + \mathcal{O}(1)}_{\text{edge/vertex selection}} + \underbrace{\mathcal{O}(|\mathcal{T}_D| + |\mathcal{T}_R|)}_{\text{edge collapse}} + \underbrace{\mathcal{O}(|\mathcal{T}_R|) + \Big(\mathcal{O}(|\mathcal{T}_D| + |\mathcal{T}_R|)\Big)}_{\text{mesh check}} = \mathcal{O}(|\mathcal{T}_D| + |\mathcal{T}_R|). \tag{7.1}$$

Although the steps to undo the previous edge collapse don't have to be executed every time (hence in brackets), they don't introduce any additional asymptotic complexity and are therefore included here to give an estimate of the worst case.

The number of iterations of the inner loop is given by the number of cells marked for coarsening, i. e., the number of elements in $\mathcal{C}$. This loop is also where parallel execution comes into play, since the marked cells are distributed among the processes. Here, an even distribution of all marked cells is assumed. Consequently, the total complexity of the inner loop is

$$\mathcal{O}\left((|\mathcal{T}_D| + |\mathcal{T}_R|) \cdot \frac{|\mathcal{C}|}{p}\right). \tag{7.2}$$

Since the number of elements in $\mathcal{T}_D$ and $\mathcal{T}_R$ depends on the number of cells adjacent to the collapsed edge and vertex, it is assumed in the following that these sets refer to the worst case.

The inner loop is encapsulated by another outer loop, but following the limitation of the coarsening attempts as described in section 5.2.3, it is ensured that this loop is not executed more than ten times. Hence, it enters only as a constant and doesn't show up in the complexity estimate.

7.1.6 Migration

The migration phase is the most complicated step to determine the complexity. The involved steps were outlined in algorithm 3.9. First, a global reduction operation is performed to determine the global number of vertices (and detect termination), but this step is going to be omitted here. Next, it includes sending requests for all vertices in $\mathcal{M}_v$ to their respective owners, i. e., in the worst case $p-1$ messages of different size $|\mathcal{M}_{v,i}|$. The owners then send requests for all vertices in $\mathcal{R}_v$ to all sharing processes, i. e., in the worst case $p-1$ messages of varying size $|\mathcal{R}_{v,i}|$. Those messages contain the global index of each requested vertex, paired with the rank of the requesting process. Hence, the total communication time for exchanging those requests is

$$T_{\text{req,comm}} = (p-1)\Big(2\tau_s + \max(|\mathcal{M}_{v,i}|)\tau_b + \max(|\mathcal{R}_{v,i}|)\tau_b\Big), \tag{7.3}$$

with latency τ_s and bandwidth τ_b.

From the received requests, each process computes the changes to its local partition before the existing distribution routine is used to apply those. Jansson [19] analyzed the complexity of this routine and found its computational cost to be in $\mathcal{O}(|\mathcal{T}|/p)$. The communication cost depends on the number of vertices per element, n_v, the geometrical dimension d and the percentage of ghost vertices, α, since those have to be exchanged in an additional communication step. This yields the communication time for distribution

$$T_{\text{dist,comm}} = (p-1)\left(4\tau_s + \left(\frac{|\mathcal{T}|}{p}(n_v + n_v d) + \frac{|\mathcal{V}|}{p}(\alpha + \alpha d)\right)\tau_b\right). \tag{7.4}$$

Regarding the computational complexity, the migration phase involves some steps to prepare and process above communication, with the most expensive ones being the export from the dynamic mesh to a regular mesh object and the import vice-versa, making it possible to use DOLFIN's mesh distribution routines. Since this mainly involves inserting every cell and vertex in the other object, their complexity is $\mathcal{O}((|\mathcal{V}|+|\mathcal{T}|)/p)$. Additionally, the connectivity between cells and vertices has to be computed. This connectivity information is necessary to obtain a mapping from vertices to adjacent cells and vice versa. In contrast to the other topological information, e. g., edge-vertex connectivity, which were explicitly avoided, this relation could not be left out. This operation has also a complexity in $\mathcal{O}(|\mathcal{T}|/p)$ with a rather large constant. Hence, the overall computational complexity of the migration phase, including distribution, is

$$\mathcal{O}\left(\frac{|\mathcal{V}|+|\mathcal{T}|}{p}\right). \tag{7.5}$$

Although migration is performed in every iteration of the outer loop, the number of iterations of the outer loop is limited to ten and hence migration is not executed more often than that.

7.1.7 Total complexity

Gathering all steps from before allows to give an estimate of the overall computational complexity of the coarsening scheme:

$$\underbrace{\mathcal{O}\left(\frac{|\mathcal{V}|}{p}\right)}_{\text{indep. set}} + \underbrace{\mathcal{O}\left((|\mathcal{T}_D|+|\mathcal{T}_R|)\cdot\frac{|\mathcal{C}|}{p}\right)}_{\text{inner loop}} + \underbrace{\mathcal{O}\left(\frac{|\mathcal{V}|+|\mathcal{T}|}{p}\right)}_{\text{migration}} \tag{7.6}$$

In almost every case, the number of adjacent cells (contained in the complexity as $|\mathcal{T}_D|$ and $|\mathcal{T}_R|$) to a vertex is some magnitudes smaller than the total number of cells in the mesh. They can only be in the same magnitude for very small meshes, which are not relevant cases here. The number of marked cells $\mathcal{C}$, however, is relatively close to the total number of cells. In typical use cases, 5-20% of all cells are marked, for the more extreme cases of uniform coarsening even all of them, making $|\mathcal{C}| = |\mathcal{T}|$. Hence a reasonable but simpler estimate of above complexity is

$$\mathcal{O}\left(\frac{|\mathcal{V}|+|\mathcal{T}|+|\mathcal{C}|}{p}\right). \tag{7.7}$$

Combining this with the communication times gives the following estimate of execution time:

$$\begin{aligned} T_{\text{coarsen}} = {} & \mathcal{O}\left(\frac{|\mathcal{V}|+|\mathcal{T}|+|\mathcal{C}|}{p}\right)\cdot\tau_f \\ & + (p-1)\Big(2\tau_s + \max(|\mathcal{M}_{v,i}|)\tau_b + \max(|\mathcal{R}_{v,i}|)\tau_b\Big) \\ & + (p-1)\left(4\tau_s + \left(\frac{|\mathcal{T}|}{p}(n_v+n_v d) + \frac{|\mathcal{V}|}{p}(\alpha+\alpha d)\right)\tau_b\right), \end{aligned} \tag{7.8}$$

where τ_f is the time to perform one floating point operation and, as before, latency τ_s and bandwidth τ_b.

This result makes clear, that the execution time of the coarsening scheme does not solely depend on the number of cells marked for coarsening, but is particularly dependent on the total size of the mesh. Furthermore, for large numbers of requested vertices, communication can drastically increase the total execution time. The time spent on communication will also rise with a larger number of processes, since more processes increase the number of vertices located on process boundaries, making it more likely for a vertex to end up in $\mathcal{M}_v$, and the number of messages increases. This is a common property to many algorithms that scale well for a small number of processes but suffer from too many.

Also important to note is that the migration step completely vanishes for serial execution. Especially the mesh import and export routines are not executed after each iteration of the outer loop, hence giving an estimated execution time of

$$T_{\text{coarsen,serial}} = \mathcal{O}(|\mathcal{V}| + |\mathcal{C}|) \cdot \tau_f. \tag{7.9}$$

7.2 Test environment

All performance tests have been performed at PDCs[1] *Ferlin*, a former resource from the *Swedish National Infrastructure for Computing* (SNIC). It has 512 compute nodes, each equipped with two AMD Opteron 2.2 GHz CPUs (2374 HE, 4 cores, 4 threads) and 16 GByte memory. The machine aims primarily on small, long-running jobs, hence the interconnect, though it is Infiniband, is not particularly optimized. Also, jobs usually cannot exceed a number of 16 compute nodes, i. e., 128 processes, but for the scalability studies the use of up to 32 compute nodes, i. e., 256 processes, was possible. Open MPI 1.4.1 was used as MPI-library and compilation done using icpc 11.1.

7.3 Strong scaling

First, the strong scaling behavior of the coarsening scheme was investigated. Strong scaling measures the reduction of the total execution time, when executing the same problem on an increasing number of processes. Ideally, twice the number of processes should halve the execution time. Two quantities, speedup and parallel efficiency, are used to measure the scaling behavior. Speedup S_p and efficiency E_p, when using p processes, are defined in the usual way:

$$S_p = \frac{T_1}{T_p} \quad , \qquad E_p = \frac{S_p}{p}, \tag{7.10}$$

with T_p being the execution time with p processes.

[1] `http://www.pdc.kth.se/`

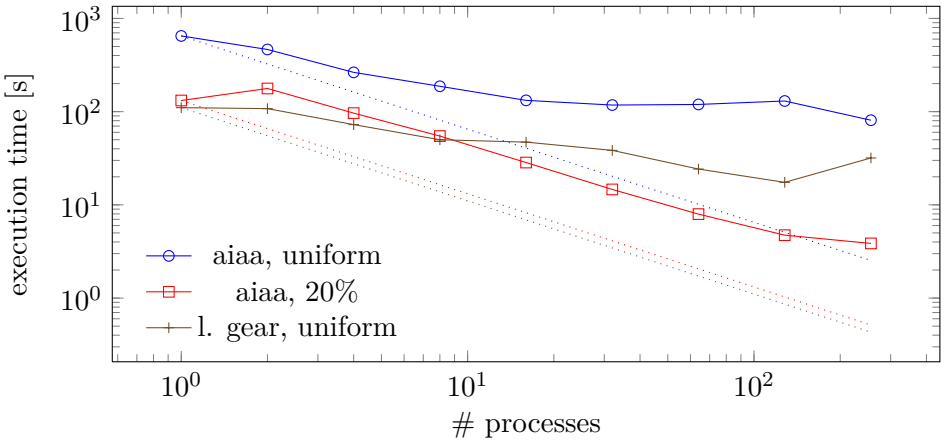


Figure 7.1. Strong scaling results for parallel coarsening of the airplane mesh. The upper curve shows uniform coarsening, the lower curve coarsening for 20% of the cells marked on each processor. The dotted lines indicate ideal scaling.

The scalability experiments were done using the mesh of the AIAA airplane model (see appendix A), which has almost 700,000 vertices and more than 3.3 million cells, in two test cases: first, the usual uniform coarsening, in which all cells are marked for coarsening, and secondly, 20% of the cells were marked on each process. As an additional third test, uniform coarsening of the landing gear mesh (168,007 vertices and 861,776 cells) was used.

The resulting runtime behavior is visible in figure 7.1. Dotted lines indicate

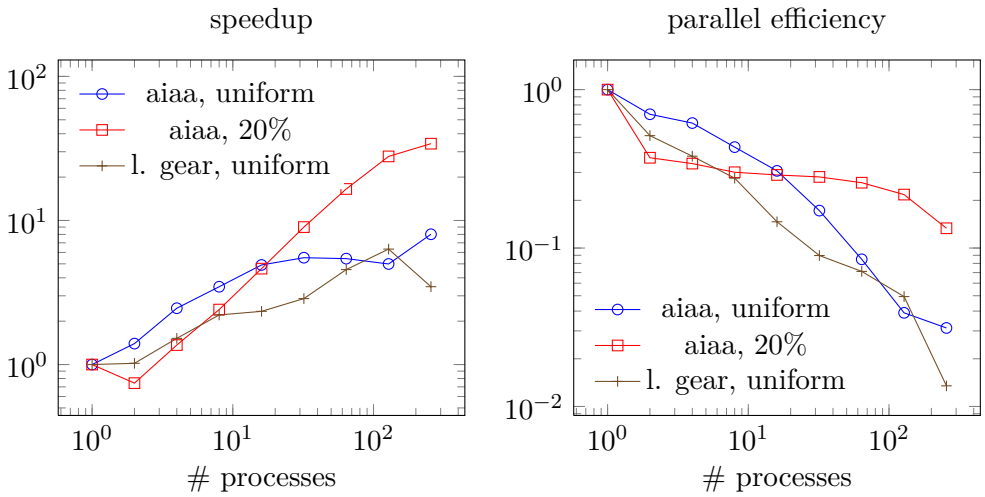


Figure 7.2. Speedup for parallel coarsening of the airplane and landing gear meshes and parallelization efficiency.

optimal scaling behavior.

With 20% marked cells in the airplane mesh, the scaling would be close to ideal when aligning the ideal curve with the result from two processes, which means, taking the additional workload from migration into the base case, since the computational effort is a lot decreased when executing in serial. But from one to two processes, execution time increased due to the large impact of the migration phase, which increased computation time and additionally added interprocess communication. Consequently, for small $|\mathcal{C}|$ the communication impact is large compared to the gain from parallelization when going from one to two processes.

For the case of uniform coarsening of the airplane, the curve is flatter and levels out to the end but does not show as much suffering from communication cost when going from one to two processes. This means, although the amount of migrated cells was higher due to coarsening at all process boundaries, communication did not take as large a portion of the total runtime. This is due to the increased local coarsening workload. But the larger amount of communication was also responsible for the fact that the slope is less steep, since with decreasing local problem sizes, communication took a larger stake in the overall execution time. This is particularly well visible in the runtimes from uniform coarsening of the landing gear mesh. Due to the smaller amount of mesh entities on each process, the local workload was smaller but the migration cost did not change as much. This led to the even flatter curve, confirming the expectations from the theoretical analysis.

The speedup confirms these observations, as shown in figure 7.2. The overall gain from parallelization is much better for the case of 20% marked cells, though it scales less well for a small number of processes. This is also clearly visible from the efficiency plot.

One should note that up to 8 processes all computation was performed on a single node and hence communication did not include any connection over the Infiniband network. However, increasing the number from 8 to 16 processes and even further did not show any obvious performance drop for the airplane mesh that could be related to the slower interconnect. In contrast, for the landing gear there is a damp in the strong scaling curve visible, emphasizing the stronger dependency on communication in this case, as it was found earlier. Therefore, the expense of the migration phase lies in the need for synchronization itself and the involved computational effort, not in a too large amount of transfered data.

7.4 Normalized scaling

With the theoretical analysis of the performance behavior in mind, it is easy to understand why it is not feasible to do conventional weak scaling studies: the dependency on both, the number of marked cells as well as the mesh size, would require to use different meshes with a constant number of cells and vertices per process. Those meshes would then have to have also similar quality properties etc. to give comparable results.

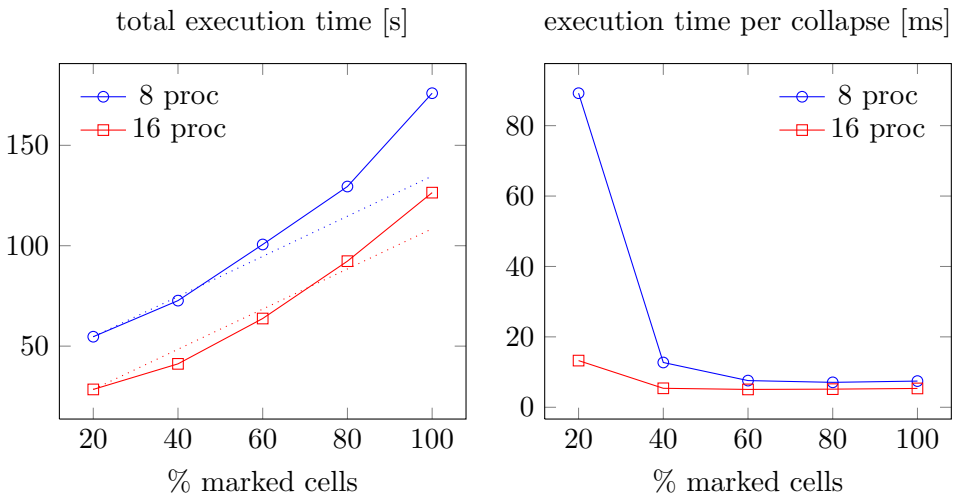


Figure 7.3. Execution time for a varying percentage of marked cells in the airplane mesh. Total execution time is shown on the left while the right shows the average execution time per successful edge collapse. Dotted lines indicate linear scaling.

Instead, the results from the experiments are evaluated using different normalizations to investigate certain characteristics of the algorithm.

7.4.1 Dependency on problem size

Marking a larger number of cells for coarsening in the same mesh increases the number of iterations of the inner loop in the coarsening algorithm 3.8. But the execution time per collapse should approximately stay the same.

The total execution time was measured for an increasing percentage of marked cells in the airplane mesh for two different numbers of processes (see figure 7.3). As expected, with more marked cells the total execution time rose. Since a larger percentage of marked cells means that more of the marked cells are located on the boundary and consequently more communication is necessary, this rise is not linear. However, normalizing these execution times by the number of successfully performed edge collapses shows that the time spent on each edge collapse is approximately the same. Only for small portions of marked cells it is higher, since in those cases the dependency on the mesh size plays a larger role in the computational complexity.

7.4.2 Dependency on mesh size per processor

One finding of the theoretical performance analysis was that the total execution time depends on the local mesh size, i. e., number of mesh entities per processor. When normalizing the times from the strong scaling experiments by the number of vertices per process, one eliminates this dependency and gets insight which relation the number of marked cells and communication have to each other in the total

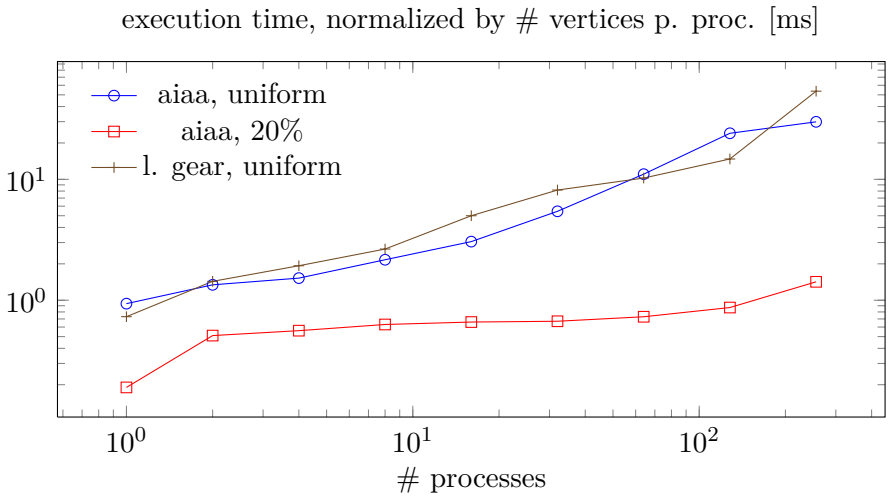


Figure 7.4. Execution time of uniform/20% coarsening of the airplane and landing gear mesh, normalized by the number of vertices per processor.

execution time. Theoretically, the computational effort should drop linearly with increasing number of processes while communication time rises.

Figure 7.4 shows that for a large number of marked processes the execution time rises, even when normalized by the number of vertices per process. This means, the communication effort is much larger than the performance gain from the reduced workload per process. On the other hand, when having less cells marked for coarsening and therefore a smaller number of entities in the migration, the normalized execution time does only slightly rise. This is in agreement with the previous section, which found that for a smaller amount of marked cells the execution time is dominated by communication and mesh size, not by the execution of edge collapses. This emphasizes once more that the highest performance gain, when increasing the number of processes, is obtained from the smaller mesh size on each process. All other components of the algorithm take, at best, the same amount of time.

Chapter 8

Application of mesh coarsening in an adaptive flow simulation

To illustrate a possible use case of the mesh coarsening algorithm, it was applied in two flow simulation scenarios: unsteady incompressible flow past a wing section and past a cylinder. Before presenting the outcomes of the simulations, a general introduction into the problem setups is given. A discussion of the obtained results concludes this chapter.

8.1 General problem description

Both simulation setups used the General Galerkin method introduced in section 2.2 as it is implemented in the incompressible Navier-Stokes solver of Unicorn. An elaborate description of the flow past a wing section was also given by Hoffman et al. [26].

Unicorn's solver implements the adaptive algorithm 2.1 as it was introduced in section 2.3: It solves primal and dual problem in each iteration of the algorithm and adapts the mesh according to the described error indicators before solving primal and dual problem once again. Projection of the solution was not used, since the increasing mesh size required to add more processes after few iterations, making it impossible to continue with the saved solutions.

The coarsening procedures were added after the existing refinement step of the adaptive algorithm. Similar to the refinement, where a fixed percentage of cells with largest errors is marked for refinement, cells with lowest errors were marked for coarsening. Those cell markers had to be projected into the refined mesh before applying coarsening. Problems with this choice of coarsening markers are discussed later.

8.1.1 Boundary conditions

The turbulent boundary layers are modeled using a slip with friction boundary condition [26]:

$$\begin{aligned} u \cdot n &= 0, \\ u \cdot \tau_k + \beta^{-1} n^T \sigma \tau_k &= 0, \quad k = 1, 2, \end{aligned} \tag{8.1}$$

with n being an outward unit normal vector, and τ_k orthonormal tangent vectors at the boundary. β is a constant friction parameter. The normal component corresponds to a slip boundary condition, tangential components model a friction boundary condition.

In normal direction the boundary condition is implemented in strong form, i. e., by modifying the algebraic system after assembly, whereas the tangential boundary conditions are applied in weak form as boundary integrals in the variational formulation.

8.1.2 Goal of this applications

The main objective of these simulations was to demonstrate the fundamental usability of the coarsening procedure paired with a possible application of it. For this reason, the outcomes of those simulations are not discussed with respect to their physical meaning but only regarding the role of coarsening.

Each setup was run twice, once only with mesh refinement and once with both mesh refinement and coarsening applied. It is expected that each of them gives the same results but with different mesh sizes, hence giving two things that can be compared: the evolution of the mesh size over the different adaptive iterations, which is expected to be different, and the computed aerodynamic forces that should be the same.

The aerodynamic force evaluated in both setups is the drag force F_D, which is compared by means of the associated *drag coefficient* [6]

$$c_D = \frac{1}{\frac{1}{2}\rho U_\infty^2 A} F_D, \tag{8.2}$$

with fluid density ρ, free stream velocity U_∞ and representative area A of the object. For the wing section, additionally the lift force F_L is evaluated, which is defined as the force orthogonal to drag. It is again compared by means of the associated *lift coefficient*

$$c_L = \frac{1}{\frac{1}{2}\rho U_\infty^2 A} F_L. \tag{8.3}$$

8.2 Flow past a cylinder

The flow past a cylinder used the structured cylinder mesh (see appendix A) and a medium friction value $\beta = 0.001$ between free and no slip boundary conditions was chosen. In each adaptive iteration, the primal solution was computed up to an

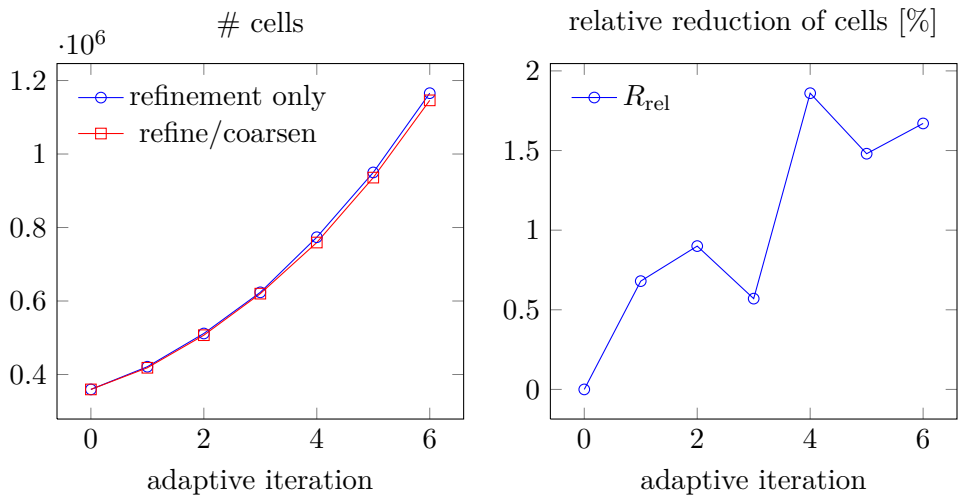


Figure 8.1. Evolution of the mesh size over adaptive iterations of the cylinder: given as total number of cells and relative reduction of the cell number due to coarsening.

end time 5.0 and the dual solution to an end time 2.5, before 10% of the cells were marked for refinement and 5% were marked for coarsening. As solver for the linear systems a stabilized version of the BiConjugate Gradient Squared method was used, together with a blocked Jacobi method as preconditioner.

Stability problems arose in adaptive iteration 6 of the simulation with refinement and coarsening, causing the solver no longer to converge. Consequently, the mesh sizes can be compared up to iteration 6 but aerodynamic forces only up to iteration 5, as this is the last complete iteration.

8.2.1 Evolution of mesh sizes

With 10% of the cells marked for refinement after each adaptive iteration, the number of cells increased continuously in both cases with and without coarsening. However, the latter also removed cells based on the 5% marked cells. But one should note that marking of 5% of the cells does not necessarily mean a reduction of the number of elements in the same range. Chapter 6 showed that the efficiency of the coarsening is rather in the range of a one- to low two-digit percentage for tetrahedral meshes. Hence, it is interesting to see how much impact the coarsening on the total mesh size had.

Figure 8.1 shows the evolution of the number of cells over the adaptive iterations, revealing a small but measurable reduction that increases with each adaptive iteration. Additionally, the relative reduction R_{rel} of the number of cells is plotted to get a better impression of the magnitude of the reduction. The relative reduction R_{rel} is defined as

$$R_{\text{rel},i} = \frac{N_{r,i} - N_{rc,i}}{N_{r,i}}, \quad i = 0, 1, 2, \ldots, \tag{8.4}$$

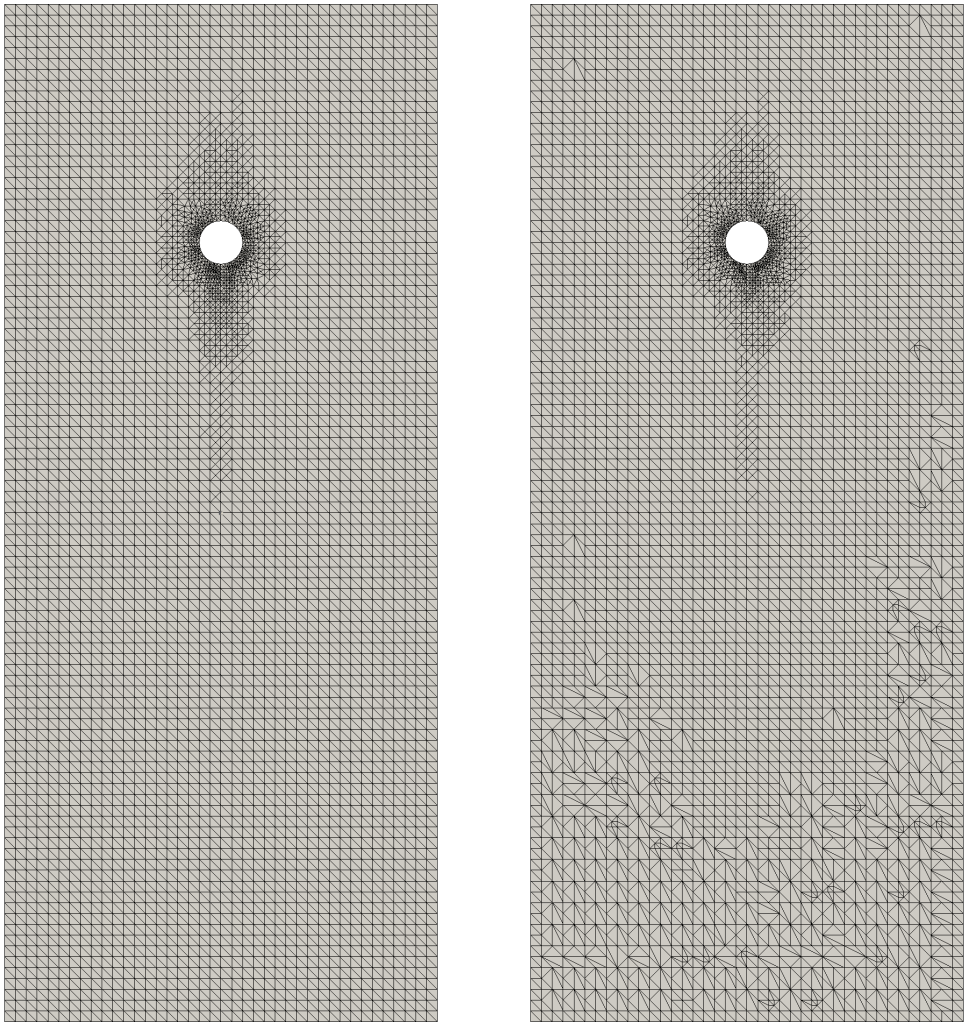


Figure 8.2. Comparison of cylinder meshes in iteration 5: An x-y-plane in the center of the mesh shows the locally refined mesh regions around the cylinder and, for the case of applied coarsening, the coarser regions in the downstream end of the domain.

with $N_{r,i}$ being the total number of cells in the i-th adaptive iteration when using only refinement and $N_{rc,i}$ when applying both refinement and coarsening. On average, the relative reduction also increases with each adaptive iteration. While in the first adaptive iterations mainly cells close to the boundary were marked for coarsening (see figure 8.6), in later iterations also cells further inside are selected and hence the coarsening more efficient.

The influence from coarsening is well visible when comparing the meshes from both runs in one of the advanced adaptive iterations. Figure 8.2 shows a central cut

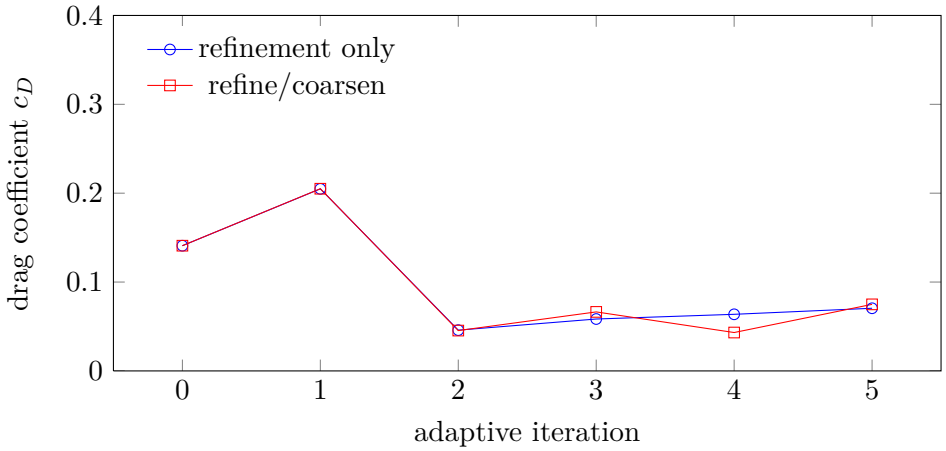


Figure 8.3. Drag coefficient for the cylinder in adaptive iterations with and without coarsening.

through the domain, clearly revealing the difference when using mesh coarsening. While local refinement happens in both cases around the cylinder, coarsening also reduces the number of cells in the downstream corners of the domain.

8.2.2 Comparing drag coefficients

The drag force was averaged over the time interval $I_{\text{avg}} = [2, 5]$ to give the system some time to settle and generate an approximately stationary drag force. The resulting drag coefficients from the two simulation runs are in good agreement to each other. Hence, the reduction of mesh resolution in certain regions of the domain did not significantly influence the resulting aerodynamic forces. However, while the drag coefficients from the simulation run with refinement only appear to be converged after iteration 3, the coefficients from the simulation run with refinement and coarsening show more variation through the iterations.

8.3 Flow past a wing section

The wing section was a three dimensional version of a NACA 0012 airfoil (for the mesh see appendix A). The computations were performed with no slip boundary conditions, i. e., $\beta = 0$, and up to an end time 1.5 for the primal and 0.5 for the dual problem in each adaptive iteration. The same solvers as for the cylinder were used and 10% of the cells were marked for refinement together with 5% for coarsening, once again.

Unfortunately, in this case convergence problems arose already in iteration 4 of the adaptive algorithm. Reasons for that are discussed in the end of this chapter. Hence, the results for mesh sizes are only available up to iteration 4 and for

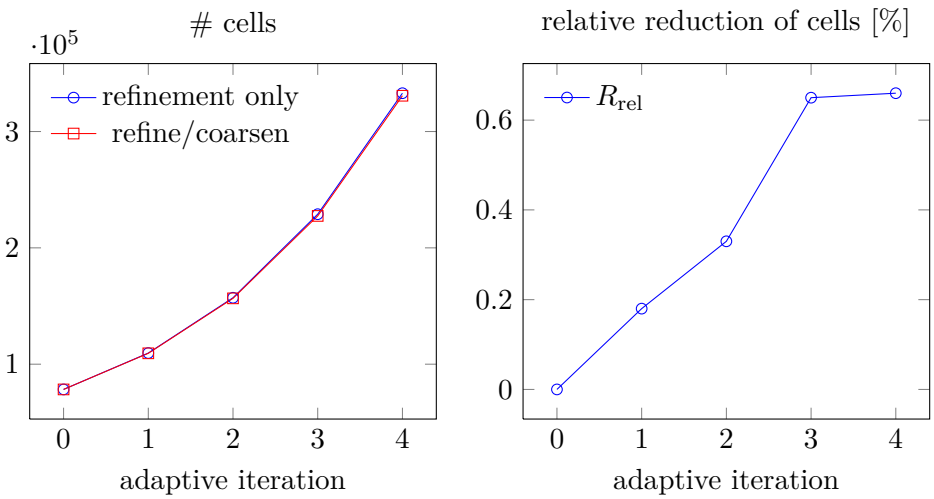


Figure 8.4. Evolution of the mesh size over adaptive iterations of the wing: given as total number of cells and relative reduction of the cell number due to coarsening.

coefficients up to iteration 3, as it is the last completed iteration.

8.3.1 Evolution of mesh sizes

The mesh size increased over the adaptive iterations similarly to the cylinder case up to iteration 4 but did not show an as significant reduction of the number of cells (see figure 8.4) as in the cylinder case.

8.3.2 Comparing drag and lift coefficients

Drag and lift-coefficients were also in good agreement between the two simulation runs (see figure 8.5). The difference is slightly larger for the drag coefficient than it is for the lift coefficient, which can be related to the fact that coarsening also appeared downstream past the wing (see figure 8.6), causing the different results. However, the differences are neglectable compared to the variation visible in successive adaptive iterations and therefore especially small compared to the deviation from the actual real values for lift and drag.

8.4 Discussion

The comparison of adaptive simulations with and without coarsening revealed that a reduction of the total number of cells is visible but comes with the cost of different, probably more erroneous, results for aerodynamic forces. Better results might be achievable with a better choice of coarsening markers.

Figure 8.6 shows the regions marked for refinement (blue) and coarsening (red) after iteration 0. It is well visible that choosing cells for coarsening on the basis

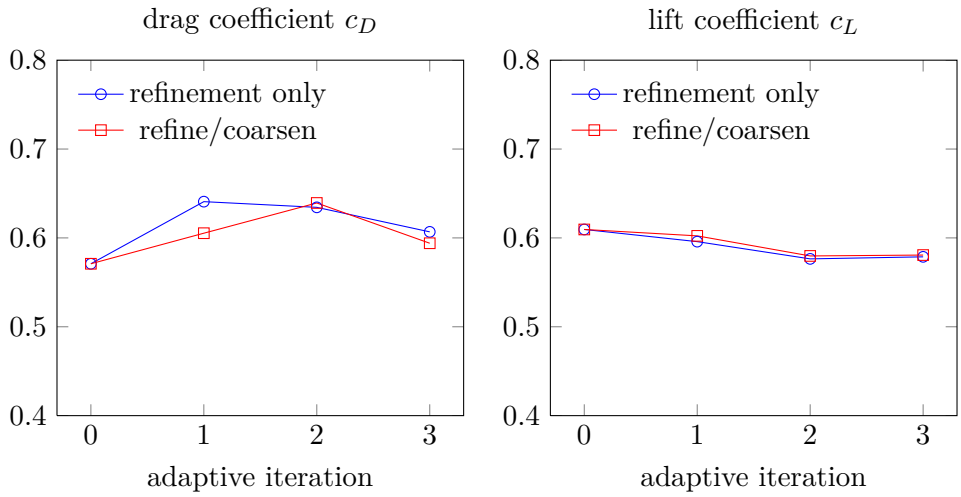


Figure 8.5. Drag coefficient (left) and lift coefficient (right) for the wing in adaptive iterations with and without coarsening.

of error indicators leads to cells being preferably marked close to the boundaries, where computed values are typically less erroneous due to the enforced boundary conditions. For the wing, it is also visible that out of neighboring cells some had been selected for refinement while others were marked for coarsening. This can be especially hazardous since coarsening also modifies neighboring cells. Hence, cells that actually require a higher resolution and are thus refined might end up getting coarsened again. Consequently, in the next adaptive iteration the error in this cell might not be reduced or, in the worst case, might even increase. This could be responsible for the unstable behavior of the wing simulation. For the cylinder, cell marker for refinement and coarsening are well separated and the simulation is more stable.

Another problem with cells marked at the boundaries is that coarsening of those cells is only possible in few cases. Since vertices on domain boundaries are included in the independent set, often only one or two vertices are available for the collapse. The boundary, however, retains its fine sampling with vertices, which leads to skinny and stretched elements. But in most cases those cells aren't coarsened at all, as it is well visible in figure 8.2. Hence, a layer of unnecessary fine cells surrounds the domain.

Also, the portion of cells marked for coarsening, in these examples 5%, is rather low. To achieve a significant performance improvement, this amount has to be increased because the percentage of actually coarsened cells is a lot less than those 5%. The execution time per element is approximately constant in Unicorn [5], hence the performance improvement is proportional to the number of coarsened cells. However, if this number is very small, this improvement might be outweighed by the additional effort from the coarsening procedure itself.

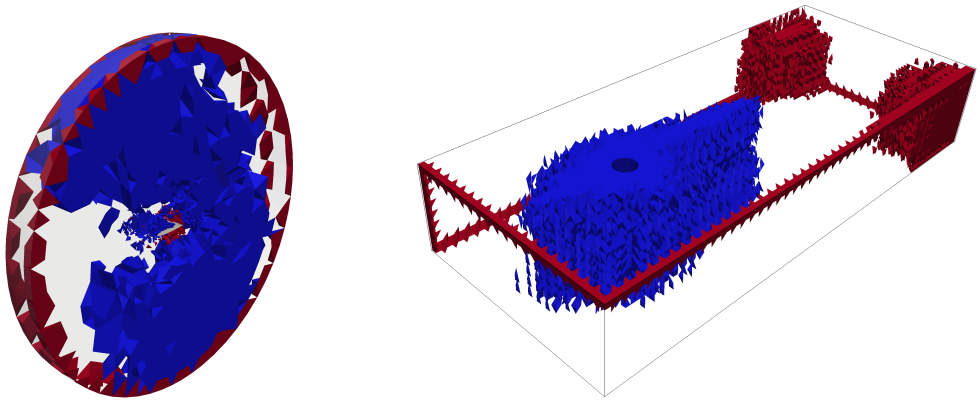


Figure 8.6. Regions marked for refinement (blue) and coarsening (red) after iteration 0 of the adaptive algorithm.

Therefore, a better way to choose these coarsening indicators has to be found. A possible approach could combine error indicators with secondary characteristics, like the position of the cell in relation to domain boundaries or the deviation of the velocity from the free stream velocity. Also, refinement markers could be taken into account, such that no neighboring cells are marked for refinement and coarsening. Additionally, it could be promising to extend the coarsening algorithm such that it includes a good scheme for boundary coarsening, e. g., incorporating some of the ideas from Garland and Heckbert [11]. Since the error is low on the outer process boundaries, a coarser boundary might be of no harm. However, the coarsening scheme has to ensure that the shape of the boundary is preserved and no nooks are created.

Nevertheless, these results show that coarsening as part of the mesh adaption step is effective and useful, provided the selection of cells for coarsening is improved.

Chapter 9

Conclusion and future work

This thesis covered methodology, implementation and application of a mesh coarsening algorithm based on local edge collapse. The focus lied on mesh coarsening in the context of finite element computations. However, most of the aspects regarding the coarsening algorithm itself are more general applicable.

An introductory part explained the finite element method. In this technique, a partial differential equation is transformed into its variational form and the domain is divided into a finite number of elements, on which ansatz functions are defined. These functions are used as basis functions and inserted into the variational form to obtain a global system of equations, which can subsequently be solved to obtain the desired solution. A variant of FEM, the General Galerkin method, uses continuous piecewise linear approximation in both space and time. Hence, it is particularly useful for time dependent systems of differential equations like the Navier-Stokes equations. An adaptive finite element solver uses a finite element method to compute the solution to a primal and dual problem, from which an a posteriori error is computed. This error is then used to determine regions of the domain that require a higher or lower resolution. Subsequently, these regions are adapted using mesh refinement or coarsening methods.

A possible coarsening technique is the edge collapse algorithm, which was presented next. The basic idea is to pull one vertex along an edge onto a neighboring vertex, causing the edge and all elements sharing this edge to vanish. The remaining elements around the pulled vertex are stretched, resulting in a locally coarser mesh. An independent set of vertices is used to maintain a proper vertex distribution when coarsening more than a few cells in connected regions. Additional care is necessary when coarsening is applied in a parallel environment, making it necessary to ensure that all possibly modified mesh entities are available on the local process before an edge collapse is performed.

In the second part of this thesis, the implementation of this edge collapse algorithm was outlined. It was done as a contribution to the FEM library DOLFIN HPC, which is part of the FEniCS project and intended for large scale computations on distributed memory machines. Despite the straightforward implementation and

parallelization of the algorithm, a number of performance optimization approaches were implemented and tested, with the most successful ones being the use of a better data structure, improvements to the migration phase and a limitation of the number of coarsening attempts.

In the last part, the effectivity and efficiency of the edge collapse algorithm were tested, revealing that the coarsening efficiency can be very diverse among different meshes. A theoretical complexity estimate showed that the execution time depends on the number of cells marked for coarsening and on the size of the mesh and the amount and size of messages exchanged between processes. Performance experiments revealed that scalability of the algorithm is highly dependent on the number of mesh entities that have to be transferred in the migration phase.

Finally, the coarsening scheme was applied in two scenarios of a fluid solver. Stability problems occurred and were linked to the selection of cells for coarsening. But the general possibility to apply coarsening in the mesh adaption phase of an adaptive simulation was shown.

Nevertheless, some issues remain with the most pressing being to find a better indicator for coarsening. Some basic ideas were mentioned in the end of chapter 8. Secondly, an extension of the coarsening method that allows to remove vertices on domain boundaries, without having significant impact on the shape of the boundary, might turn out to be advantageous for the coarsening efficiency. Since the number of mesh entities is directly proportional to the total execution time, this would give further performance improvements in the finite element solver. The addition of an efficient mesh optimization algorithm could allow to make the quality criterion less strict and, consequently, increase the coarsening efficiency.

But also improvements to the coarsening scheme itself are imaginable: since a lot of coarsening attempts fail due to the mesh quality of the remaining elements after the collapse, the edge collapse could be performed on a different edge (not necessarily the shortest) or both vertices could be collapsed onto the edge midpoint. But choosing the correct approach calls for a theoretical backup that predicts the outcome of either of these methods. Also, it is subject to further investigations if such modifications of the algorithm turn out to be successful at all.

Regarding the implementation, performance optimizations are still possible in the migration phase. Since in this step entities are only exchanged with neighboring processes, a better communication pattern might be possible. Also, replacing the existing mesh distribution routines with own migration functionalities, which would not require to convert the dynamic mesh back and forth to regular mesh objects, might give further runtime improvements.

A full integration of all mesh adaption routines is also desirable: refinement, coarsening and eventually mesh optimization. First of all, this would allow to execute all mesh modifications on one dynamic mesh, once again saving the need to convert to regular mesh objects in between. Secondly, it would also omit the need to project coarsening markers into the refined mesh and, thirdly, might allow to approach mesh adaption with a new look on its entirety.

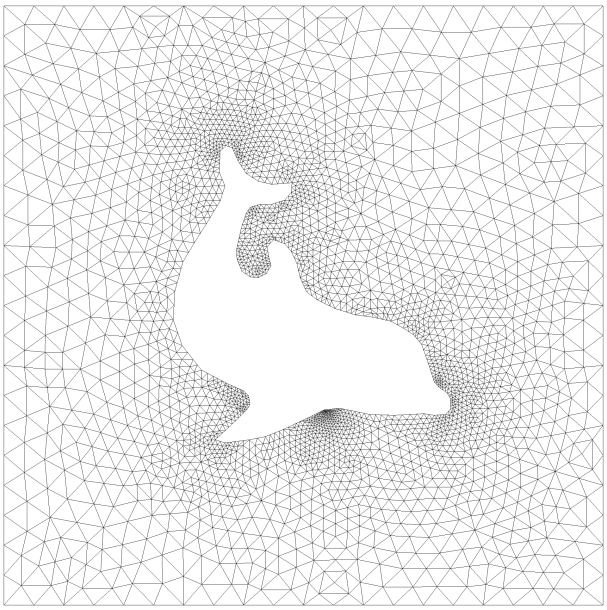
Appendices

Appendix A

Meshes

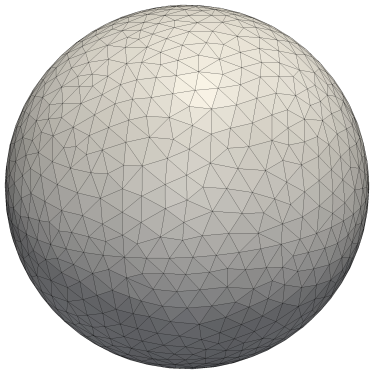
A short overview of the meshes mentioned and used in this thesis. They are categorized by their geometrical dimension and roughly ordered by increasing size.

A.1 Triangular meshes

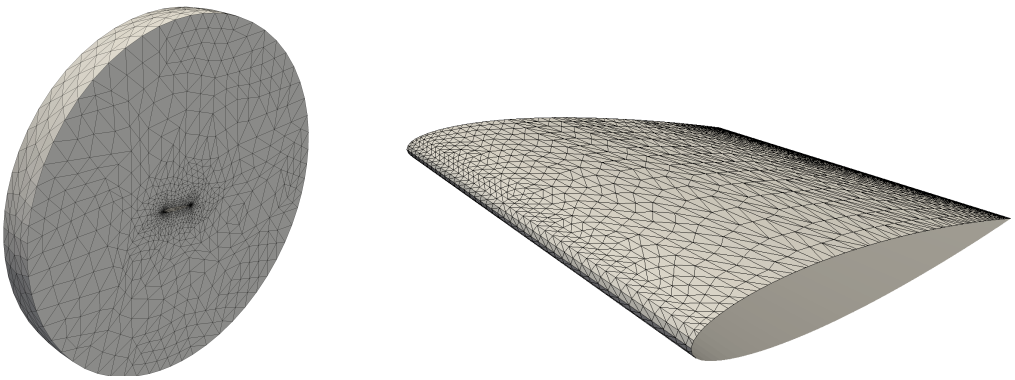


Dolphin Although many more simple 2D meshes were used for functional verification and minor testing during the development, an unstructured mesh of DOLFIN's heraldic animal was the only triangular mesh used within this thesis. It had 5400 cells and 2868 vertices and was used as a showcase for the coarsening efficiency in chapter 6.

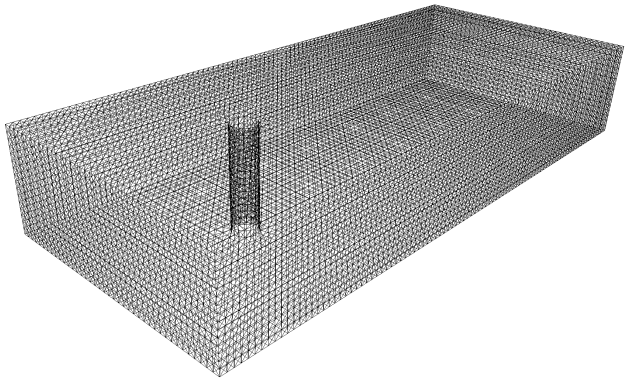
A.2 Tetrahedral meshes



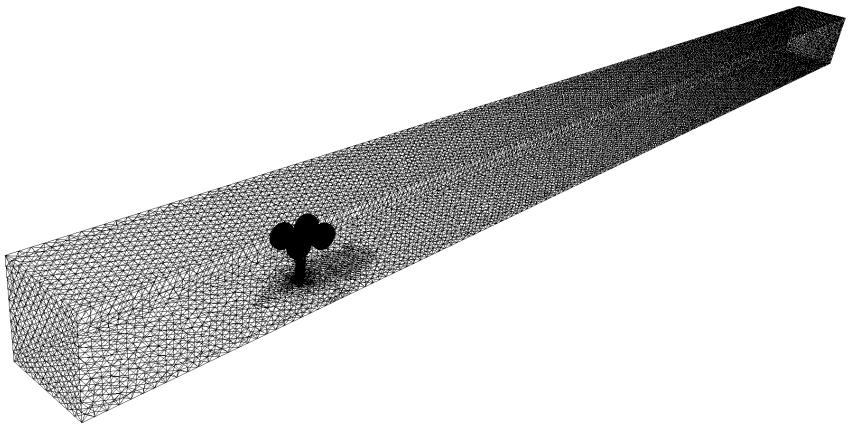
Sphere A simple sphere, triangulated with an unstructured mesh with 9312 cells and 2129 vertices. It was only used for testing of the initial algorithm and improved datastructures due to its comparably small size and hence short runtimes.



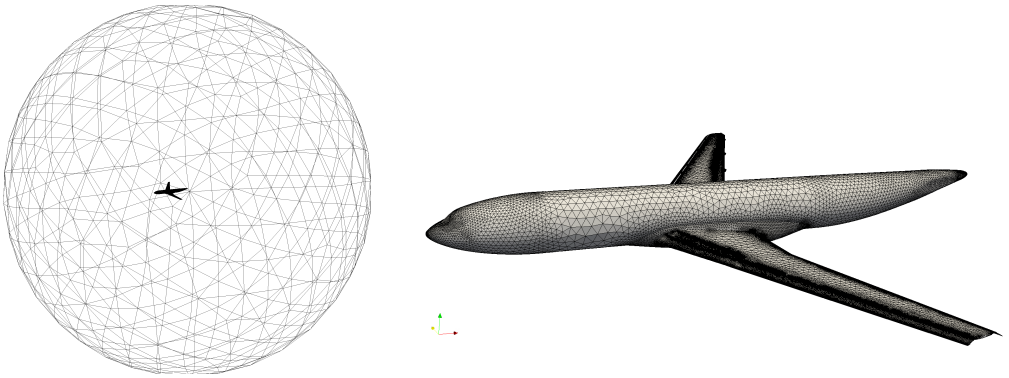
Wing An unstructured mesh of a cylindrical domain (on the left) with a wing section from a NACA0012 airfoil in the center (see right picture). It had 78,304 cells and 15,405 vertices initially and was used in the flow simulation in chapter 8.



Cylinder Originally a structured mesh of a cuboidal domain with a cylinder inside. It was uniformly refined to 359,424 cells and 66,062 vertices and used in the flow simulation (chapter 8). The adaptively refined version from iteration 6 served as input for the performance optimization tests in chapter 7.



Landing gear An unstructured mesh of a long cuboidal domain with a simplified model of a landing gear inside. It has 861,776 cells and 168,007 vertices and was used in the optimization tests in section 5.2.



Airplane An unstructured mesh of a large spherical domain with the model of a civilian aircraft in the center. With 3,344,535 cells and 694,236 vertices it was the largest mesh available and used for the performance analysis in chapter 7.

Appendix B

Bibliography

[1] Richard Phillips Feynman. *QED: The Strange Theory of Light and Matter*, page 10. Princeton University Press, 1985. ISBN 0-691-08388-6.

[2] Robert C. Kirby and Anders Logg. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 2: The finite element method. Springer, 2012. doi: 10.1007/978-3-642-23099-8. URL `http://dx.doi.org/10.1007/978-3-642-23099-8`.

[3] Niclas Jansson. *High performance adaptive finite element methods for turbulent fluid flow*. Licentiate thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, March 2011. TRITA-CSC-A 2011:02.

[4] Niclas Jansson, Johan Hoffman, and Johan Jansson. Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedal meshes. *SIAM Journal for Scientific Computing*, 34(1):C24–C41, 2012. doi: 10.1137/100800683. URL `http://dx.doi.org/10.1137/100800683`.

[5] Johan Hoffman, Johan Jansson, Rodrigo Vilela de Abreu, Niyazi Cem Degirmenci, Niclas Jansson, Kaspar Müller, Murtazo Nazarov, and Jeannette Hiromi Spühler. Unicorn: Parallel adaptive finite element simulation of turbulent flow and fluid–structure interaction for deforming domains and complex geometry. *Computers & Fluids*, 80:310–319, July 2013. ISSN 0045-7930. doi: 10.1016/j.compfluid.2012.02.003. URL `http://dx.doi.org/10.1016/j.compfluid.2012.02.003`.

[6] Johan Hoffman and Claes Johnson. *Computational Turbulent Incompressible Flow*, volume 4 of *Applied Mathematics: Body and Soul*. Springer, 2010. doi: 10.1007/978-3-540-46533-1. URL `http://dx.doi.org/10.1007/978-3-540-46533-1`.

[7] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. Simultaneous refinement and coarsening: Adaptive meshing with moving boundaries. In *7th International Meshing Roundtable*, pages 201–210, 1998.

[8] M. Darwish, J. Rached, and F. Moukalled. Unstructured adaptive mesh refinement and coarsening for fluid flow at all speeds using a coupled solver. In *AIP Conference Proceedings*, volume 1168, pages 577–680, 2009. doi: 10.1063/1.3241527. URL http://dx.doi.org/10.1063/1.3241527.

[9] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, May 2001. ISSN 0272-1716. doi: 10.1109/38.920624. URL http://dx.doi.org/10.1109/38.920624.

[10] Jiri Vad'ura. Parallel mesh decimation with GPU. 2011. URL http://www.feec.vutbr.cz/EEICT/2011/sbornik/03-Doktorske%20projekty/09-Grafika%20a%20multimedia/07-ivadura.pdf. Available online.

[11] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 209–216, 1997. ISBN 0-89791-896-7. doi: 10.1145/258734.258849. URL http://dx.doi.org/10.1145/258734.258849.

[12] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the GPU. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 161–166, 2007. ISBN 978-1-59593-628-8. doi: 10.1145/1230100.1230128. URL http://doi.acm.org/10.1145/1230100.1230128.

[13] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46:1101–1125, 1999. doi: 10.1002/(SICI)1097-0207(19991110)46:7<1101::AID-NME741>3.0.CO;2-E. URL http://dx.doi.org/10.1002/(SICI)1097-0207(19991110)46:7<1101::AID-NME741>3.0.CO;2-E.

[14] Benjamin S. Kirk, John W. Peterson, Roy H. Stogner, and Graham F. Carey. libMesh: a C/C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. doi: 10.1007/s00366-006-0049-3. URL http://dx.doi.org/10.1007/s00366-006-0049-3.

[15] Rupak Biswas and Roger C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13 (6):437–452, 1994. ISSN 0168-9274. doi: 10.1016/0168-9274(94)90007-8. URL http://dx.doi.org/10.1016/0168-9274(94)90007-8.

[16] Frédéric Alauzet, Xiangrong Li, E.Seegyoung Seol, and MarkS. Shephard. Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3):247–258, 2006. ISSN 0177-0667. doi: 10.1007/s00366-005-0009-3. URL http://dx.doi.org/10.1007/s00366-005-0009-3.

[17] Gary L Miller, Dafna Talmor, and Shang-Hua Teng. Optimal coarsening of unstructured meshes. *Journal of Algorithms*, 31(1):29–65, 1999. ISSN 0196-6774. doi: 10.1006/jagm.1998.0990. URL http://dx.doi.org/10.1006/jagm.1998.0990.

[18] E. L. Lawler. A note on the complexity of the chromatic number problem. *Information Processing Letters*, 5(3):66–67, 1976. ISSN 0020-0190. doi: 10.1016/0020-0190(76)90065-X. URL http://dx.doi.org/10.1016/0020-0190(76)90065-X.

[19] Niclas Jansson. Adaptive mesh refinement for large scale parallel computing with DOLFIN. M. Sc. thesis, KTH Royal Institute of Technology, Stockholm, 2008. TRITA-CSC-E 2008:051.

[20] A. Logg, K.-A. Mardal, and G. N. Wells, editors. *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012. doi: 10.1007/978-3-642-23099-8. URL http://dx.doi.org/10.1007/978-3-642-23099-8.

[21] Anders Logg and Garth N. Wells. Dolfin: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2):20:1–20:28, April 2010. ISSN 0098-3500. doi: 10.1145/1731022.1731030. URL http://doi.acm.org/10.1145/1731022.1731030.

[22] Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3):417–444, September 2006. ISSN 0098-3500. doi: 10.1145/1163641.1163644. URL http://doi.acm.org/10.1145/1163641.1163644.

[23] Robert C. Kirby. Algorithm 839: Fiat, a new paradigm for computing finite element basis functions. *ACM Transactions on Mathematical Software*, 30(4):502–516, December 2004. ISSN 0098-3500. doi: 10.1145/1039813.1039820. URL http://doi.acm.org/10.1145/1039813.1039820.

[24] M. S. Alnaes, A. Logg, K-A. Mardal, O. Skavhaug, and H.P. Langtangen. Unified framework for finite element assembly. *International Journal of Computational Science and Engineering*, 4(4):231–244, 2009. doi: 10.1145/1039813.1039820. URL http://dx.doi.org/10.1145/1039813.1039820.

[25] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96. IEEE Computer

Society, 1996. ISBN 0-89791-854-1. doi: 10.1145/369028.369103. URL http://dx.doi.org/10.1145/369028.369103.

[26] Johan Hoffman, Johan Jansson, and Niclas Jansson. Simulation of 3D unsteady incompressible flow past a NACA 0012 wing section. Technical Report KTH-CTL-4023, KTH Royal Institute of Technology, October 2011. URL http://www.publ.kth.se/trita/ctl-4/023/.

Appendix C

Lists

C.1 List of Figures

C.2 List of Tables

C.3 List of Algorithms

TRITA-MAT-E 2013:37
ISRN-KTH/MAT/E—13/37-SE

www.kth.se