M0004226TP

Optimising Subdomain Aspect Ratios for Parallel Load Balancing

Frank Schlimbach

A thesis submitted in partial fulfillment of the requirements of the University of Greenwich for the Degree of Doctor of Philosophy

July 2000

Abstract

In parallel adaptive Finite Element simulations the work load on the individual processors can change frequently. To (re)distribute the load evenly over the processors a load balancing heuristic is needed. Common strategies try to minimise subdomain dependencies by minimising the number of cut edges in the partition. For many solvers this is the most influential factor. However for example, for certain preconditioned Conjugate Gradient solvers this cutsize can play only a minor role, but their convergence can be highly dependent on the subdomain shapes. Degenerated subdomain shapes can cause them to need significantly more iterations to converge. Common heuristics often fail to address these requirements. In this thesis a new strategy is introduced which directly addresses the problem of generating and conserving reasonably good subdomain shapes while balancing the load in a dynamically changing Finite Element Simulation. A new definition of Aspect Ratio is presented which assesses subdomain shapes. The common methodology of using adjacency information to select the best elements to be migrated is not considered since it is not necessarily related to the subdomain shapes. Instead, geometric data is used to formulate several cost functions to rate elements in terms of their suitability to be migrated. The well known diffusive and Generalised Dimension Exchange methods which calculate the necessary load flow are enhanced by weighting the subdomain edges in order to influence their impact on the resulting partition positively. The results of comprehensive tests are presented and demonstrate that the proposed methods are competitive with state-of-the-art load balancing tools.

I certify that this work has not been accepted in substance for any degree, and is not concurrently submitted for any degree other than that of Doctor of Philosophy (PhD) of the University of Greenwich. I also declare that this work is the result of my own investigations except where otherwise stated.

[illegible]

Contents

Chapter 1

Introduction

In recent years parallel processing has developed into one of the most important and powerful ways to solve very large problems. These problems can be large in terms of the amount of computational work, the amount of data to be processed or both. Often conventional single processor machines are not capable of solving these problems due to their limited memory and/or power. Nowadays parallel machines or workstation clusters can, however, cope with problems that some years ago were rated as too large.

One major field of very large problems are scientific applications (e.g. see the many papers in several conference series [1, 2, 3]). They often need to solve large partial differential equations (PDE) which model physical systems. One of the most popular approaches for this is the Finite Element Method (FEM). This technique splits a continuous problem into a finite number of discrete subproblems, the finite elements. The discrete mesh thus only approximates the continuous problem. If the mesh is very fine (i.e. is built with many small elements) the problem more closely approximates to the continuous problem than a coarse mesh. However, uniformly refined meshes can easily be too large to be solved. Hence some FEM simulation tools adaptively refine the mesh in several stages. After first solving the problem on a coarse mesh and estimating an error, the mesh is refined on those parts that have large errors and might be coarsened where the solution is very exact. These adaptive methods allow good solutions without generating many elements that consume redundant space as well as computation time.

One of the big advantages of the FEM is its capability for parallelisation. To apply the FEM in a parallel environment the computational work has to be distributed over the processors. For that the mesh is usually divided into the required number of subdomains and iteratively the subproblems are solved individually in parallel and finally combined to give a global solution. Solvers following this approach are called domain decomposition (DD) based solvers. The partitioning of a mesh should produce subdomains that have the same computational size. This is not a trivial task since not all possible partitions are equally suitable. Besides the work load other criteria affect the quality of a partition. The common methodology of what is a good partition is dominated by the number of dependencies between subdomains. Dependencies increase the communication volume of the parallel application. Thus minimising the dependencies minimises the communication costs.

Finding a decomposition of the mesh with some optimisation criterion is known as the *partitioning problem*. It has been shown to be NP-complete for optimising the number of interprocessor dependencies or cut edges [43] and hence it is very unlikely that an algorithm can be found which determines an exact solution in a tolerable amount of time. When an adaptive FEM simulation is run in parallel, the load balance is usually destroyed after each refinement/coarsening step. For an optimal use of the parallel computational resources the load should be balanced after each refinement step. However, a load balancing algorithm that consumes more time than it saves for the solver is obviously undesirable. Therefore a heuristic restoring the balance should be as efficient as possible.

In recent years solvers have been developed that are very sensitive to the subdomain structures [11, 40, 95]. Degenerated subdomain shapes cause them to converge very slowly, whereas for example partitionings into square-like subdomains lead to a very fast convergence. For such solvers, optimising the partition only for a low communication overhead seems not to be very helpful. Some attempts have been made recently to include the shapes of subdomains in the partitioning strategy [40, 94, 101]. However, up to now no parallel partitioning or load balancing heuristic existed that explicitly and exclusively optimises the subdomain shapes.

The impact of high communication cost on the running time is highly dependent on the the actual hardware on which the parallel application is executed. Not only the physical capability to communicate efficiently is important but also the mapping of the logical decomposition onto the actual topology of the processors in the parallel machine. The mapping problem is the more important the less efficient the communication infrastructure of the hardware is. Mapping is a large field of research of its own [25, 109] and is not dealt with in this work.

This thesis introduces a new heuristic for balancing load in a dynamically changing FEM simulation for parallel distributed memory machines. It does not consider the conventional paradigm of optimising cutsize but tries to create or maintain subdomains with good shapes. The parallel algorithm and a sequential version were implemented in an adaptive FEM environment developed at the University of Paderborn [21]. After introducing definitions and notations to be used in the thesis, chapter 2 will give a short introduction into the field of partitioning and load balancing, describe their aims and problems in parallelisation and motivate subdomain shape optimisation. In chapter three, the most important parallelisable partitioning and load balancing strategies are described and algorithms are introduced that are needed for the understanding of the following chapters. Chapter 4 and 5 then present the new heuristic and some implementational aspects. In chapter 6 the suggested algorithms are finally tested and compared.

Definitions and notation

This work is restricted to 2D and therefore all definitions will be given for two dimensions only (however see section 4.6 for ideas on extending the work to 3D).

- $coord(a)$ denotes the *Cartesian coordinates* of an object a. Cartesian coordinates consist of the two values x, y; $\overrightarrow{coord(a)}$ represents the vector from the origin to $coord(a)$.
- $dist(a, b)$ specifies the Euclidean distance of two coordinates $a = (x_a, y_a)$ and $b = (x_b, y_b)$ and is defined as

$$dist(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}.$$

  If a and/or b are objects rather than coordinates, $dist(a, b)$ denotes the short notation for the geometric distance $dist(coord(a), coord(b))$ of their coordinates.
- A graph $G = (V, E)$ consists of a set of vertices $V = \{v_0, \ldots, v_{n-1}\}$ and a set of edges $E = \{ed | ed \in V \times V\}$. Each edge $ed = (v_a, v_b)$ connects two vertices $v_a, v_b \in V$. If every pair of edges $ed_i = (v_a, v_b)$ and $ed_j = (v_b, v_a)$ each connecting the same vertices v_a and v_b are identical, the graph is called undirected, otherwise it is directed. A weighted Graph $G = (V, E, w, c)$ is a Graph, where the function $w : V \mapsto R$ assigns a weight to each vertex $v_i \in V$ and $c : E \mapsto R$ assigns a weight to each edge $ed_i \in E$. If not explicitly declared otherwise a graph G will be undirected and unweighted.

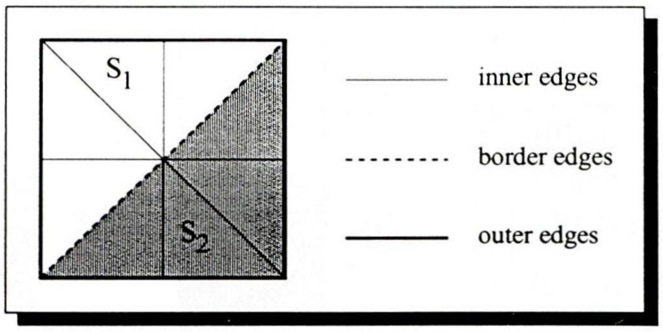


Figure 1.1: Edge types

- A *path* $p \subseteq E$ in $G = (V, E)$ is a sequence of edges $p = (ed_0, \dots, ed_n)$ where for each $0 \leq i < n$ $ed_i = (v_i, v_{i+1})$ and $ed_{i+1} = (v_{i+1}, v_{i+2})$ share one vertex. The *length* of p is defined by its number of edges.
- The *diameter* of a graph is the smallest value n that for every pair of vertices $v_a, v_b \in V$ there is a path from v_a to v_b with length $\leq n$.
- The *cardinality* of a graph is defined by its number of edges.
- A *mesh* is a graph with geometric information, in other words each vertex is assigned coordinates.
- A *Finite Element Mesh* is a mesh $M = (V, E, L)$ where the vertices build a set of elements L. These elements usually are triangles or rectangles and thus consist of three or four vertices and edges. If not explicitly declared otherwise a mesh will denote a triangulated Finite Element Mesh.
- A *subdomain* will describe a part of a divided mesh or graph. To avoid misunderstanding the term *partition* will be used to denote the decomposition of the mesh/graph only.
- The *area* of an triangular element consisting of the three vertices a, b and c is calculated as

$$A_e = \frac{1}{2} \cdot |(x_a \cdot (y_b - y_c) + x_b \cdot (y_c - y_a) + x_c \cdot (y_a - y_b))|,$$

  and its *centre (of mass)* as

$$coord(e) = \frac{1}{3} \cdot (coord(a) + coord(b) + coord(c)).$$

- In the following the *centre* or *coordinates* of a subdomain will be the *centre of mass* of that subdomain. The centre C_p of subdomain p is calculated as follows:

$$C_p = \frac{\sum_{e \in L_p} coord(e) \cdot area(e)}{\sum_{e \in L_p} area(e)}$$

  where L_p are the elements of subdomain p.

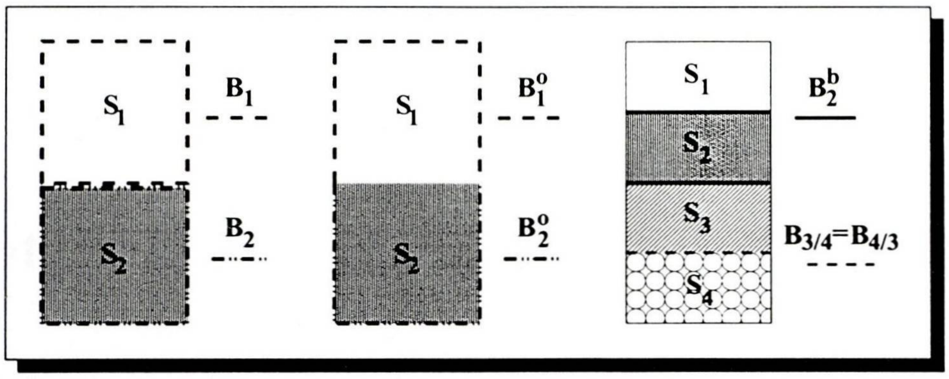


Figure 1.2: Definitions of B_x, B_x^o and B_x^b, B_x^y

- An edge can be an *inner edge*, a *border edge* or an *outer edge*. If the edge lies on the exterior (of only one element) it is positioned on the surface of the domain and thus is an outer edge. An edge is called a border edge, if the two adjacent elements are in different subdomains. All other edges are inner edges (see figure 1.1).
- Symbols: N denotes the number of elements in a mesh or the number of vertices in a graph and P the number of subdomains of a partition. Usually e will refer to an element, v to a vertex, ed to an edge and S to a subdomain. A_i will denote the area of subdomain i. The area of an element e will be referred to similarly as A_e. The boundary length of partition S_i will be known as B_i, its sum of lengths of outer edges as B_i^o and its sum of lengths of border edges as B_i^b. The border length between two subdomains S_1 and S_2 will be shortened to $B_{1/2}$ or $B_{2/1}$ (see figure 1.2). The same border notations apply for elements, but with a lower case b. Additionally the sum of lengths of the inner edges of an element e will be denoted as b_e^i.
- A *subdomain graph* $SG = (V_{SG}, E_{SG})$ of a partitioned mesh is a graph where the vertices $v_i \in V_{SG}$ represent the subdomains and the edges $ed_j = (S_a, S_b) \in E_{SG}$ represent borders between the subdomains S_a and S_b. S_a and S_b are called *neighbouring* or *adjacent* to each other if they share at least one edge: $(S_a, S_b) \in E_{SG}$.
- In data parallel environments some kind of communication between the processes is usually unavoidable. In the following a communication will be denoted as *global*, if it involves other processors than those adjacent to the communicating process and *local*, if it is restricted to only its neighbours. It is assumed that interprocessor processor communication costs are homogeneous.

Chapter 2

Partitioning and Load Balancing

2.1 Basics

Many parallel algorithms share similar problems which are characteristic for parallel optimisation applications and do not occur for their sequential versions. In the Single Program Multiple Data paradigm (SPMD) each processor executes the same program but on different data. Thus the data which is to be processed has to be distributed over the processors. When the Finite Element Method is parallelised, often the data to be distributed is some representation of the mesh and domain decomposition based solvers are applied. Usually the mesh has to be divided into as many equal sized sub-meshes as processors are available. The size of a sub-mesh in this case is its workload. Finite Element analysis is element based and hence the number of elements in a mesh usually correlates with the workload for FEM simulations reasonably well. It will therefore be assumed that the sizes of a mesh and its subdomains can be approximated by its number of elements.

Not every possible partition is equally suitable and, combined with some optimisation criterion, the problem of finding an optimal partition is known as the *partitioning problem* and has been shown to be NP-complete [43]. If the mesh is partitioned into two subdomains it is known as a *bisection* and if decomposed into k parts as a *k-partitioning*.

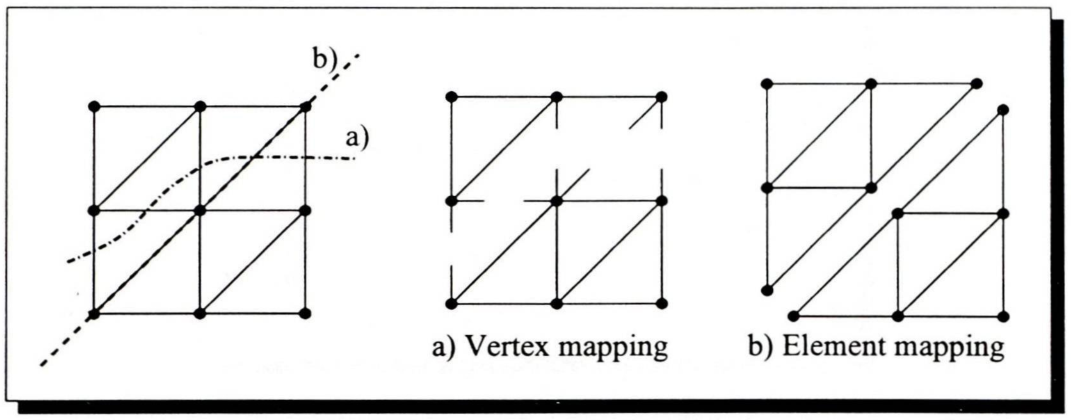


Figure 2.1: Vertex and element mapping

2.1.1 Partitioning

The partitioning problem can be interpreted as a mapping problem

$$\mathfrak{E} \mapsto \mathfrak{S}.$$

The set of elements or vertices $\mathfrak{E}$ is mapped to a set of subdomains $\mathfrak{S}$. Thus the subdomains can be viewed as a set of vertices or elements. Assuming a mesh is given it makes a substantial difference whether the vertices or the elements are being mapped. Figure 2.1 shows a mesh which is decomposed into two subdomains via vertex and element mapping. Subdomains built out of elements (b) instead of vertices (a) results in cuts that go along the edges instead of across the edges and no edges are cut. In principal the two methods are similar since the element mapping is a vertex mapping in the *element graph*. The element graph is constructed by representing the elements by vertices and connecting each two of these vertices with an edge if their corresponding elements share a vertice in the original mesh (see figure 2.2). Every element is assigned a centre, which is its centre of mass computed with the coordinates of their vertices (see chapter 1).

Element mapping implies a problem which can be seen in example (b) of figure 2.1: the vertices and edges on the borders must be stored on all participating processors (vertices can participate in more than one cut!). Practically they will be assigned to one of the subdomains and duplicated on the others.

Although common partitioning strategies talk about vertex mapping the representation as an element mapping is preferred in the remaining of this thesis, because it shows the actual process for FEM applications much more clearly.

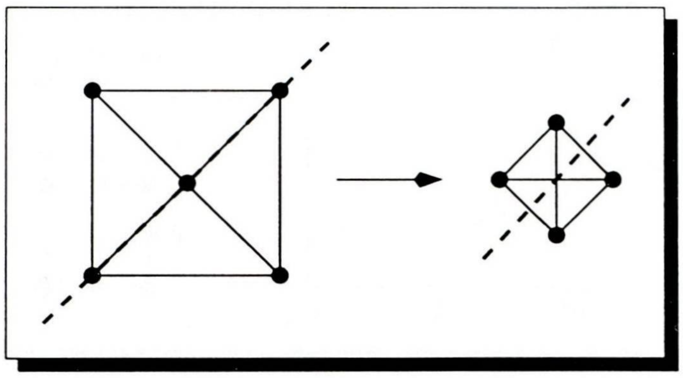


Figure 2.2: An element graph

2.1.2 Local minima

One common way to categorise optimisation algorithms in general is to group them into *local* and *global* strategies. The local algorithms usually are employed to improve a given solution whereas global algorithms determine the solution from scratch. In parallel applications the processors have often only limited knowledge of the problem. Thus global strategies sometimes cannot be solved in a truly parallel fashion, but local algorithms usually offer a high potential to be parallelised.

Defining the *state space* as the set of all valid solutions, a local algorithm would start at an arbitrary state and move through the state space to find an optimal solution. The algorithm cannot move arbitrarily through the state space. A transformation function defines how one solution can be transformed into another. Usually those functions define a small change in the given solution. States that can be directly transformed into one another are called neighbours. The solutions combined with the neighbourhood structure can be represented as a graph, the *state graph*, where solutions are vertices and each two neighbouring solutions are connected with an edge.

One of the simplest types of local strategy is the *greedy* algorithm, which can be applied to many optimisation problems. It starts with an arbitrary solution and follows that edge in the state graph that results in the best neighbour solution. From there it continues choosing the best neighbour until no improving neighbours exist.

One of the major problems of local algorithms are *local minimum traps.* In figure 2.3 a possible state graph and a diagram is given. In the diagram the states are listed on the x-axis and their cost on the y-axis. Assuming a minimisation problem state 5 represents the optimum. If the local algorithm started from state 9, say, it can choose to move to state 8 or 10 since both have the same cost. If it

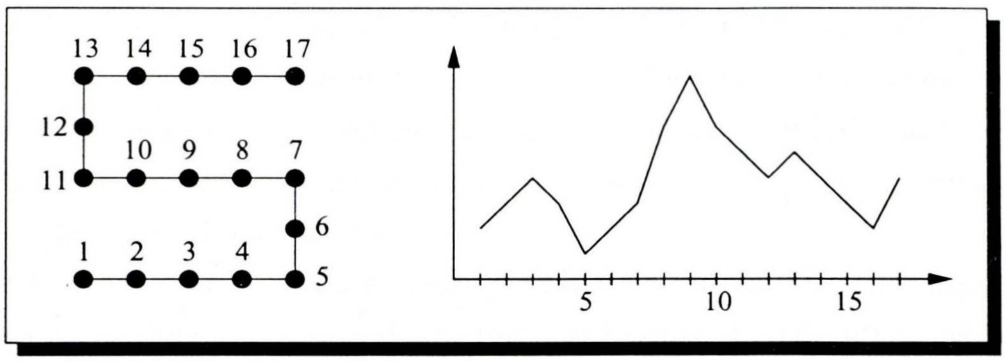


Figure 2.3: A state graph and its state costs

chooses 10 it will move down to state 12 where no neighbour exists that has a lower cost. However, if it started moving to 8 it reaches the optimum. A state such as number 12 that has no neighbours with lower costs is called *local minimum.* The example shows clearly that local minima not necessarily are global minima. Local algorithms often have problems to escape those traps, since the limited knowledge of neighbouring solutions makes it impossible to decide whether it is a global or a local minimum.

2.2 Optimisation criteria

What makes the partitioning problem NP-complete is the optimisation criterion. This criterion is responsible for the quality of the decomposition. There is no uniform answer to the question what is a 'good' decomposition. Of course the runtime of the complete simulation is the effective measure but assessing the decomposition afterwards is not of much help when trying to generate a good partition. Some related criterion has to be found that can be used while the partitioner is running. Although the number of inter-subdomain dependencies has developed to the commonly accepted criterion other assessments have been and still are investigated such as the cardinality of the inter-subdomain connectivity [12, 109], spatial connectivity within subdomains [12] and subdomain shapes [31, 40, 95]. Since this thesis addresses subdomain shapes the importance and influence of domain shapes is described in detail in the following and Aspect Ratio is compared with cutsize.

2.2.1 Cutsize

Up to now the common belief is that the runtime is dominated by the inter-subdomain dependencies. It was inspired by the aim of keeping the total communication volume as low as possible. Inter-subdomain dependencies exist where two elements in different subdomains share an edge. When the partitioning strategy generates edge separators the edges on the subdomain boundaries are virtually 'cut'. The number of cut edges is therefore called *cutsize.* Conventional partitioning strategies optimise cutsize.

In [53] Hodgson and Jimack compare different partitioning strategies on a linear and a non-linear solver and conclude that conventional cutsize as defined above is not the best method of assessment of a partition for their solvers. They find the largest number of boundary vertices of any subdomain correlates best with the run times of the solvers. Their mathematical scalability analysis confirms this observation and shows that circular or square type subdomains represent the optimal cases.

2.2.2 Aspect Ratios

A common term for assessing shapes is *Aspect Ratio* (AR). Many different definitions of Aspect Ratio occur in literature. They all try to specify a method that gives a suitable rating of shapes but differ significantly in their behaviour due to the difference in the approaches and their objectives. Most important for the choice of the appropriate AR is its suitability to reflect the actual subdomain shape for all possible scenarios. In other words the AR should guarantee comparability between different shapes. Another important issue is the time complexity of algorithms that are needed to use the Aspect Ratio during the load balancing process. Two aspects of time complexity need to be addressed: the initial computation of the AR and the update operation when the subdomain shape changes. The subdomain shapes can change during the load balancing process when elements migrate between the subdomains. In the following a motivation for optimising subdomain Aspect Ratio is given and the most important ways to define AR are presented. Their suitability and time complexity are analysed and compared, where the number of elements (per subdomain) is used as the problem size for the time analysis.

Motivation

The need for Aspect Ratio optimisation is motivated by the requirements of a class of solution techniques. Nowadays important solvers exist that are influenced by the subdomain shapes. Iterative methods like the *Conjugate Gradient* (CG) algorithm [90] together with domain decomposition preconditioners (DD-PC),

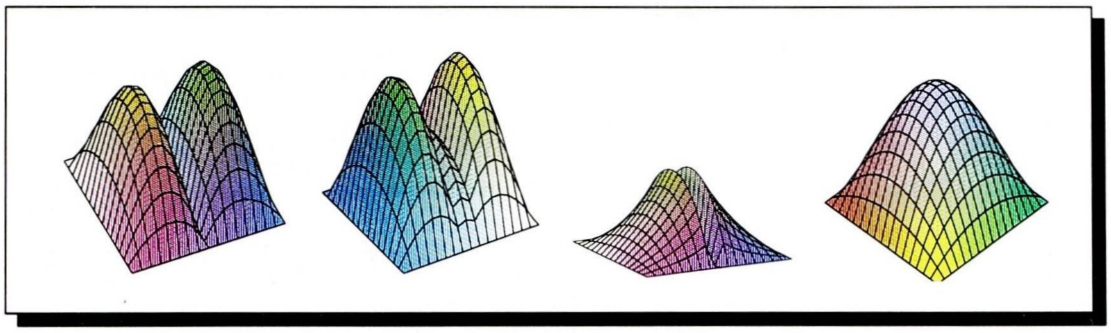


Figure 2.4: The three steps of a DD-PCG algorithm (left to right)

e.g. [11, 54, 67] take advantage of the partition of a mesh intò subdomains. Usually in a first step the original problem is solved on each subdomain by the processes independently. This local step excludes the subdomain boundaries or artificial boundary conditions are imposed on them. Therefore the subdomain solutions are independent of each other and can be determined in parallel without any communication between processors. In a second step an interface problem is constructed and solved. This second step obviously needs communication and the communication volume is largely dependent on the size of the interface problem. The two solution(set)s are then put together and give the new search direction in the CG algorithm. This process is then iterated until convergence.

As an example figure 2.4 shows the function of the *Conjugate Gradient Boundary Iteration Method* [11]. The left hand figure shows two independent subsolutions. The next picture adds the interface problem which is dependent on the jump over the boundaries. The interface problem gives new conditions on the inner boundaries for the next step of subdomain solutions (mid-right figure). The right hand picture shows the solution of the whole problem after 4 iterations.

The time needed by such a preconditioned CG solver is determined by two factors, the maximum time needed by any of the subdomain solutions and the number of iterations of the global CG. Both are at least partially determined by the shape of the subdomains. It is commonly known that the local convergence is highly dependent on the shapes of the elements [40, 54]. Bad element shapes result in badly conditioned operators. Most domain decomposition methods treat subdomains as as sort of super-elements when constructing the interface problem. This super-element interpretation suggests that the solution of the interface problem is dependent on the subdomain shapes [40] in the same way that the finite element method is dependent on the mesh quality (or the element Aspect Ratio). Hence the condition number of the interface matrix (a value expressing how 'easy'

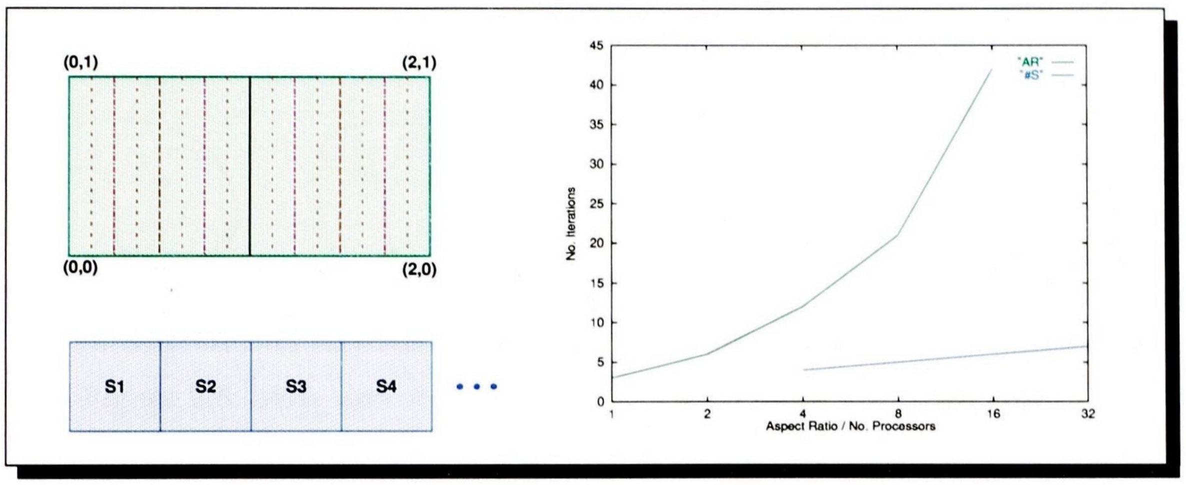


Figure 2.5: Subdomain shapes on a PCGS

it is to solve) will increase [40, 94] and the global convergence slows down with subdomain shapes getting worse.

Tests, which are presented later in this section, suggest that the shapes of the subdomains influence the convergence of those solvers immensely. An expressive example is also given in figure 2.5 where the number of subdomains is increased in two different ways. The upper example is a fixed problem which is simply decomposed by recursively applying a coordinate based partitioning algorithm, cutting perpendicular to the x-axis to produce parallel stripes. The lower example has fixed subdomain shapes and the domain is generated by adding square subdomains per processor and hence the number of unknowns increases linearly with the number of subdomains. The diagram on the right hand shows the number of iterations a domain decomposition based preconditioned Conjugent Gradient solver (DD-PCGS) needed to solve a heat distribution problem for these two examples. Although for the lower scenario the problem size increases with the number of subdomains, the number of iterations increases only very slowly with the number of subdomains. In contrast the necessary number iterations for the upper example this number rises dramatically for the lower example in spite of the constant problem size. Surprisingly the number of iterations increases with the factor that is equal to the ratio between the longest and shortest side of the rectangles. This demonstrates in an impressive way that the subdomain shapes could be an essential criterion when partitioning for this type of solver.

In the following five definitions of Aspect Ratio are presented which are known from literature. After a comparison a new formulation of Aspect Ratio is introduced and a the effect of several of these definitions on a DD-PCGS is examined.

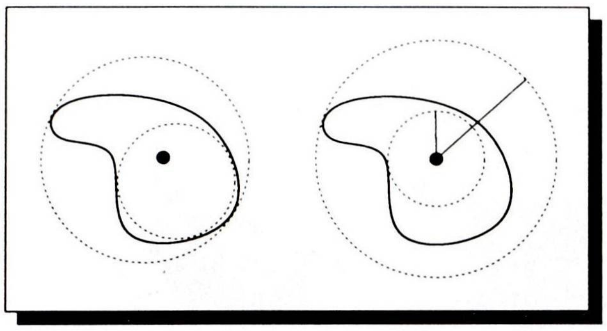


Figure 2.6: AR_1 and AR_2

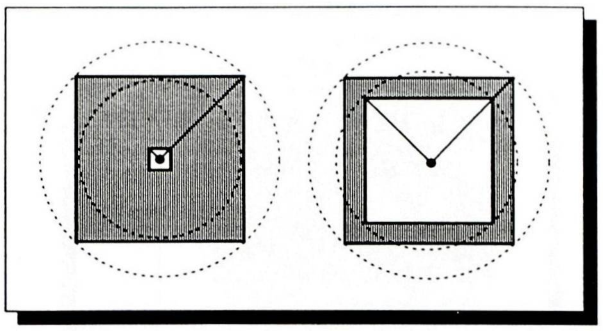


Figure 2.7: AR_2 fails

AR_1 of minimal outer over maximal inner circles

This formulation of Aspect Ratio was invented to assess the quality of triangles in mesh triangulations [75]. It is defined by the ratio between the diameter of the smallest circle that fully includes the given object, in this case a subdomain, and the diameter of the biggest circle that can be placed entirely inside the object (neither circle cuts an outer side/edge!). A better scaling can be achieved by using the area of the two circles or the square of the diameter instead of simply their diameter. Obviously this Aspect Ratio is optimal for circles. For triangles and rectangles this is a very convenient definition since it can be found easily for these cases. For polygons with many sides it becomes much more difficult. The more sides the polygon has, the more difficult it is. The centres of the two circles are usually not the same as the centre of the polygon or even of an arbitrary shape. The left hand example in figure 2.6 illustrates this strategy. AR_1 returns very good assessment of shapes but since a subdomain is usually not a triangle or rectangle the two circles cannot be found very quickly.

AR_2 of minimal over maximal distances

A simplification of AR_1 assumes both centres of the two circles to be the same as the centre of the subdomain. AR_2 is now defined as the ratio between the distances of the furthest and nearest border element to the subdomain centre (see right hand example in figure 2.6) [73]. Again, using the area instead of the diameters gives a better scaling. Usually this method produces good assessments too, but in cases where the centre is not inside of one of its elements this approach fails completely. Figure 2.7 shows that AR_2 might not be appicable to such scenarios. Allthough in both examples the centres of both circles are located inside holes of the domains, the smaller circle (largest circle that can be positioned

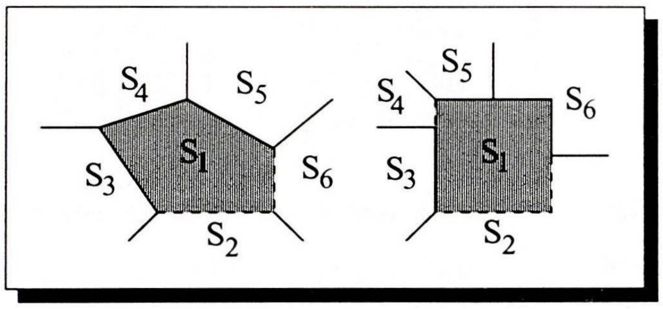


Figure 2.8: AR_3

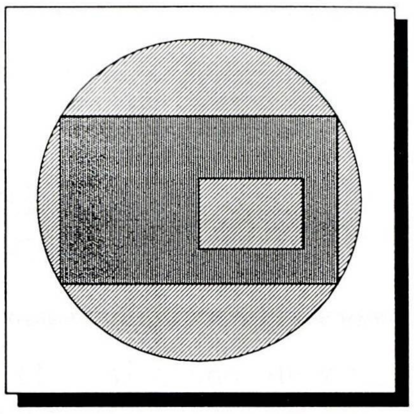


Figure 2.9: AR_5

entirely inside the polygon) exists only for the right hand example. The hole of the left hand example is that big that the smallest circle that can be placed outside the hole and inside the domain is already too large and cuts outer borders.

AR_3 of longest over shortest borders

This approach also originated in the field of mesh generation. It assesses a polygon by the ratio of its longest over its shortest sides [73]. However, a subdomain has in the worst case as many sides as it has outer and border edges and usually they are about the same length. Furthermore finding sets of edges that build a side of the subdomain polygon can be very costly. The natural sides of a subdomain are its borders to other subdomains and so AR_3 can be accordingly defined by the lengths of its borders. However, when dealing with unstructured meshes the shapes of subdomains almost never correlate with the subdomain borders. On the left hand of figure 2.8 a well structured example is shown. The shortest border is to partition S_6 and the longest to S_2 and so AR_3 would rate this as a subdomain with good shape. In contrast the right hand example in figure 2.8 shows a typical scenario. The ratio between longest (with S_2) and shortest (with S_4) border is quite large although the subdomain shape is a square and thus intuitively good.

However, by alternatively replacing the geometric lengths of inter-subdomain borders by the number of edges, vertices or elements on the borders AR_3 might be useful to asses the influence of unbalanced dependencies between subdomains. In the following AR_{3a} will denote the ratio of border lengths and AR_{3b} the ratio of the numbers of elements.

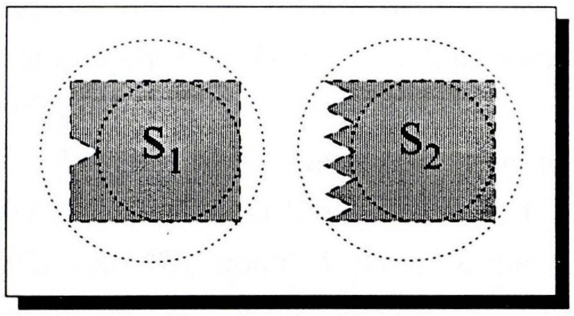


Figure 2.10: ARs on jagged borders

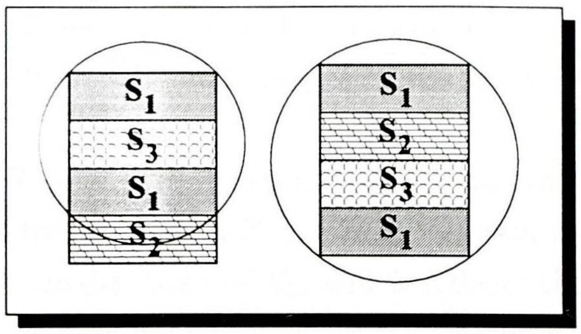


Figure 2.11: ARs and disconnected subdomains

AR_4 of distance of vertices

In [94] the subdomain Aspect Ratio of subdomain S_s in 2D is defined as

$$AR_4 = \sum_{v \in S} (x_v - x_s)^2 + (y_v - y_s)^2,$$

where x_v, y_v denote the x- and y-coordinates of vertex v and x_s, y_s the centre of subdomain S_s, as defined in chapter 1. This definition is similar to the sum of distances of all vertices/elements in the subdomain. AR_4 can be easily computed and maintained but as discussed below, it is not capable of comparing two subdomain shapes properly.

AR_5 of outer circle over area

AR_5 directly compares the given shape with a suitable circle. It is defined by the ratio of the area of the smallest circle that fully includes the subdomain (see AR_1) and the area of the subdomain itself [40]. The example in figure 2.9 shows that AR_5 is independent of the position of holes in the subdomain.

Comparison

AR_4 is unsuitable to compare subdomains with each other, since it is dependent on the geometric distances and number of vertices and is not normalised. Hence subdomains with a large area or many vertices will automatically get a bad AR. Even normalising it by dividing it by the number of vertices will only partially solve this problem. However, Vanderstraeten et al. [96] included the sum of these ARs into their global cost function and thus could help optimising the subdomain shapes by trying to distribute the centres evenly. As mentioned above

AR_3 is unsuitable for unstructured meshes and AR_2 fails for certain scenarios completely (see figure 2.7). The assessments of AR_2 and AR_3 are thus not very reliable.

The remaining Aspect Rations are AR_1 and AR_5. Figure 2.10 shows, that AR_1 would assign the same Aspect Ratio to subdomain S_1 as to subdomain S_2 whereas AR_5 would devalue S_2 due to the smaller area of S_2, which reflects the actual shapes better. In scenarios where a subdomain consists of two (or more) physically disconnected parts as subdomain S_1 in figure 2.11 the Aspect Ratio should be independent of the position of the individual parts, although intuitively such a shape should be avoided. The example in figure 2.11 in fact shows that apart from AR_{3a} all Aspect Ratios presented return different values for S_1 for the two scenarios although in both decompositions S_1 consists of the same parts. The dependence on the subdomain centre, either due to the computation of circles or due to the use of distances of vertices to it, makes this unavoidable.

Another disadvantage of the Aspect Ratios that use circles is that they are very costly to determine. The fastest known (to the author) algorithm for finding one of the two circles needed for AR_1 and AR_5 uses Voronoi diagrams and needs $O(c \cdot N \cdot \log N)$ time where N denoting the number of vertices (which is a constant multiple of the number of elements) and a large c [9, 82]. An update procedure has an worst case time complexity of $O(N \cdot \log N)$ as well, which is too costly if a frequent update is necessary.

In the next section a new definition of Aspect Ratio is introduced which does not have the disadvantages discussed above.

2.2.3 Aspect Ratio of boundary length over area

To avoid the dependency on the subdomain centres only centre-independent characteristics are used in the new formulation of Aspect Ratio. The ratio between the complete boundary length and the area of the subdomain S_s $AR = \frac{B_s}{A_s}$ adapts well to any change in shape and is visualised in figure 2.12. This measure is also used a lot in characterising particulates in chemical, material and minerals engineering [7]. In order to get values that can be interpreted easily some shape has to be chosen as a reference. The question what is a perfect shape for a subdomain is not easily answered. AR_1, AR_2 and AR_5 assume a circle to be the optimum and only allow values $>= 1$ where a circle is assigned 1. Indeed no circumference of any shape can enclose more area than a circle, thus a circle is intuitively a perfect shape. However, in figure 2.13 an example is given which shows that

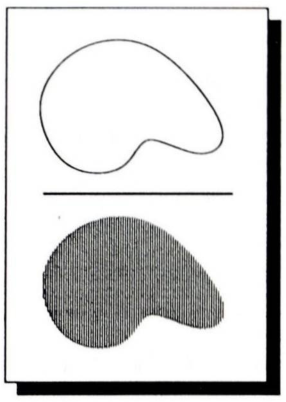


Figure 2.12: New definition of AR

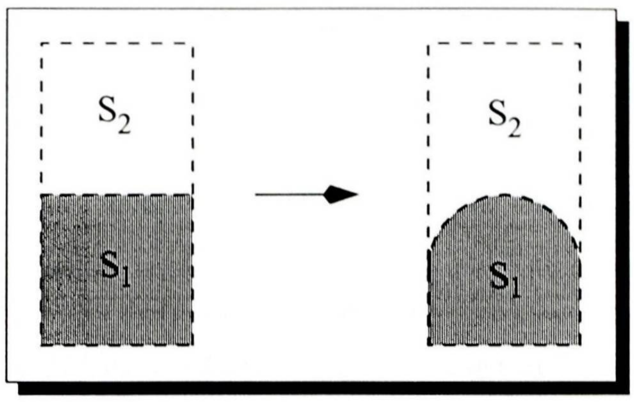


Figure 2.13: Circle as ideal shape

optimising a subdomain (S_1) by making it closer to a circle affects the convexity of the shape of its neighbour (S_2). Thus it could be inferred that a square like subdomain is a better choice for a reference. On the other hand, the concave shape of S_2 obviously influences the global Aspect Ratio and thus a perfect optimiser for global Aspect Ratio would avoid those solutions. Furthermore using the formula for squares as the optimal shape (i.e. assigning 1 to a square) can produce values < 1, e.g. for a circle, whereas the majority will be > 1, which is generally inconvenient for computations and comparisons.

The minimal perimeter B that can enclose a given area A occurs for circles and is $B = 2 \cdot \sqrt{\pi \cdot A}$. Thus a formula guaranteeing values ≥ 1 is

$$AR_{\circ} = \frac{B}{2 \cdot \sqrt{\pi \cdot A}}.$$

The area A of a square is enclosed by a circumference of $B = 4\sqrt{A}$ and thus a formula that considers the square as the ideal shape is

$$AR_{\square} = \frac{B}{4 \cdot \sqrt{A}}.$$

These two definitions can be adjusted to a quadratic scaling, on which some optimisers like Simulated Annealing react positively, [32]. This conforms with the quadratic behaviour of AR_1, AR_2, AR_4 and AR_5 and leads to the final formulations

$$AR = \frac{B^2}{4 \cdot \pi \cdot A}$$

and

$$AR = \frac{B^2}{16 \cdot A}.$$

One major advantage of this new definitions of Aspect Ratio is their independence from any length scale: since $AR_{\Box} \propto AR_{\circ}$ and $AR_{\circ} = \frac{B^2}{16 \cdot A} \propto \frac{B^2}{A}$ for an arbitrary measure of length l the following holds

$$\frac{B^2}{A} \propto \frac{l^2}{l^2} \propto 1$$

Furthermore the new definitions of Aspect Ratio are able to differentiate the influence of jagged borders (see figure 2.10) even better than AR_5 since not only the area becomes smaller with increasing number of jags but also the boundary length increases at the same time. The existence of holes in the domain is also captured as well and their position does not influence the assessment. Furthermore the new formulations of Aspect Ratio are the only useful definitions not influenced by the position of disconnected parts of the subdomain (see figure 2.11), since the total area and boundary lengths are the same wherever the parts are positioned.

In contrast to AR_1 and AR_5 this Aspect Ratios can be determined in linear time, since the area A and boundary length B can be calculated in $O(N)$ time. Furthermore an update can be done in constant time. Note that the computation of the linear Aspect Ratios is slightly more costly than the quadratic versions since usually the square root is a very costly operation, and this might be crucial since as discussed below in section 4.2.2, an update operation includes four versions of this formula.

These observations lead to the assumption that this definitions of Aspect Ratio should be most suitable to assess subdomain shapes. In section 2.2.5 this assumption is backed with tests, that compare the convergence of the solver with different Aspect Ratios and cutsize. In the rest of the thesis AR or the term *Aspect Ratio* will refer to this new definition $AR_{\circ}$.

To give a rough idea how the values of Aspect Ratio are related to regular shapes figure 2.14 shows on the y-axis the Aspect Ratios of rectangles, triangles and ellipses. The respective shapes are constructed with a factor f given on the x-axis. For rectangles it is the factor by which one side is longer as the other, e.g. if $f = 1$ it is a square with an AR of 1.27 and if $f = 4$ the rectangle is four times wider as high (or vice versa) with an AR of 1.99. Triangles are constructed similarly by fixing the length of one side and varying the remaining two with the factor f. The triangle with the best Aspect ratio (1.65) is the one with three equal sides. Note that for $f = 0.5$ the triangle is collapsed to a line, thus the Aspect Ratio increases dramatically when the polygon becomes very flat. For the ellipses f is the factor by which one 'radius' is bigger than the other. Obviously

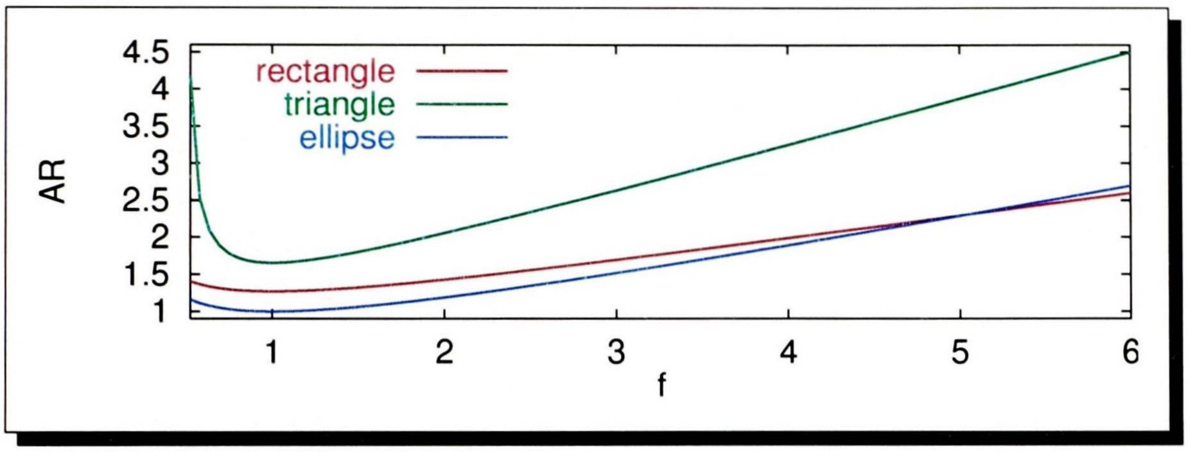


Figure 2.14: ARs of rectangles, triangles and ellipses

for $f = 1$ the ellipse is an circle with the optimal Aspect Ratio of 1. Table 6.1 on page 103 lists the Aspect Ratios for some irregularly shaped domains (see also next section) which are illustrated in the appendix.

Note that the new definition of Aspect Ratio is easily generalised to 3D as

$$AR_{\bullet} = \frac{(area\ of\ surface)^2}{\pi^{2/3} \cdot (6 \cdot volume)^{4/3}}$$

and

$$AR_{\blacksquare} = \frac{(area\ of\ surface)^2}{16 \cdot volume^{4/3}},$$

where $AR_{\bullet}$ considers the sphere as the ideal shape and $AR_{\blacksquare}$ the cube.

2.2.4 Global border length AR_{BL}

One criterion that is closely related to AR and cutsize can be described as optimising global border length. The areas of all subdomains of a decomposition of a given mesh always sum up to the area of the mesh. Thus the only variable value in the global Aspect Ratio is the sum of border lengths, which differ between different partitions. Seeing the area of the mesh as a constant then only leaves the minimisation of this global border length (this also excludes the edges on the boundary of the mesh, since their position cannot be altered). Intuitively this is as well a suitable criterion to assess the average compactness of the subdomains, since a small global border length does not provide the possibility for many ratios of long circumferences and small areas.

AR_{BL} is also closely related to the cutsize, since if all edges have the same length it only counts the number of edges on the borders – the cut edges. Thus this Aspect Ratio can be interpreted as cutsize (of the element graph) where the edges are assigned weights and any conventional partitioner or load balancer that is capable of edge weighted graphs (weighted with floating point numbers) can optimise global border length. See also [101] for another interpretation of AR_{BL}.

2.2.5 Impact on a Domain Decomposition based Preconditioned Conjugate Gradient Solver

Tests

Several tests were made to investigate the influence of different partitions on the convergence of a domain-decomposition-based preconditioner. On four meshes (t60k, uk, whitaker and crack, shown in the appendix on pages 152–155) and three different numbers of subdomains an artificial but very simple problem was investigated. A heat distribution problem with Dirichlet boundary conditions was solved using the conjugate gradient solver supplied by the PETSC library [5]. As the preconditioner the domain decomposition method implemented in the ParPre package [35] has been chosen. The resulting DD-PCGS uses the Schur Complement method with a form of Block Jacobi preconditioning on the interface problem (see [16, 17] for more details).

Tables A.1, A.2 and A.3 on pages 136–137 show the numbers of iterations the solver needed to converge and different Aspect Ratios and cutsize of the applied partitions. Each table is horizontally divided into two parts representing two partitions: the first partition was achieved by optimising Aspect Ratio and in the lower part cutsize was optimised. All partitioning was done using Jostle [98] optimising Aspect Ratio or cutsize respectively. The solver terminated the iteration prematurely for crack divided into 16 subdomains, therefore these iteration numbers are not listed. The three tables are hard to interpret and hence a way to present condensed data is introduced in the following.

Ideally a criterion that assesses a partition should correlate with the run time of the solver applied on this decomposition. However this a difficult task, different criteria such as Aspect Ratio or cutsize that are used to assess a partition can be compared on their ability to predict the convergence of a solver. If for example two decompositions are to be compared where the first is assigned a good quality and the second a low quality, the solver should converge with less iterations on the better partition than on the worse partition.

Given two different partitions on one example (as they are presented in tables A.1–A.3) a *relative difference* in Aspect Ratio, cutsize or number of iterations can be defined as the ratio of one value over the other value (e.g cutsize of partition one divided by the cutsize of partition two). If Aspect Ratio or cutsize correlates perfectly with the number of iterations (the solver needed to converge) the relative difference in cutsize should be equal to the relative difference in the number of iterations. Let I_{AR} and I_{cut} denote the number of iterations needed for the partition generated while optimising Aspect Ratio and cutsize respectively. For any global cost function Υ (AR_1, cutsize etc.) the values for the two partitions of each scenario Υ_{AR} and Υ_{cut} can now be used to define a *deviation factor* ϱ_Υ :

$$\varrho_\Upsilon = \left| \left(1 - \frac{I_{AR} \cdot \Upsilon_{cut}}{I_{cut} \cdot \Upsilon_{AR}} \right) \cdot 100 \right| .$$

Thus ϱ_Υ defines the relative difference in the number of iterations I divided by the relative difference of Υ subtracted from 1 scaled by 100. If ϱ_Υ is small the relative difference of the two partitions correlate very well with the number of iterations the solver needed to converge. The optimal value for ϱ_Υ is 0.

In the following sections different definitions of Aspect Ratio are compared on their correlation with the convergence of the DD-PCGS and the behaviour of cutsize is contrasted to the new definition of Aspect Ratio.

Aspect Ratios

Figures 2.15, 2.16, and 2.17 show the above introduced deviation factors ϱ for several definitions of Aspect Ratio and for cutsize grouped by the number of subdomains and the meshes. The values represent the average over all subdomains per partition. Figure 2.18 summarises the data and gives the average deviation factor for each assessment, derived from the average values as above and additionally the maximal values per partition (the maximal AR or cutsize for any subdomain in each partitioned mesh) respectively.

Generally the diagrams back the above comparison of the different Aspect Ratios. Whereas AR and AR_5 mostly show good correlations the other definitions often produce large factors and only on a few occasions correlate reasonably with the number of iterations. In particular $AR_{3[ab]}$ show extremely bad correlations which conforms with the assumptions made above in sections 2.2.2 and 2.2.3. Also the similarity between AR_5 and AR is reflected in these figures since they show the smallest deviations. However, the new definition indeed correlates best with the number of iterations. AR_2 surprisingly shows better results than AR_1, which might be due to the generally well shaped partitions.

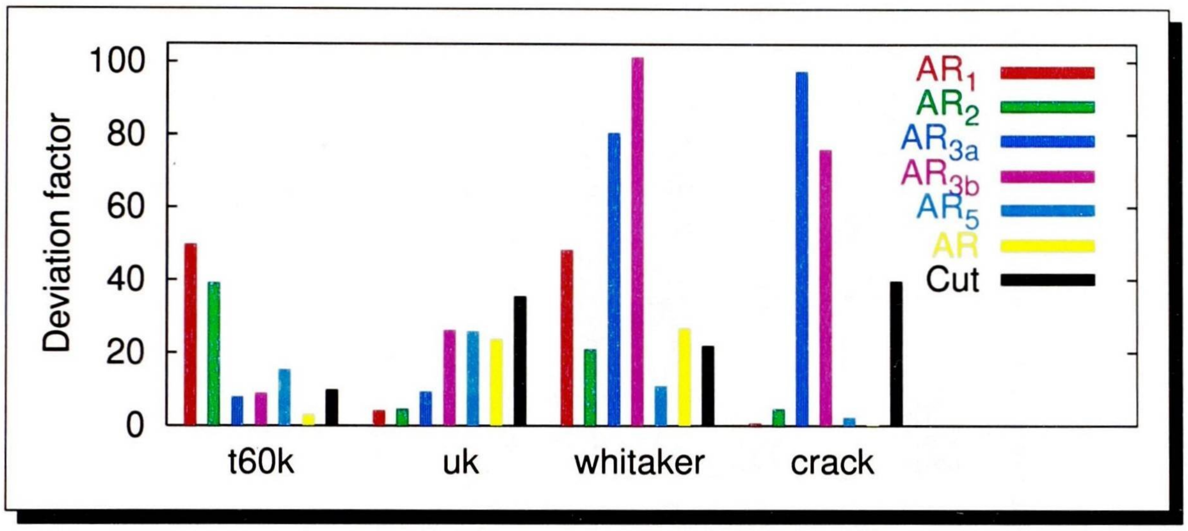


Figure 2.15: Deviation factors on a DD-PCGS for 8 subdomains

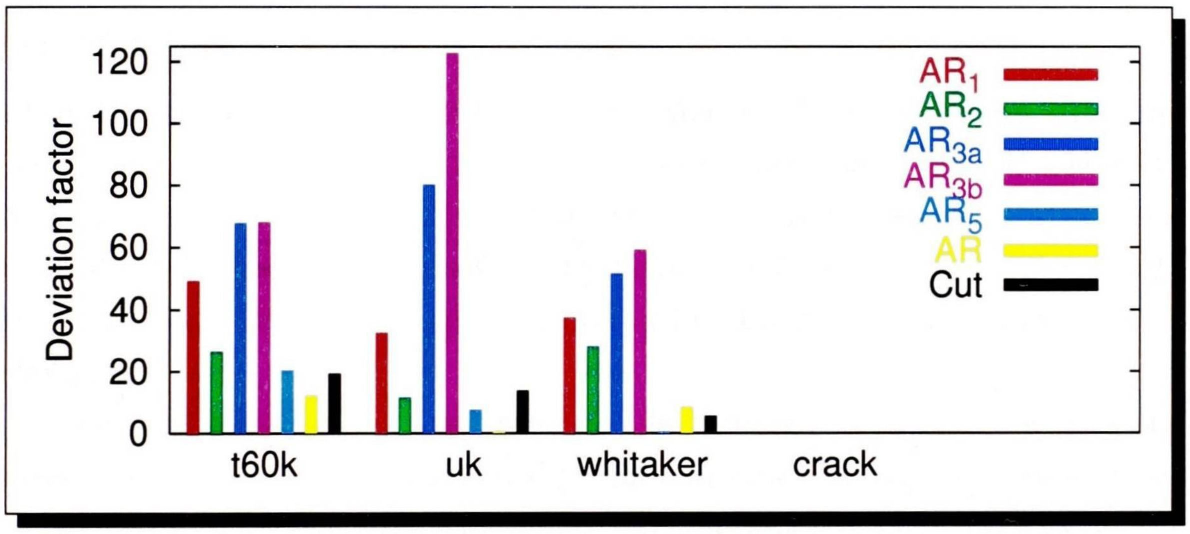


Figure 2.16: Deviation factors on a DD-PCGS for 16 subdomains

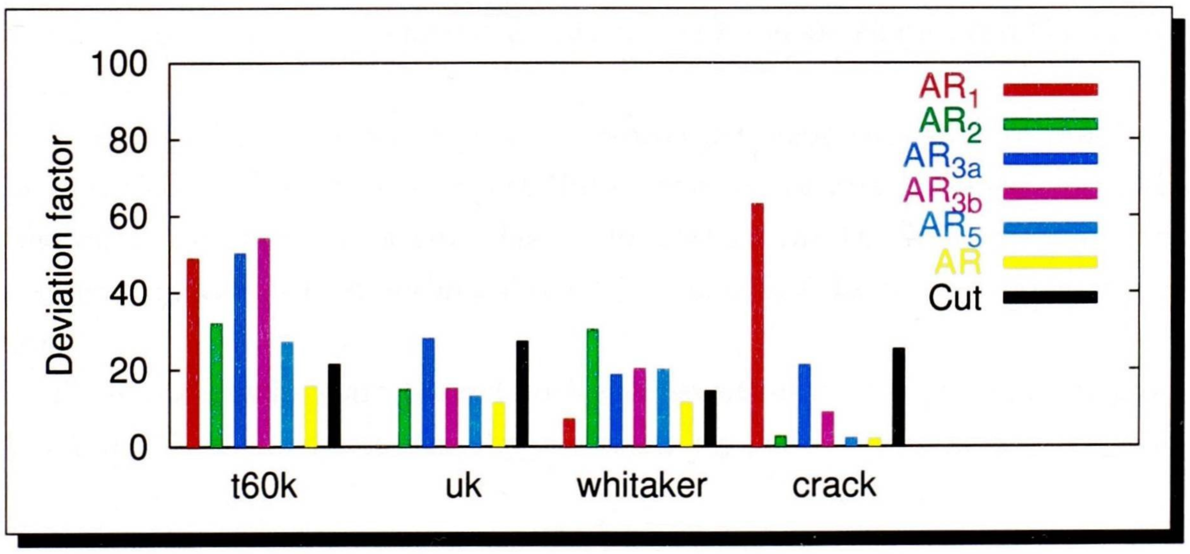


Figure 2.17: Deviation factors on a DD-PCGS for 32 subdomains

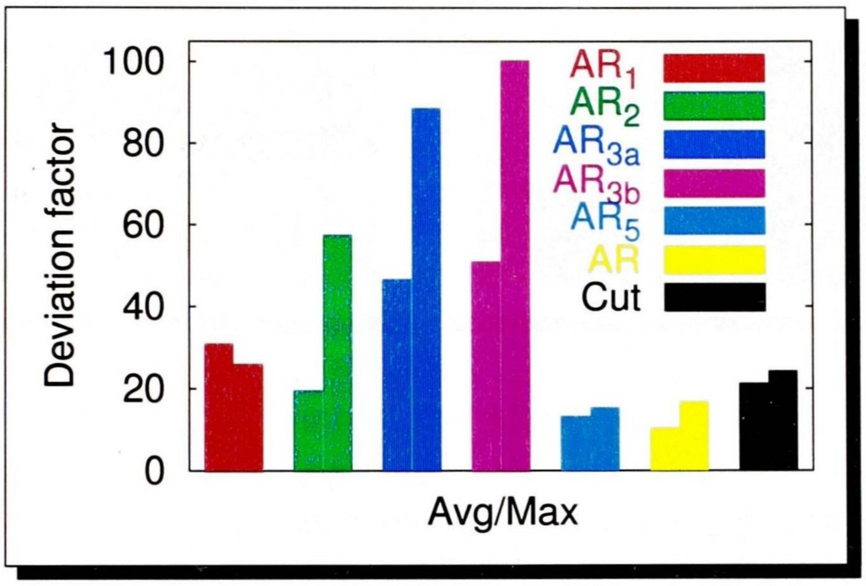


Figure 2.18: Average deviation factors (avg. and max.) on a DD-PCGS

Aspect Ratios versus cutsize

Intuitively a low cutsize might imply a low Aspect Ratio but figure 2.19 shows that this is not necessarily true. If the mesh is partitioned as in the left hand example the total cutsize can be minimised to 2, but the Aspect Ratio is non-optimal (1.99 for both subdomains). On the right hand the partition is optimal in Aspect Ratio (1.27) but the cutsize is doubled to 4 in comparison to the previous example.

To give a more applied motivation for developing a load balancer that optimises Aspect Ratio in contrast to the common aim to keep the number of cut edges to a minimum the values for cutsize are not only included in tables A.1–A.3 but are also given in figures 2.15–2.18. Tables A.1–A.3 show that with only three exceptions the AR-optimiser indeed produced better subdomain shapes but higher number of cut edges than its cutsize-optimising counterpart. This shows that also on unstructured scenarios the optimisation of subdomain shapes is different from minimising cutsize.

In 9 out of 11 examples the solver converged significantly more rapidly on the partition with optimised Aspect Ratio than on the respective decomposition with minimised number of cut edges. On average the DD-PCGS needed 8.6% less iterations on a shape-optimised partition than on a decomposition with lower cutsize.

These observations are backed with the deviation factors shown in diagrams 2.15–2.18. Even though cutsize shows a relatively good correlation (in particular

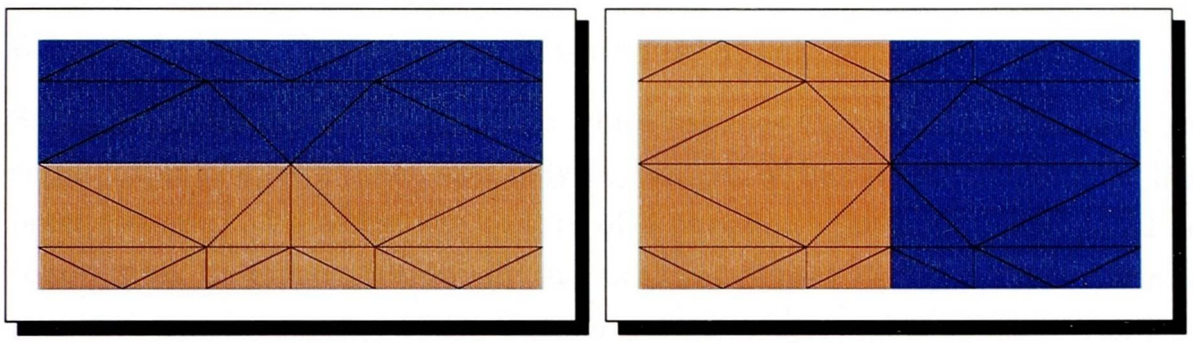


Figure 2.19: Optimal cutsize and Aspect Ratio

in figure 2.18) the extremely high deviation factors of AR_1, AR_2 and AR_3 might give a distorted impression. Comparing only cutsize and the new definition AR discloses that cutsize shows a deviation factor that is more than two times greater than the factor for AR (21.3 and 10.6). The diagrams for the individual scenarios show similar data. In almost all cases ϱ_{AR} is much smaller than ϱ_{cut}. The only two exceptions occur for whitaker divided into 8 and 16 subdomains where cutsize resembles the difference of convergence better, although for 8 subdomains even the factor for cutsize is rather bad.

The data derived from the tests show that for this DD-PCGS it is worth optimising Aspect Ratio rather than cutsize. The experiments on very simple problems suggest that not only the convergence of this type of solver is highly dependent on the way the mesh is partitioned but it also reacts positively on good subdomain shapes. Furthermore the subdomain dependencies seem to play a less significant role than the subdomain Aspect Ratios.

Another strength of shape optimising strategies in comparison to conventional methods is their ability to keep subdomains connected. Common load balancing algorithms optimising cutsize sometimes tend to lose the connectivity of the subdomains if the mesh changes heavily during the simulation. Since they ignore the subdomain geometry the resulting shapes may become less and less circle like and after several refinement steps they might finally fall into several parts. When optimising Aspect Ratio this is less likely since the main aim is to prevent the subdomains from getting degenerated. However, modern approaches optimising cutsize tend to overcome this problem for example by applying the multi-level idea.

2.2.6 Aspect Ratio of partitions and meshes

So far no definition of Aspect Ratio which assesses the partition of a mesh into subdomains has been given and only the shapes of individual subdomains have been discussed. To assess the quality of a partition of a mesh some global criterion must be found. Intuitively the maximal Aspect Ratio of any subdomain should determine the condition of the interface matrix and thus the convergence of the solver. Therefore the maximum Aspect Ratio should be chosen to assess a partition. Because cost functions optimising maxima are notoriously difficult to handle, cutsize optimisers do not optimise for the maximal number of cut edges per subdomain but use the sum of cut edges. For Aspect Ratio the same problem occurs. However figure 2.18 compares for all tests on the DD-PCGS the average deviation factors for the average and maximal values per partition. It discloses that only AR_1 gets better results for the maximal Aspect Ratios in comparison to the average. All other assessments correlate better when their average values are used, in particular the deviation factor for the maximal AR is 61% worse than the one derived from the average values.

Since the average Aspect Ratio seems to correlate better with the needs of this class of solvers and it is furthermore much easier to handle, in the following the global quality of a partition (AR) will be defined by the average Aspect Ratio

$$\frac{\sum_i^P AR_i}{P}.$$

Of course the sum of Aspect Ratios would serve equally well in terms of computational effectivity. However the average value gives a more intuitive impression of the partition independent of the number of subdomains.

Finally the Aspect Ratio of a mesh will denote the Aspect Ratio of the global unpartitioned domain. This can give a hint how complicated the domain is and thus what Aspect Ratio a partition of this mesh might have.

2.3 Static partitioning

A static partitioning strategy decomposes a mesh once from scratch and does not need or use any information of previous partitions. For many applications it is sufficient to partition the mesh once at the start of the simulation. The solver then computes the problem and the simulation ends. For parallel applications where the load dynamically changes static methods usually are not suitable since they can result in inefficient processor usage. For a more detailed description of some important static partitioning strategies see section 3.1.

2.4 Dynamic load balancing

A *load balancing* strategy deals with meshes that dynamically change their load or with variable computational resources. Either the amount of work per unit can change and/or the number of basic computation units changes and/or the available computational resources change. A typical application is a dynamic FEM simulation. During the computation an error is estimated and where the error is too high the FE Mesh is refined. Since the refinement might not be distributed uniformly over the whole mesh a load imbalance can occur. This might make it necessary to repartition the mesh many times during the simulation. Several approaches are possible to solve this problem (see section 3.2). Simply partitioning the mesh from scratch seems not to be the best choice since usually repartitioning can keep the amount of data movement very low in comparison to the cost a new partitioning produces (see section 3.2.2). Thus an incremental partitioning strategy is desirable that keeps data movement as low as possible, restores balance and at the same time optimises some global cost function.

In the following it is assumed that a load balancer balances the number of elements, where any element can be interpreted as an undivisable unit of workload. Even though in more complicated cases elements could represent some individual estimate of workload based on knowledge of the solver or timings, such cases are not considered in this thesis, but see [4] for a detailed discussion of such cases. With this assumption an algorithm for balancing load in a mesh or graph needs to determine

- the number of elements that have to be moved to restore balance and
- which elements must change subdomains.

Both subproblems combined with the applied optimisation criteria (e.g. minimising data movement and optimising global cost) are NP-complete on their own and hence are a large area for research. The problem of finding the number of elements that have to move and the subdomain edges on which these elements are exchanged can be presented as a network flow. Subdomains with load that is above the average are represented by source vertices and those below the threshold by sinks. If the network flow problem is solved, each subdomain knows how many elements it has to send to which neighbour and possibly which neighbour sends elements to it. Some heuristics split the load balancing physically into the two parts of computing the flow and moving elements (see for example section 3.2.2, others solve the problems at once (see for instance section 3.2.3).

The load balancing problem only occurs in parallel environments. Solving this part sequentially might be suitable for small problems where only little data needs to be moved between the parallel machine and the host. For larger problems this bottleneck can become the crucial time factor and above all the host might not have enough memory to store the whole mesh. Therefore only parallel algorithms are considered in chapter 3.

2.5 Parallelism

Several problems have to be solved that only occur in parallel environments. Apart from data locality, which limits knowledge about the mesh and makes it more difficult to run a load balancing or partitioning algorithm in parallel, the concurrent execution of several actions can cause severe problems. In the following sections several important issues are discussed.

2.5.1 Data locality

Load balancing itself is very communication intensive. This is not only due to the basic task of migrating data (load) between the processes, but also an optimisation of the partition cannot be done without the knowledge of non-local data. If, in a message passing environment, every non local piece of data that is to be accessed is to be sent at any time it is requested, the communication overhead is likely to be far too big to put the balancer in the position to save the simulation time by balancing the load. Hence many parallel strategies use an overlap at the borders of each subdomain. This overlap is simply a copy of a set of non-local elements that are located on the other sides of the subdomain borders. The term *halo* is widely used to specify these sets of copies of objects.

Figure 2.20 shows a partition and a possible distribution of halo elements along the borders, where only those elements are copied that share an edge with the local subdomain. It is desirable to have in each halo only those data items that are actually needed. The identification of this data starts by identifying any objects that might be accessed during the optimisation. Of course it might not be possible to only generate copies of those objects that are really accessed afterwards. In fact, the larger the halo, the better the optimisation algorithm

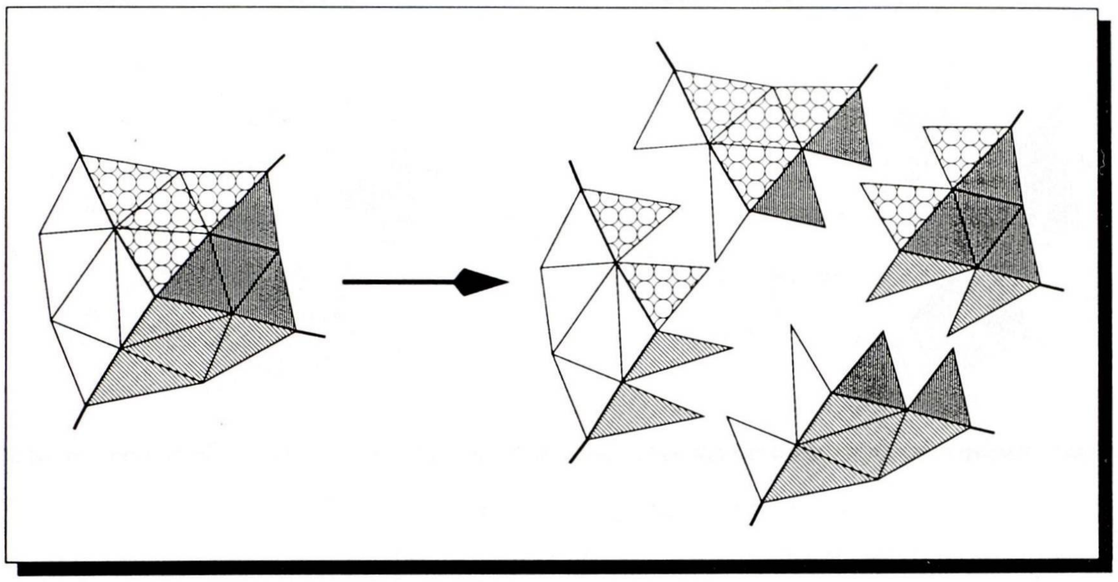


Figure 2.20: Halos

may work, but this probably implies copies of unused data.

For the load balancing algorithm it is usually sufficient to duplicate elements with the information about their area, their borders and their subdomain (where duplicating here is not restricted to one copy per data item, since it might be in the halo of more than one process). Obviously copied data on a different processor always implies the risk of data inconsistency. During load balancing the only data that might change locally or non-locally is the subdomain the element currently is assigned to. There are several strategies to keep this information consistent. Some use synchronous transfer steps followed by one or more synchronous update steps. This approach minimises the memory requirements by avoiding keeping track of the copies made around the processors. An object oriented approach would put each object in charge of updating itself instead of inferring necessary updates from data changes passed from other processes. The responsibility for initiating the update process could either be assigned to the halos or to the original. Both approaches have advantages and disadvantages. A further discussion can be found in section 4.2.4.

The example in figure 2.21 shows a typical scenario where a halo element is not up-to-date. Element e is migrated from S_1 to S_2 and thus these two subdomains should both have the same accurate and consistent data about e and its neighbouring elements. However, unless S_3 gets informed about this migration, it still assumes e is in S_1. This can cause several problems, in particular when element migrations are allowed to be initiated by non-local subdomains. Furthermore

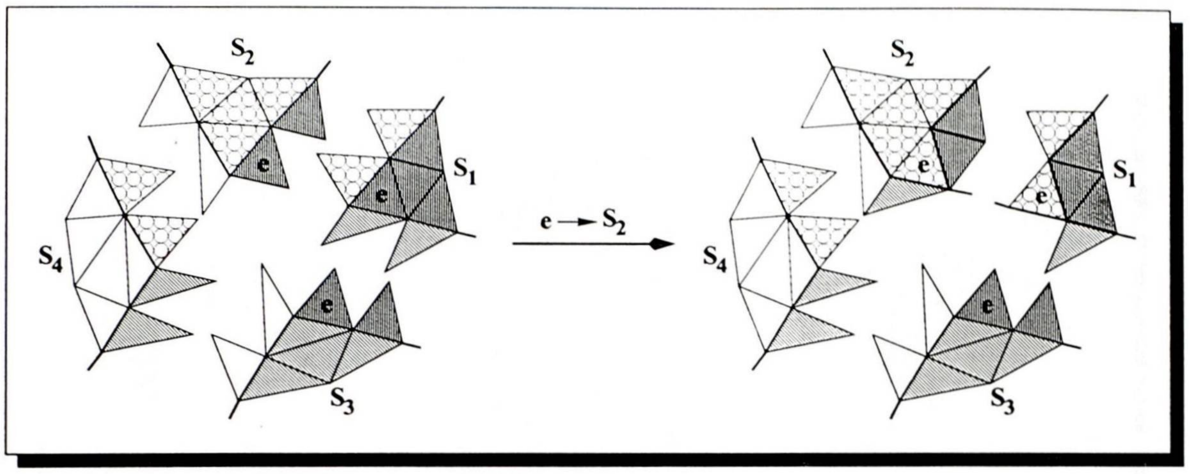


Figure 2.21: Data inconsistency

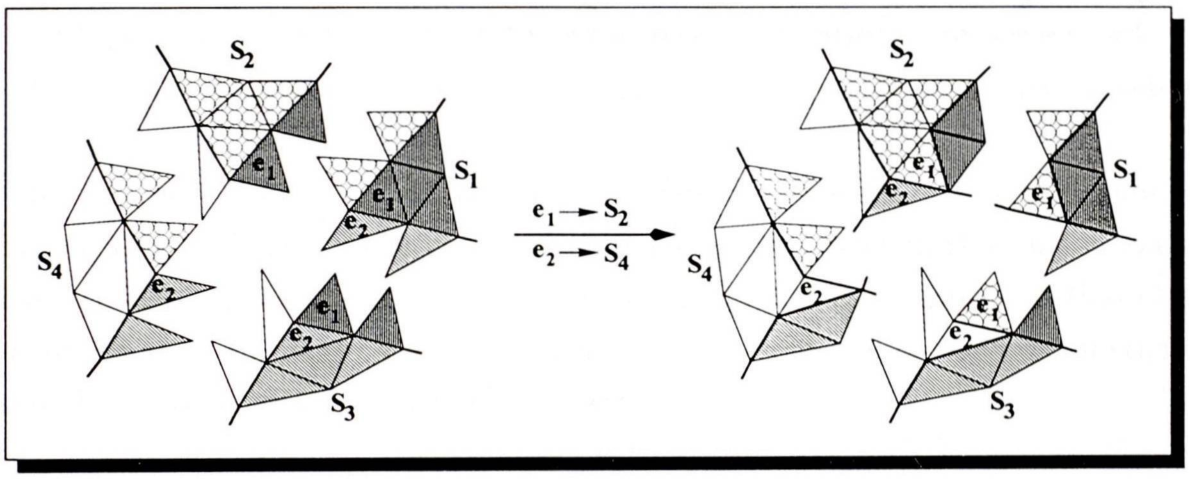


Figure 2.22: 2 communication steps are necessary to ensure data consistency

synchronous communication might run into a dead lock, if such an inconsistent move changes the subdomain graph.

Development of connectivity

In figure 2.21, apart from the data inconsistency a second problem arises. Before the migration of e subdomains S_2 and S_3 were not adjacent. However after the move e builds the border between those two subdomains. If the migration step includes moving adjacent objects as well, S_2 should be aware of that change, whereas S_3 needs to be informed about this. If the migration step is synchronised, this information can be passed at the same time to update the neighbour's data. However, in a more complicated situation the data consistency cannot be ensured in only one communication step (see section 4.2.4). In figure 2.22 the starting partition is the same as in the previous example but this time two elements, e_1 and

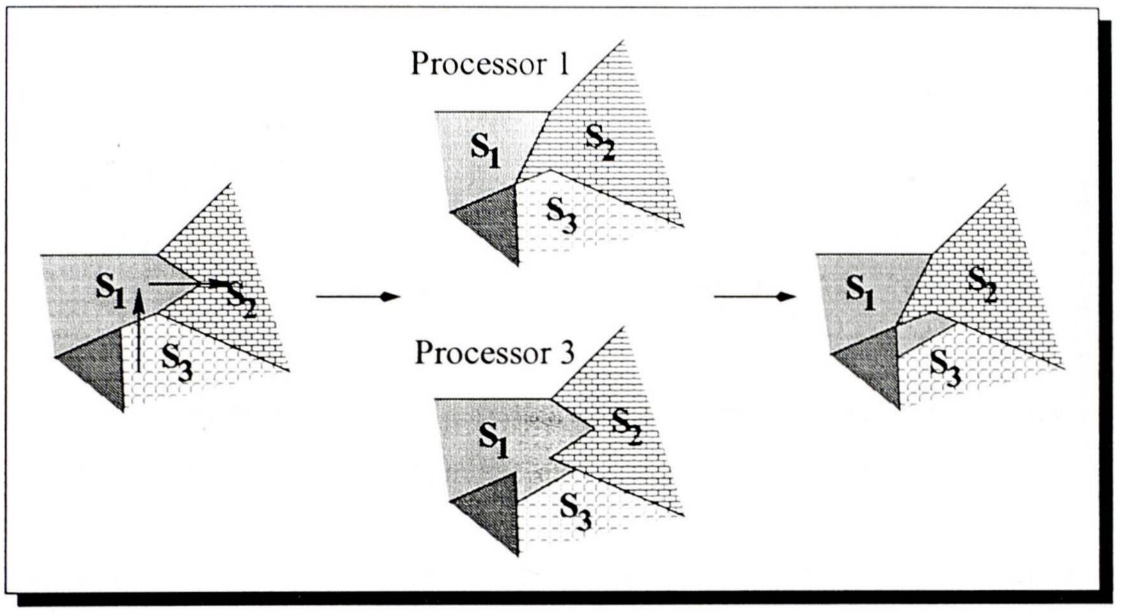


Figure 2.23: Concurrency causes subdomains to receive disconnected subsets

e_2 are migrated to S_2 and S_4, respectively. Since neither S_2 nor S_3 know about the other subdomain's move they will propagate wrong data to their neighbours. This inconsistency can only be resolved by adding another communication step to correct this. However, if the update was not done within the migration step a second communication is necessary anyway.

2.5.2 Loss of connectivity

Concurrency

When flows occur on subdomain borders that are close to each other, the parallel migration of elements can cause subdomains to become disconnected. In figure 2.23 S_1 migrates all elements along its border with S_3 to S_2. Since S_3 at the same time moves elements 'across' that border to S_1, S_1 will receive a set of elements that will not be connected to it due to the moves to S_2. Sequential algorithms would satisfy one flow at a time and know at any time of the execution the current state. Thus S_3 would know before migrating elements to S_1 that these two subdomains are no longer adjacent.

Disappearance of subdomain edges

This specific problem is typical for load balancing strategies that first calculate the flow and then try to satisfy it. Figure 2.24 shows an example where the source partition S_1 has two outgoing flows. If first the horizontal flow to S_2 is satisfied

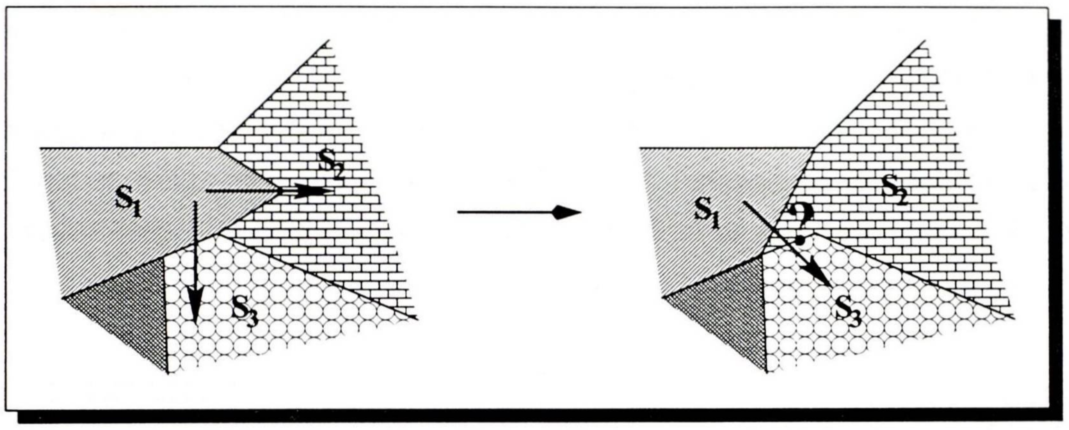


Figure 2.24: Subdomain edges can disappear while satisfying flow

the elements that are on the border with S_3 are migrated to S_2 the subdomain edge between S_1 and S_3 disappears. S_1 cannot decide which elements are the best to be moved to S_3, since none of its elements is adjacent to S_3. Whichever elements are moved to satisfy the flow S_3 will split into two or more disconnected subsets.

2.5.3 Termination detection

Sometimes the limited knowledge of data causes the need for a termination detection strategy. If for example an algorithm runs some calculation on each subdomain until all sub-solutions have some specified precision, each processor can only detect that its own computation has reached that criterion but others might still be working. Usually parallel algorithms depend on regular communications, at least between neighbours. Thus an early termination of one process could easily run into a dead lock or terminate the whole calculation prematurely without having globally reached the specified target precision. A sequential model obviously does not have this problem. Also sequential simulations of parallel executions usually do not need an additional termination detection.

Various strategies have been suggested in literature. Generally two different approaches exist. A trivial attempt would simply perform a global operation like a global AND and the whole problem is solved with only one command. The disadvantage of this approach is that communication between non-adjacent processors is usually much slower than local communication and a global operation may include quite a lot of communication between such processes. On the other hand, strategies that work with only local communications have a lower bound time complexity of $O(D)$, where D is the diameter of the processor graph. Hence the question is whether at least D local communications or one global operation is faster. For this no uniform answer can be given, since it depends heavily on the underlying hardware. Generally modern parallel environments provide very fast global communication facilities. Thus no local strategies will be discussed in more detail here, however a popular algorithm due to Dijkstra can be found in [33].

2.6 Summary

In this chapter an introduction into the fields of partitioning and load balancing was given. In section 2.1 the general partitioning problem was defined and in section 2.2 several criteria which distinguish partitions from one another were discussed. In particular the use of subdomain optimisation was motivated in section 2.2.2 and several definitions of Aspect Ratio which are known from literature were presented and compared. In section 2.2.3 a new formulation of Aspect Ratio was introduced and compared to existing definitions. The observations made in this discussion were backed up by experimentation in section 2.2.5. The meaning and importance of static and dynamic partitioning were explained in sections 2.3 and 2.4 and the chapter closed with a discussion of several problems of the parallel execution of load balancing algorithms (section 2.5).

As key results for this thesis the discussion and tests conclude that firstly the new definition of Aspect Ratio seems to be the most useful formulation (sections 2.2.3 and 2.2.5). Secondly, in contrast to the common criterion of minimising the number of cut edges, subdomain shape optimisation may be preferred for a certain class of solvers since good subdomain Aspect Ratios can significantly improve the convergence of these solvers (section 2.2.5).

Chapter 3

Algorithms and Heuristics

In this chapter several heuristics that are known from literature are introduced. Most of them do not solve exactly the problem of dynamic load balancing and at the same time optimising the subdomain shapes. However, the introduced heuristics give an overview over the actual state of the art in the field of mesh partitioning and dynamic load balancing. Some basic ideas and strategies used in the presented heuristics have been applied in this thesis.

The chapter starts with introducing static graph partitioning heuristics. The most popular partitioning strategy first presented by Kernighan and Lin and the most important heuristics which use coordinate information for finding a suitable decomposition are described. Strategies for dynamic load balancing are presented before the chapter ends with the introduction of three parallel algorithms for calculating flow in a network. The flow heuristics are described in more detail since these strategies are part of this thesis.

3.1 Partitioning

Much research has been done in the field of partitioning and several partitioning tools are available like CHACO by Hendrickson and Leland [47], PARTY by Preis and Diekmann [80, 81], TOP/DOMDEC by Farhat and Simon [37], SCOTCH by Pellegrini [77], METIS by Karypis and Kumar [64] or JOSTLE by Walshaw [98]. CHACO and PARTY provide a number of different partitioning strategies like the recursive spectral bisection [49, 78], the inertial method (see section 3.1.2), other coordinate based methods [28, 36] and greedy methods [25, 91]. Other heuristics like Simulated Annealing [32, 66, 110] or tabu search [94] were also used to solve the partitioning problem. Since this work addresses dynamic load balancing, these static partitioning methods are not discussed in detail. However some ideas which where originally invented for static partitioning but are used for load balancing are briefly introduced in the following.

3.1.1 Kernighan-Lin and related methods

The basic bisection heuristic

In 1970 Kernighan and Lin [65] proposed a partitioning heuristic for optimising cutsize. The algorithm is one of the most common strategies for improving initial partitions because of its robustness and simplicity. The local algorithm is not suitable to solve the dynamic load balancing problem on its own. It assumes a pre-balanced partition and never changes the loads of the individual subdomains during the optimisation. Nevertheless its basic idea can be and is used in many other partitioning and load balancing strategies. In the following the heuristic will be referred to as KL.

Kernighan and Lin used the fact that for every non-optimal bisection two equal sized subsets of vertices (one for each subdomain) exist which, if swapped, lead to the optimal partition. KL aims to find these sets. The idea of KL is to approximate these subsets by performing several loops in which subsets of vertices are exchanged to improve cutsize. These sets are constructed with pairs of vertices swapping subdomains in a straight forward way – see Algorithm 3.1.

Algorithm 3.1 KL_Loop

```
1:  $m = 0$;
2:  WHILE still pairs of unlocked vertices available DO
3:    select vertex pair $p_m = (v_i, v_j)$ with highest gain in cutsize
4:    $g_m$ = gain($i, j$);
5:    lock $v_i$ and $v_j$;
6:    FOR $k$ = all unlocked vertices neighbouring to $v_i$ and $v_j$ DO
7:      update $gain_k$;
8:    END FOR
9:    $m = m + 1$;
10:   $l = \max_j(\sum_{i=0}^{i<j} g_i)$;
11:   FOR $i$=0 to $l$ DO
12:     swap($p_i$);
13:   END FOR
14: END WHILE
```

Here $gain_i$ (line 7) denotes the gain in cutsize if vertex v_i was moved to the other subdomain. It is calculated as the difference in number of its incident edges that connect it to vertices in the target subdomain and those that connect it to vertices in the same subdomain. The function gain(v_i, v_j) in line 4 returns the gain in cutsize if the two vertices v_i and v_j swapped subdomains. If $gain_i$ and $gain_j$ are known gain(v_i, v_j) can be easily computed as $gain_i + gain_j - 2 \cdot c_{ij}$ where c_{ij} denotes the connectivity between v_i and v_j (i.e. 1 or 0).

The loop described above virtually exchanges a series of pairs of vertices in a greedy style. In contrast to a pure greedy algorithm KL does not stop swapping vertices when a move increases the cut. This enables KL to overcome local minima (see section 2.1.2). After virtually swapping all possible pairs only the first l pairs which produce the best improvement of cutsize are actually moved. This loop (lines 2-14) is repeated until no further cutsize improving sets are found.

Improvements by Fiduccia and Mattheyses (FM)

With a time complexity of $O(n \cdot \log n)$ for each virtual move the original KL algorithm is very expensive. Instead of finding the best pair it is much faster to search only for one vertex which improves the cutsize most. Fiduccia and Mattheyses [41] improved the KL heuristic by applying this idea. Instead of pairs of vertices, only one vertex is virtually moved and locked until no more vertices are unlocked. The problem arising from this strategy is to maintain balance since moving only one vertex might destroy it. This can be easily solved by allowing only those moves which do not cause too high an imbalance. The limit can be chosen arbitrarily (the maximum allowed difference in numbers of vertices per subdomain however must be at least one) and is dependent on the requirements on the partition. FM also uses a bucket data structure which allows very fast access to the vertices during updates and fast determination the best possible move (see also section 4.2.5). Fiduccia and Mattheyses even proved a constant access time for those actions, which is a substantial improvement on the complexity of KL. Furthermore this heuristic can be easily used to dynamically balance unequal sized subdomains.

k-Partitioning

Improving partitions of more than two subdomains can be done in several ways. Kernighan and Lin suggest using the same algorithm on any two neighbouring subdomains until no further changes are made. The conventional recursive method (recursively splitting the problem into two equal sized subproblems) is suggested as well as iteratively splitting off one subdomain at a time and optimising the generated border in conventional style.

More KL-variants

Several further variations of KL and FM have been investigated in the past [14, 47, 24, 48, 79, 85, 103, 106]. The Helpful-Set heuristic (HS) [24, 79] for example no longer exchanges pairs of vertices but tries to find pairs of sets for improving cutsize. After moving a set of vertices from one subdomain to another, a set of vertices is searched which restores balance but does not worsen cutsize to less than the value before the move of the first set.

In [85] the maximum allowed imbalance is dynamically decreased during the optimisation. At the start a high imbalance is tolerated. In several steps the imbalance is adjusted to the required value.

Walshaw et al. present in [108] a parallel variant. To keep the number of collisions as low as possible the concept of relative gain was introduced. For selecting the best vertex not only the actual gain in cutsize for a certain target subdomain is accounted for. The average gain in cutsize of the adjacent vertices in the target subdomain is subtracted. This gives an indication how likely these neighbour vertices are to be migrated in the opposite direction and cause a collision. In [104] an interesting approach was presented that includes the radial distance of vertices to the graph centre in the gain function. The graph centre here is not determined as the centre of mass but as the (set of) vertices that are located furthest from the subdomain borders. The radial distance of a vertex then is the the shortest path from the centre to it. This strategy showed to produce very compact subdomains.

3.1.2 Coordinate based partitioning

Very simple and fast heuristics [8, 38, 60, 74] exist, which are not concerned about the connectivity of the graph but about the geometric locations of vertices. Often they produce very poor partitions in terms of cutsize, but the subdomains can be very compact and thus the Aspect Ratios can be very good. They are often used as an initial solution to apply a local heuristic like KL, HS or FM.

Recursive bisection

The Recursive Orthogonal Bisection heuristic (ROB) [8] starts by determining a geometric cut that is parallel to the x- or y-axis. The cut divides the graph into two equal sized subdomains using the coordinate information of the vertices. The resulting subdomains are now recursively divided with cuts orthogonal to the previous cuts and so on until the required number of subdomains is constructed. The advantage of this algorithm is its simplicity, but the resulting subdomains can be disconnected and their AR can be very poor. The Aspect Ratios could be improved by selecting the cut that produces the best AR in each level rather than strictly alternating between x- and y-directions.

One of its variants, the Unbalanced Recursive Bisection (URB) [60] tries to improve the subdomain shapes. The initial cut optimises the AR of the resulting subdomains by specifying a separator, which divides the graph into subdomains with $\frac{N \cdot l}{k}$ and $\frac{N \cdot (k-l)}{k}$ vertices if k subdomains are to be generated from N vertices. The resulting subdomains are recursively divided with the same paradigm. This algorithm is also very fast and simple but additionally produces subdomains with better Aspect Ratios than ROB.

Inertial

The inertial bisection method [111] tries to account for the fact that the graph or mesh may have natural axes of geometric symmetry that are not aligned along the defined axes. To find a better orientation for cutting the mesh the *preferred axis of rotation* or *main principle axis* of the mesh is found first. The cut is made parallel to this axis such that the two subdomains are equal sized. For computing the principle axis the centre of gravity of the graph and the distances from it of all vertices are used. In 2D a 2×2 matrix M is formed with all possible combinations of the accumulated vertex distances in x- and y-directions. The principle axis u is then the eigenvector of the only non-degenerate eigenvalue of M.

To generate more than two subdomains several strategies can be applied. The recursive approach partitions the generated subdomains by applying the same algorithms to the two sub graphs until the required number of subdomains is found. The linear method simply slices the graph in the required number of parallel subdomains such that each subdomain has an equal number of vertices, [111]. A more sophisticated approach [45] tries to generate square like subdomains. With the knowledge of u an orthogonal vector $u^{\perp}$ is known as well. Finding the maximal distances between vertices in directions u and $u^{\perp}$ a minimal rectangle can be defined that completely encloses the graph. To generate k subdomains a factorisation $n \times m$ is determined such that the ratio n/m is closest to the ratio of the two sides of the rectangle. Now the graph is cut in n parallel equal sized slices in the direction according to the specified factorisation ratio. By dividing each of the resulting n parts into m parallel slices in the orthogonal direction the graph is partitioned into $n \cdot m = k$ subdomains. If k cannot be factorised properly one or more subdomains are cut off until the number can be factorised. The advantage of this approach is the usually very even structure of the subdomain graph. It combines the advantages of the recursive and the linear method by computing the partition directly (without recursion), producing good subdomain shapes and being able to cope with an arbitrary number of required subdomains.

The advantage of the inertial strategy is its low cost. It is one of the fastest known partitioning algorithms but for optimising cutsize it can produce very poor results. This is not surprising as the method does not use the connectivity information of the graph. However it can produce good subdomain shapes and can provide good solutions to apply a local optimiser on, either after the partitioning is completed or even during recursion [34].

3.1.3 The multi-level paradigm

The multi-level method was first applied to the partitioning problem by Barnard and Simon [6]. The idea of the multi-level paradigm is to iteratively create a series of reduced sized graphs down to a certain limit. This is done by finding independent subsets of vertex pairs and collapsing each vertex pair to one 'super-vertex'. The subsets are independent in the sense that no two edges in the set are incident to the same vertex. After the contraction the small graph can be partitioned very rapidly and the graph is finally expanded again.

Hendrickson and Leland [48] introduced local optimisation during the expansion phase. After the reduction is complete the graph is iteratively expanded and at each level optimised until the original granularity is restored. This technique has proved to be very powerful in terms of speed and quality of the achieved partitions. Modern partitioning tools therefore incorporate this approach into their heuristics [64, 80, 98]. They differ in the way they find the pairs of vertices to be collapsed, the limit to which they reduce the graph to and the optimisation strategy applied at each level.

3.2 Load balancing

In this section an overview of existing approaches for dynamic load balancing is given. They reflect the current state of the art in the field of dynamic load balancing (see also [46]).

3.2.1 Scratch/remap

One possible way to find a balanced partition is to decompose the unbalanced mesh from scratch without using the information of the current distribution. The drawback of this approach is that the amount of necessary data migration can be arbitrarily high. On the other hand this strategy might find better solutions and for a large imbalance it might even be faster since expensive flow computations are omitted. Biswas and Oliker suggested an intelligent remapping algorithm to minimise the data migration [10]. The remapping process is very fast and this scratch/remap algorithm produces better results than conventional diffusive methods if the imbalance is large. This observation is backed by a similar algorithm provided by Metis [61, 89] (see also section 3.2.3).

3.2.2 Walshaw/Jostle

Jostle [98] is a software package developed at the University of Greenwich by Walshaw *et al.* In section 6.2.9 the results of Jostle are compared to those produced by the heuristic presented in this thesis.

Walshaw *et al.* [107] also split the problem into two subproblems. The flow is calculated with an algorithm first proposed by Hu and Blake [58] which is described in section 3.3.3. It is related to the diffusive methods and reduces the problem to solving a system of linear equations.

An algorithm related to FM selects the vertices that migrate in order to satisfy the flow and the optimisation on the borders is integrated in the balancing [99]. To perform this optimisation in a SPMD environment the borders of the neighbouring subdomains are built with halo vertices and stored on the local processor. The relative gain briefly described in section 3.1.1 prevents the algorithm from producing collisions.

In addition the multi-level method (see section 3.1.3) is applied and extended to give a more global perspective. The reduction of the graph is performed until the number of vertices is equal to the number of subdomains, while ensuring that (if possible) all vertices take part in the matching of each collapsing step. If a parallel vertex matching was performed strictly locally it is possible that no reduction would take place for bad initial partitions. To avoid this situation several vertices might be migrated before being collapsed, to find a matching that involves all vertices. Again a border of halo vertices is used and needs to be updated during the parallel matching. After the reduction is complete the coarsening/optimising step includes a balancing step by introducing a flow on each border that indicates how many vertices have to be migrated in order to restore balance. When migrating only those (super-) vertices can be selected that do not violate this flow constraint [99].

Recently a sequential version of Jostle was enhanced with the ability to optimise Aspect Ratio [102]. The definition of Aspect Ratio developed in this work is applied as the cost function and is used to find the matching when collapsing the graph as well as optimising the partition whilst expanding the graph to its original structure. It has been shown to produce very good results in terms of the resulting Aspect Ratios. In particular in comparison to the version optimising cutsize it shows a significant difference between these two optimisation criteria.

3.2.3 Karypis/Metis

The partitioning and load balancing tool Metis/ParMetis [64, 61] was developed at the University of Minnesota by Karypis & Kumar. It provides several methods for partitioning and load balancing, some of which are tested and compared to the heuristic presented in this thesis (see section 6.3.4).

All provided methods use a multi-level strategy which is combined with a remapping algorithm (see section 3.2.1) or with variations of the diffusive method. Schloegel, Karypis and Kumar suggest a strategy that applies diffusion at all levels of the expanding phase similar to Jostle (see section 3.2.2) [88]. In addition to this *directed* approach they have also tested a very localised algorithm which balances the load based only upon the knowledge of the current load of the local subdomain and its neighbours. Vertices are preferably migrated if the move is scheduled to an underweight subdomain [88]. Tests have shown that this *undirected* heuristic often produces better partitions but the load is much better balanced if the directed algorithm is used. Hence they propose a combination of these two methods which preserves the advantages of both approaches. Here diffusion is applied only at the coarsest level.

The so called *Wavefront Diffusion* [89] aims to produce better results for heavily unbalanced scenarios. To prevent subdomains from migrating all their elements Schloegel *et al.* suggest to iteratively allow only those subdomains to move objects if their ratio of outgoing and incoming flow is above a threshold. This guarantees that subdomains that have to migrate many elements receive objects before they start migration. Their tests suggests that this strategy can not only made using the gain in cutsize, but also the distances to the source and target subdomain are included (as previously suggested by Chrisoides, Houstis and Rice in [18]). Some of these ideas have been applied and improved on in the work presented in chapter 4.

3.2.6 Hierarchical meshes

Some refinement strategies keep track of the mesh hierarchy which arises from the adaptivity. If the evolution of the fine mesh is known the load balancing needed due to a possible refinement can be accelerated by applying the repartitioning procedure on the parent mesh or even on a mesh more than one hierarchical level above the fine mesh, which can be significantly smaller. For this the parent elements are simply assigned weights which are equivalent to the number of child elements. The resulting partition can then be easily mapped to the fine mesh.

Touheed and Jimack suggest an approach which recursively generates more groups of subdomains which are intelligently balanced [93]. The groups are split up into receiver and sender groups which then exchange load until the original number of subdomains is restored. During this balancing the actual topology of the subdomains is taken account of and therefore unfavourable element migration is omitted.

The research by Biswas and Oliker mainly focused on the remapping algorithm mentioned in section 3.2.1. They do not propose a new partitioning strategy but use existing partitioners to apply their remapping algorithm. In their framework for adaptive numerical computations PLUM [76] the possibility is given to precisely predict the refinement which makes it possible to apply the balancing step before the mesh is actually refined.

Even though hierarchical load balancing cannot guarantee perfect balance, tests on both approaches show highly balanced partitions.

3.3 Flow computation

The algorithms described in this section aim to calculate the *flow* of elements in the subdomain graph to achieve a global balance of work load. To apply the second part of the load balancing problem, i.e. the actual migration step, the algorithm calculating the *flow* needs to specify the following for each subdomain:

1. The subdomains to which elements have to be moved
2. The exact number of elements that are to be sent there

With this information the actual choice of elements to be moved can be made by each subdomain individually. The following sections describe three parallel approaches to solve the flow problem which do not necessarily rely on global communication.

3.3.1 Diffusive method

The diffusive method was named after nature's drive to distribute things evenly. Molecules in a liquid solution for instance will distribute themselves equally in it. A similar observation can be made when heating a piece of metal at one side. The heat will spread over the whole piece even if only a small part is heated.

This idea of diffusion was applied to the network flow problem where the load simply diffuses through the graph as given in Algorithm 3.2. Here S_i denotes subdomain i, $Load_i$ the current load of S_i and $Flow_{ij}$ the number of elements that are to be moved from S_i to S_j. Essentially, in each round, each processor compares its current load with each of its neighbours. If the two subdomains have unequal load the flow between these two processes will be altered depending on the size of this imbalance and a factor $0 < \alpha < 1$.

Algorithm 3.2 Diffusive_Flow

```
FOR ALL (subdomains $S_i$) DO IN PARALLEL
  WHILE not converged DO
    $Load_i^{new} = Load_i$;
    FOR ALL neighbouring subdomains $S_j$ DO
      $x = \alpha \cdot (Load_i - Load_j)$;
      $Flow_{ij} = Flow_{ij} + x$;
      $Load_i^{new} = Load_i^{new} - x$;
    END FOR
    $Load_i = Load_i^{new}$;
  END WHILE
END FOR
```

Thus after loop $t+1$ the load of subdomain p is

$$Load_p^{t+1} = Load_p^t + \sum_{q=0}^{q<|N(S_p)|} \alpha \cdot (Load_q^t - Load_p^t), \quad S_q \in N(S_p),$$

where $N(S_p)$ denotes the set of subdomains neighbouring to S_p. The loop is repeated until all processors detect the load to be balanced. At the end of Algorithm 3.2 the amount of elements that S_i has to send to S_j is stored in $Flow_{ij}$.

Processors detect the load to be balanced when the load difference between any two neighbouring subdomains is lower than a specific threshold τ. The lower the τ the better the balance that will be achieved. To guarantee the algorithm will converge a threshold $\tau > 0$ has to be chosen. This implies that this strategy cannot guarantee global balance. Theoretically the maximal possible load imbalance is $\tau \cdot diameter(SG)$, in practice however it is usually much lower. To force a flow that is almost perfectly balanced τ simply has to be set to $\frac{1}{diameter(SG)}$ but a value $0.25 < \tau \leq 1$ is accurate enough for most cases, since a maximal possible imbalance of $diameter(SG)$ (which is usually only a fraction of the number of subdomains) is smaller than 1% if each subdomain has more than $100 \cdot diameter(SG)$ elements. Considering state-of-the-art simulations with 100000 to 10000000 elements on machines with at most 1024 processors the maximal possible imbalance then easily gets smaller than 0.0001%, which will probably not considerably influence the run time.

One major criterion for the convergence of the diffusive method is the suitable choice of α. If $\alpha > \frac{1}{degree(S)}$ subdomain S might have to send more elements to neighbouring subdomains than it actually holds. Thus α must be set to

$$0 < \alpha < \frac{1}{degree(S)}.$$

When the program is run sequentially the termination detection is trivial, whereas on a parallel machine this is much more complicated. When a subdomain detects balance in round i this does not imply that this will be the same in round $i + 1$. This makes it necessary to add an extra termination detection scheme (see section 2.5.3). For a sequential simulation of a parallel algorithm no such termination detection is necessary.

Some variants of this algorithm are known. A schedule that improves convergence is proposed in [44] and uses information of the former round to calculate the new flow increment between two subdomains. However, this scheme requires the calculation of the eigenvalues of the subdomain graph before the diffusion commences. Horton [55] describes a multi-level approach which also improves convergence and additionally guarantees global balance. Unfortunately it requires some global information about the subdomain graph. In the work presented in this thesis an enhancement was established, which aims to adapt the resulting flow to the geometry of the subdomain graph by assigning a weight to each edge of the subdomain graph. This makes it possible to avoid flows on 'bad' borders, for example those which share only one element. For a more detailed description of this method see section 4.3.3.

Diffusion matrix

The diffusion process can also be presented as an method iterating over the time t. It uses the load vector l, where l_i is the current load of subdomain S_i and a *diffusion matrix* D:

$$l^{t+1} = D \cdot l^t$$

If the above presented approach is followed the diffusion matrix is filled with $D_{ij} = \alpha$ if subdomains S_i and S_j share a border and with 0 otherwise [44]. Related methods [44, 58, 70] use this formulation of diffusion for their analysis.

The convergence of the diffusive algorithm is influenced by the choice of α. Depending on the structure of the subdomain graph different values of α result in the best convergence, which are known for certain structured examples [19, 29, 73]. In section 4.3.3 a new approach is presented which replaces the uniform α with individual α_{ij} not to increase convergence but to direct the flow on favourable borders.

3.3.2 Generalised Dimension Exchange (GDE)

An approach that is very similar to the diffusive method was first proposed in [112] and is a generalisation of an algorithm designed for the hypercube [19]. In the diffusive method each subdomain exchanges load with all its neighbours before its load is updated. The GDE method updates its load immediately after communicating with each neighbour. More precisely the subdomain edges incident to one subdomain are coloured with a minimal number of colours such that any two edges incident to the same subdomain have different colours. The inner loop now iterates over the colours of the subdomain edges and the algorithm stops when the load is balanced. A sequence of $degree(S_p)$ steps is called a *sweep* and is the counterpart to one round of the diffusive method. The load of a subdomain p after one sweep of the GDE method is

$$Load_p^{t+1} = Load_p^t + \sum_{q=0}^{q<|N(S_p)|} \alpha \cdot (Load_q^{t-q\cdot\Delta} - Load_p^{t-q\cdot\Delta}),$$

where all $S_q \in N(S_p)$, $\Delta = \frac{1}{degree(S_p)}$ and again $N(S_p)$ denotes the set of subdomains adjacent to S_p. The difference between GDE and the diffusive method is very small, but GDE usually converges faster than its counterpart.

3.3.3 Hu and Blake's algorithm

Hu and Blake [58] solve the load balancing problem by solving a linear system $Lx = b$, where L is a Laplacian matrix. Each diagonal entry of L L_{ii} is the degree of subdomain i in the subdomain graph and non-diagonal entries L_{ij} are -1 if subdomains S_i and S_j are adjacent and 0 otherwise. The vector b contains the differences between the load of the subdomains and the average load. After solving the system with the conjugate gradient method vector x contains the diffusion solution. The load that has to be transferred from subdomain S_i to subdomain S_j is then $x_i - x_j$.

This approach has the advantage that it converges rapidly and yet computes a perfectly balancing flow. Hu and Blake prove that their algorithm is closely related to the diffusion equation (sometimes known as the heat-conduction equation) and that the amount of transferred load is minimal. Later this has been proven more generally for diffusion-related algorithms [22, 57].

Chapter 4

The load balancing algorithm

4.1 Introduction

The algorithm presented in this chapter is motivated by the need for a load balancing strategy that produces and maintains subdomains with good Aspect Ratios. The conventional optimisation criterion (cutsize) often fails to meet this goal.

The coarse structure of the basic algorithm is similar to well known approaches [60, 64, 73, 98]. Ideally the balancing flow is determined (see section 3.3) and afterwards a migration phase satisfies this prescribed flow while optimising the partition (see also section 2.4). Often subdomains lose their adjacency during the migration phase (see section 2.5.2) which makes more than one execution of the flow and migration phases unavoidable. However, repeated execution of those two parts can also improve the partition since the balancing might deteriorate the partition quality in the first loop due to a large prescribed flow. A second execution of the migration phase might then be in the position to do more optimisation than balancing. Thus a loop is applied that iterates over the flow calculation and the migration phases. Adding a preshaping algorithm and a postprocessing leads to algorithm 4.1.

Algorithm 4.1 Balance

```
Preshape();
REPEAT
  Calculate_Flow();
  Migrate();
UNTIL (no elements were migrated);
Postprocessing();
```

The algorithm starts with the preprocessing step that is presented in section 4.4.1. It tries to preshape the worst subdomains by simply cutting off the parts that are too far from the centre. This phase might increase imbalance but since the necessary flow is calculated afterwards it hopefully creates a better solution to start with. An optional postprocessing step is added because some solvers cannot deal with subdomains consisting of disconnected parts. Therefore the final step simply deletes possible disconnected subdomain parts. The produced imbalance is tolerated since an imbalance that is not too large might not affect the run time significantly enough to justify another load balancing step. As this part is only an emergency step even a large imbalance that might be caused by the postprocessing is tolerated, since it will most probably be balanced in the next round of the simulation. This algorithm is described in more detail in section 4.4.2.

In contrast to conventional algorithms for calculating the balancing flow, a new approach is presented which enhances conventional diffusive methods. It aims to direct the flow in such a way that the migration phase is in the position to generate subdomains with good Aspect Ratios. Details can be found in section 4.3.

In the migration phase, the elements that are migrated to satisfy the prescribed flow are chosen with the help of geometric information. Several cost functions are introduced to assess the suitability of elements to be migrated. This algorithm, which is related to KL (see section 3.1.1), is presented first in the next section. The description of the new heuristic starts by introducing the migration phase since results from this section are helpful for the understanding of the remaining parts.

Splitting the main loop into two parts, the flow and the migration phase, often obstructs the optimisation phase by prescribing unfavourable flows, e.g. excessive large flows that are unavoidable due to a large imbalance. Therefore a new strategy is introduced in section 4.5 which interleaves the flow calculation with the migration phase.

4.1.1 Balance

The question of whether a mesh is balanced can be defined in different ways. The most common approaches measure the maximal or minimal load and use the difference from the optimal (average) load. If this difference is small enough the partition is assumed to be balanced. Since the global run time theoretically is dominated by the heaviest subdomain the imbalance of the lightest subdomain can be ignored. In the following the balance will therefore be assessed by the imbalance of the heaviest subdomain even if the smallest subdomain shows a greater difference from the optimal load.

4.2 Migrating elements

The part of the load balancing heuristic that distinguishes this work most from conventional algorithms (that optimise cutsize) is where the elements which actually migrate are determined. This is executed after some flow has been calculated that prescribes how many elements change from one subdomain to another. Unfortunately finding the optimal set of elements for each border is very costly. To minimise the expense of this task, a KL-like strategy (see section 3.1.1) is applied in which each subdomain individually tries to optimise Aspect Ratio while satisfying its outgoing flows.

Essentially a loop is performed in which an element is selected and virtually moved and all necessary data is updated. After that the first k moves that result in the best partition are actually executed. Obviously it is vital to the quality of the algorithm which elements are selected and in which order they are chosen. The choice of the next element is dependent on two criteria. Firstly only those elements are allowed to be (virtually) migrated that keep the outgoing flow and the balance within a certain limit. Secondly from all those elements that fulfill the first criterion the element is finally chosen that promises to help most in optimising Aspect Ratio. A detailed discussion of the two criteria can be found in sections 4.2.1 and 4.2.2.

Common strategies [41, 60, 64, 65, 73, 98] only satisfy/optimise one border at a time. Each subdomain is responsible for a certain number of borders/flows which it optimises one after the other. In that approach elements that are adjacent to more than one (foreign) subdomain might not be migrated to the subdomain that improves the partition most, simply because another border was optimised first and due to the prescribed flow it was forced to move. Furthermore this common strategy can easily cause borders that have not yet been dealt with to disappear, as described in section 2.5.2. To avoid these problems the algorithm introduced here selects the best element according to any of the candidate target subdomains rather than satisfying one flow after another. Thus each subdomain is responsible for all its borders and all flows are satisfied/optimised within the same loop. This direct (and possibly more global) approach promises better results by avoiding unfavourable moves while satisfying a single flow.

It might not be obvious from algorithm 4.2 that not only local elements are virtually moved but also elements that are in the border of a neighbouring subdomain (and therefore in the halo) might be moved to the local subdomain. Without this the algorithm might not be in the position to improve the partition but would only satisfy the outgoing flows. Since elements are only selected if they improve the balance or keep it in a certain limit the algorithm naturally divides into two parts. At the beginning, where the load is not yet balanced only local elements are moved to satisfy the outgoing flows. At a certain point the balance might reach a threshold where moves to the local subdomain can be tolerated. From then on the balancing phase starts. The two functions `enable_optimising()` and `enable_balancing()` (lines 3/16 and 5 in algorithm 4.2) were added to deal with this change. Although theoretically this distinction is not necessary it makes the algorithm more flexible since these two phases have different demands, in particular on the function which assesses the quality of a move (see also sections 4.2.2 and 6.2.5).

Algorithm 4.2 Migrate

```
 1: FOR ALL subdomains $S_i$ DO IN PARALLEL
 2:   IF (balanced()) THEN
 3:     enable_optimising();
 4:   ELSE
 5:     enable_balancing();
 6:   END IF
 7:   $i$=0;
 8:   $best$ = -1;
 9:   $m_i$ = best, valid element/subdomain pair($e_j$,$S_k$);
10:   WHILE $m_i \neq NULL$ DO
11:     lock $e_j$;
12:     move $e_j$ virtually to $S_k$;
13:     IF (best_solution OR balancing) THEN
14:       $best$ = $i$;
15:     END IF
16:     IF ((balancing) AND balanced()) THEN
17:       enable_optimising();
18:     END IF
19:     update all necessary data;
20:     $i$++;
21:     $m_i$ = best, movable element/subdomain pair($e_j$,$S_k$);
22:   END WHILE
23:   execute($m_0,\ldots,m_{best}$);
24: END FOR
```

Moves that were (virtually) made to satisfy flows might obstruct the optimisation phase since these elements are locked and thus are not allowed to be selected again. Often those elements could substantially improve the partition if they were migrated back. To allow this all elements that were virtually migrated in the balancing phase are unlocked within `enable_optimising()`. Now the algorithm can undo unfavourable moves that were forced to balance the flows and replace them with ones that are now available but improve Aspect Ratio.

Assuming the halo was build before algorithm 4.2 is executed no communication is done until the actual migration takes place (`execute(`$m_0, \ldots, m_{best}$`)`, line 22). In other words, each processor determines its own (new) mapping of the elements which it actually holds in its local memory. Obviously this might produce mappings that are inconsistent with the solutions on other processes (see also section 2.5.1 for more concurrency problems). How this is resolved is discussed in section 4.2.4.

4.2.1 Balance constraints

Only those elements which do not increase the balance above a threshold are possible candidates for being virtually migrated. Two objectives compete with each other when setting the balance constraints. A large tolerance in balance might help the algorithm to produce better results by offering escape from local minimum traps, but this clearly contradicts the aim of a load balancing strategy.

As described in section 4.1.1 the work is said to be balanced when no subdomain has more than a certain number of elements. Even if a subdomain can guarantee this locally, it cannot ensure this for its neighbours. Migrating elements to adjacent subdomains can cause its neighbours to receive more elements than they are allowed. If each process migrates the exact amount of elements to each of its neighbours as prescribed by the flow this problem might not occur but the drawback is that it restricts the possibilities of the optimisation strategy.

Therefore during the balancing phase only those elements are selected that better the balance by decreasing outgoing flows. When (local) balance is finally achieved and the optimisation starts, additionally the algorithm can select those that do not produce an imbalance greater than twice the allowed tolerance. At the same time they are not allowed to increase the flow above a flow threshold. A value smaller than 2 for this flow threshold restricts the algorithm too much and values greater than 4 seem not to significantly improve its behaviour. Of course only those states can be chosen as the best solution where local balance is ensured (line 13 in algorithm 4.2). Here local balance means that the current balance is within the allowed tolerance and that all outgoing flows are satisfied.

4.2.2 Rating elements

Choosing the best element requires some kind of rating of the element moves by assigning qualities to the elements. More precisely a move which can be identified by an element/target subdomain pair rather than by an element is assigned a quality. This *quality* specifies in some way how profitable the move of that element to the specified subdomain is. KL uses similar values called *gain* [65]. The term *gain* is applicable for optimising cutsize, since in this particular case the *quality* is identical to the gain in global cost (the cutsize of the partition). For optimising Aspect Ratio other methods for rating elements had to be developed. Usually they do not reflect an actual gain in some cost function, e.g. Aspect Ratio, and therefore the term *quality* is preferred to describe these values rather than gain. However they rate the elements of a subdomain, a high value does not necessarily imply an improvement in the Aspect Ratio of the partition. This will become clearer in the following sections.

If the mesh is decomposed into more than two subdomains, elements can be neighbours to more than one subdomain. It is obvious, that moving the element to one of these subdomains will most probably produce a different Aspect Ratio than moving it to another. The quality of an element therefore is dependent on the target subdomain. The elements used in this thesis are triangles and since elements are only allowed to migrate to neighbouring subdomains an element can be part of at most three different moves. Even if the elements were polygons with more than three edges it is very unlikely that an element has more than 3 neighbouring subdomains which change the Aspect Ratio similarly.

Four basic assessment methods are suggested in the following and additionally variations and combinations of these functions are introduced.

Gain in global cost Γ_{AR}

Similar to the gain in the number of cut edges for cutsize optimisation a simple approach could use the gain in Aspect Ratio of the partition to rate elements. Recall from section 2.2.5 that the global Aspect Ratio is defined as the average Aspect Ratio of the subdomains in the mesh. Using simply the sum of subdomain Aspect Ratios instead changes the value only by a constant factor but simplifies the computation immensely. This is particularly important for the rating function since it is used very frequently. Therefore the sum is used for this function rather than the average value.

The gain in Aspect Ratio then is the difference in the Aspect Ratio (sums) before and after the move. If an element e moved from subdomain S_s to S_t, only the Aspect Ratio of the participating subdomains S_s and S_t can change. Thus the formula needs to consider only the values for S_s and S_t:

$$\begin{aligned}\Gamma_{AR} &= \textstyle\sum_j AR_j^{old} - \sum_j AR_j^{new} \\ &= \left(AR_s^{old} + AR_t^{old}\right) - \left(AR_s^{new} + AR_t^{new}\right) \\ &= \left(\tfrac{B_s^2}{4\cdot\pi\cdot A_s} + \tfrac{B_t^2}{4\cdot\pi\cdot A_t}\right) - \left[\tfrac{(B_s - b_e + 2\cdot b_e^i)^2}{4\cdot\pi(A_s - A_e)} + \tfrac{(B_t + b_e - b_e^t)^2}{4\cdot\pi\cdot(A_t + A_e)}\right]\end{aligned}$$

(see chapter 1 for notations). This equation is quite complicated and cannot be simplified any further, which is one reason why optimising Aspect Ratio is usually slower than optimising cutsize.

Change of global border length Γ_{border}

A similar but much easier variant is to simply calculate the gain in global border length (see section 2.2.4 for this definition). If an element e moves from subdomain S_s to S_t, only those edges incident to e that originally were inner edges and which are on the border of S_s and S_t can change the global border length. The latter ones will decrease cost, since they will become inner edges, whereas inner edges change to border edges and hence increase the border length.

The most common scenarios that can occur for triangulated meshes are shown in figure 4.1. It is assumed that only border elements are allowed to be migrated. If the element e in the up-most example moved from S_s to S_t its two border edges become inner edges and the inner edge becomes a border edge. Thus the global border length is shortened. The second example shows a similar state, but here two inner edges are converted to border edges and hence the global border length increases. The last scenario includes an outer edge which, of course, will never change to an inner or border edge. It can be easily seen that it has no impact on the global border length, since it simply changes the 'owner'. Note that the same applies to border edges with a third subdomain. Situations such as elements with two outer edges or three border edges are not shown in figure 4.1. However their behaviour can be inferred from the above discussion (actually they are much simpler because their contributing edges all either increase or decrease the value).

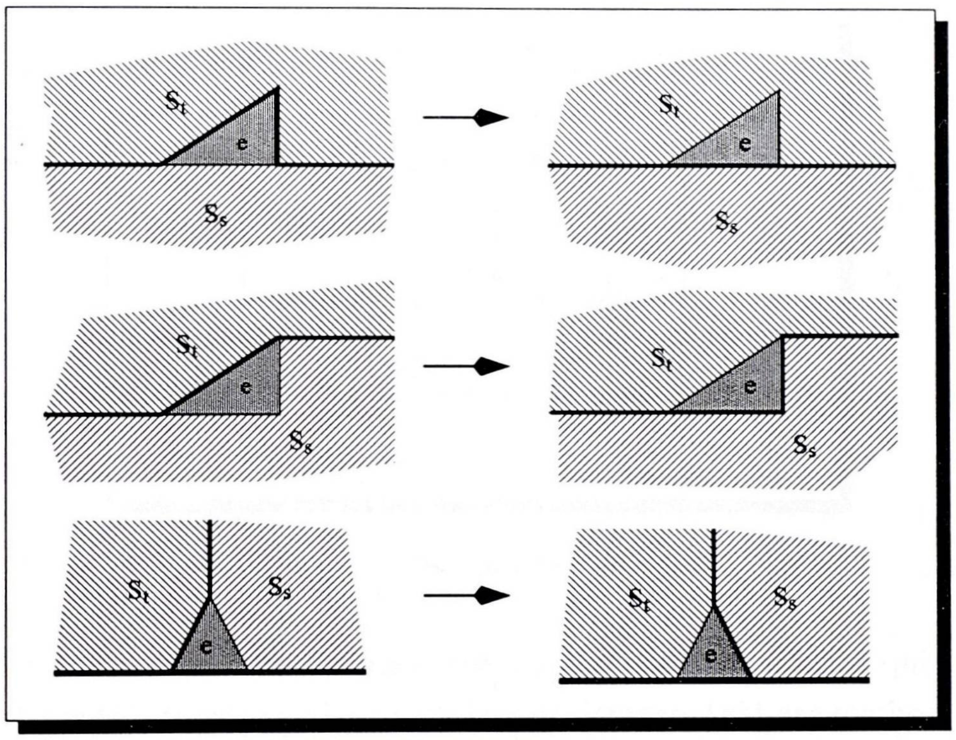


Figure 4.1: Change in border length

The global gain in border length can be easily computed as

$$\Gamma_{border} = b_e^t - b_e^i$$

which is much simpler than Γ_{AR}. An element with three inner edges will be assigned a negative quality and an element with two or three border edges will have a very high quality. This is very convenient since elements in the interior of a subdomain should not change subdomains, whereas those inside a foreign subdomain should be migrated.

This function is also closely related to the gain in cutsize Γ_{cut}. If all edges have the same length, Γ_{border} is identical to Γ_{cut} on the element graph. Thus Γ_{border} can be interpreted as the gain in cutsize (of the element graph) where the edges are assigned weights and any conventional cutsize optimising partitioner or load balancer that is capable of edge weighted graphs can optimise global border length.

Distance to the centre of its subdomain Γ_{dist}

This rating simply sorts by the distance

$$\Gamma_{dist} = dist(e, C_s)$$

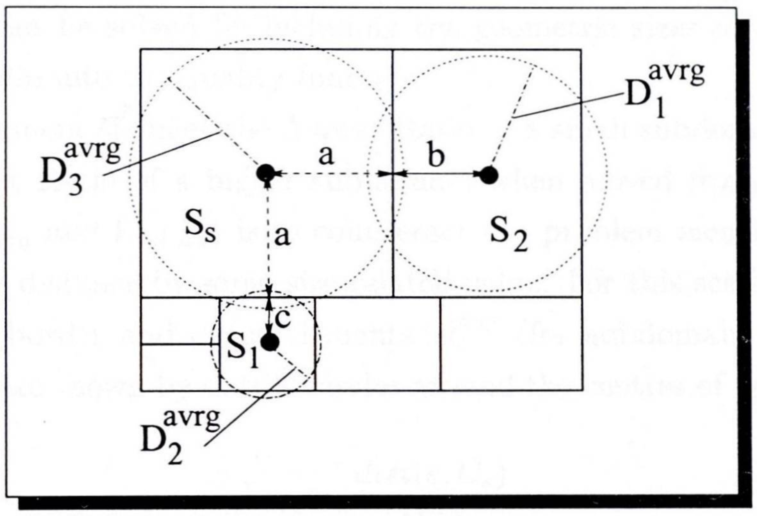


Figure 4.2: Γ_{rel_dist} needs scaling

of an element e to the centre of its current subdomain S_s. This is quite sensible since applying this function tries to send away elements that are furthest from the centre of the subdomain and thus eventually brings the shape closer to a circle.

Relative distance Γ_{rel_dist}

One main weakness of Γ_{dist} is its independence of the target subdomain. In other words Γ_{dist} assigns the same quality to a move of an element for any target subdomain. However, target subdomains should be preferred that might profit when receiving the element. If the centre of a target subdomain is close to the element it should be preferred to those that are further from it. To punish moves to subdomains that are further from the element than others, the distance to the centre of the target subdomain can be subtracted from Γ_{dist}. This leads to

$$\Gamma_{rel\_dist} = dist(e, C_s) - dist(e, C_t).$$

Scaled distances Γ_{avg} and Γ_{rel_avg}

The geometric sizes of neighbouring subdomains can differ heavily. Thus, Γ_{rel_dist} can rate elements improperly as the example in figure 4.2 shows, where S_s is not able to differentiate suitably. Moves scheduled from subdomain S_s to subdomain S_1 will have better qualities than those for S_2, since the qualities are $\Gamma^e_{rel\_dist} = a - c$ or $\Gamma^e_{rel\_dist} = a - b$ respectively and $c < b$. However, migrating elements from S_s to S_2 could improve the AR of S_2 and only slightly worsen the shape of S_s. In contrast, moving elements to S_1 destroys the Aspect Ratios of both subdomains.

This problem can be solved by including the geometric sizes of the target and source subdomain into the quality function.

A (fixed) element changes the Aspect Ratio of a small subdomain much more than the Aspect Ratio of a bigger subdomain when moved from or to it. The intuition for Γ_{avg} and Γ_{rel_avg} is to counteract the problem mentioned above by normalising the distance by some size-related value. For this scaling the average distance of all border and outer elements D_i^{avg} (for subdomain i) is used. In figure 4.2 they are shown by dotted circles around the centres of the subdomains. By defining

$$\Gamma_{avg} = \frac{dist(e, C_s)}{D_s^{avg}}$$

and

$$\Gamma_{rel\_avg} = \frac{dist(e, C_s)}{D_s^{avg}} - \frac{dist(e, C_t)}{D_t^{avg}}$$

for element e being moved from subdomain S_s to S_t the geometric sizes of subdomains scale the values. In figure 4.2 the ratio between average distance for subdomain S_1 and possible elements e on a border with S_1 is smaller than the one for S_2: $\frac{dist(e,C_2)}{D_2^{avg}} < \frac{dist(e,C_1)}{D_1^{avg}}$. Hence Γ_{rel_avg} would assign a higher quality to elements scheduled for S_2 than to those for S_1. Note that Γ_{avg} will behave exactly as Γ_{dist} within one subdomain since it is only multiplied by the constant factor $\frac{1}{D_i^{avg}}$. However, these functions are different when directing the flow which is described in detail in section 4.3.

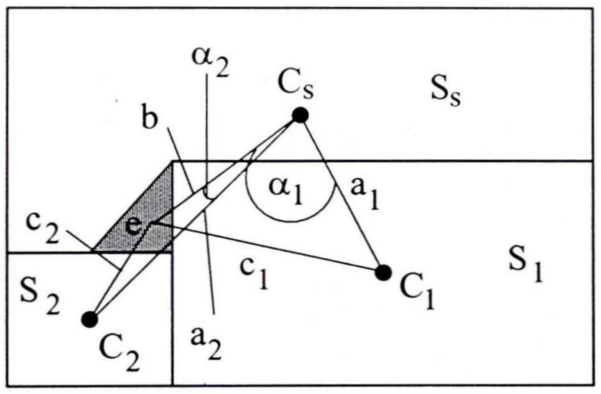
Direction of the move Γ_{angle}

The intuition for this function is to direct elements to regions of subdomains that are close to the target subdomain centre. If an element e is positioned far from the line connecting the source and the target subdomain centres, C_s and C_t, the move of that element runs the risk of producing degenerated shapes. Thus Γ_{angle} uses the angle α between the vectors $\overrightarrow{C_s, e}$ and $\overrightarrow{C_s, C_t}$ to rate element moves.

Figure 4.3 shows an example where two angles α_1 and α_2 for two possible moves of element e in different directions are given. Intuitively the move from subdomain S_s to S_2 with angle α_2 should be rated better than the move to S_1. If the angle α is in the range $-90° < \alpha < 90°$ it is intuitively acceptable whereas the range $90° < \alpha < 270°$ occurs for moves in the wrong direction. In the example given above $\alpha_1 \approx 90°$ and thus should get a lower quality than α_2.

Fortunately

$$\Gamma_{angle} = \cos \alpha_i$$



$$\begin{aligned} a_i &:= dist(C_s, C_i) \\ b &:= dist(C_s, e) \\ c_i &:= dist(C_i, e) \end{aligned}$$

Figure 4.3: Γ_{angle} rates the angles α_i

returns a very good assessment of α and can be calculated very easily. Let S_s be the source subdomain, S_i a target subdomain, e the element to be moved, a_i the line connecting the centre of subdomain S_s and the centre of S_i and let c_i be the line connecting the element e with the centre of a target subdomain S_i (see figure 4.3), then a_i, b and c_i build a triangle and

$$\cos \alpha_i = \frac{a_i^2 + b^2 - c_i^2}{2 \cdot a_i \cdot b}.$$

Angles close to 0° will result in values close to 1 and those which would move an element in the opposite direction of $\overrightarrow{S_s, S_i}$ will be punished with negative values. The non-linearity of the cosine function is very convenient since the range of 'good' angles is widened. A linear function would devalue moves that are still acceptable too quickly.

Due to the lack of adjacency information this method produces poor results if applied on its own. However, as discussed below it can substantially improve the behaviour of other functions.

Comparison

The quality functions introduced above differ enormously in their capabilities and limitations. Certain aspects can be observed and are explained in the following.

The functions Γ_{AR} and Γ_{border} show a similar behaviour. The difference is that Γ_{border} does not consider the areas and thus the Aspect Ratios of the elements and the subdomains have no influence on Γ_{border}. Therefore Γ_{AR} does differentiate better between different subdomains. However, experimental results using both functions as an optimising function show that both methods perform

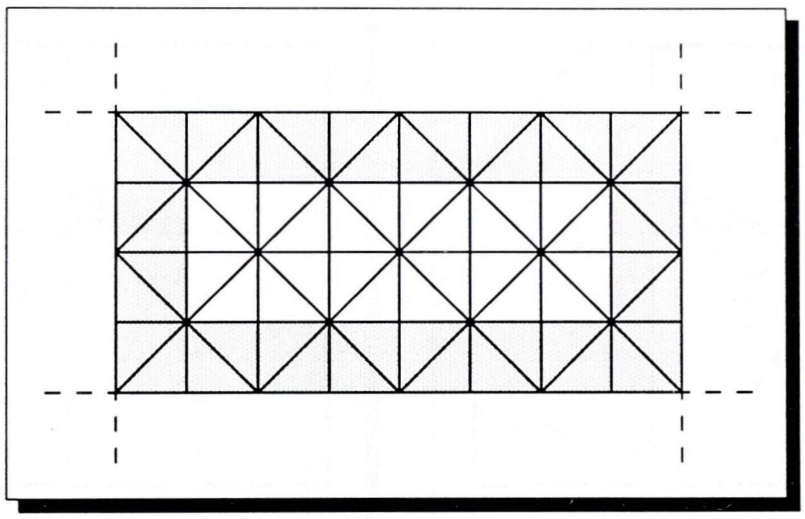


Figure 4.4: A scenario with a local minimum trap

almost equally well, considering only those parameter settings that produce good results. A more detailed discussion of this is given in section 6.2.5 and see also [101].

Γ_{AR} and Γ_{border} usually find the best move if it is cost improving. But a local minimum often cannot be resolved properly. The example in figure 4.4 shows a subdomain with a shape of a rectangle composed of identical triangles. Whichever border element (shaded) is moved it produces exactly the same change in Aspect Ratio, provided the neighbouring subdomains are all of equal geometric size. Thus Γ_{AR} might make a bad choice. Γ_{border} also rates all elements equally, even if the neighbouring subdomains are of unequal size. However, cutting the rectangle on a far side would help to produce a square like subdomain which is often the best shape for a subdomain (see section 2.2.3). Hence an element on a far side should be preferred to one near to the centre of the subdomain. Although Γ_{dist} often returns very poor results and is relatively simple it would, in this example, not run into this trap. Γ_{dist} might have problems in other local minima situations, where two or more elements are located the same distance from the subdomain centre. However, for the long term shape development the outcome would be very similar for each of the candidate moves. No matter which element is chosen first, one of those candidates that had the same quality will be chosen next. No other move will get a better quality than those candidates unless all or at least most of them have been migrated. In contrast a bad decision made with Γ_{AR} or Γ_{border} might, in the long term, even cut the rectangle into disconnected parts. The distance functions are totally independent from the local cost development but follow a strict global optimisation strategy which is intuitively somewhat related

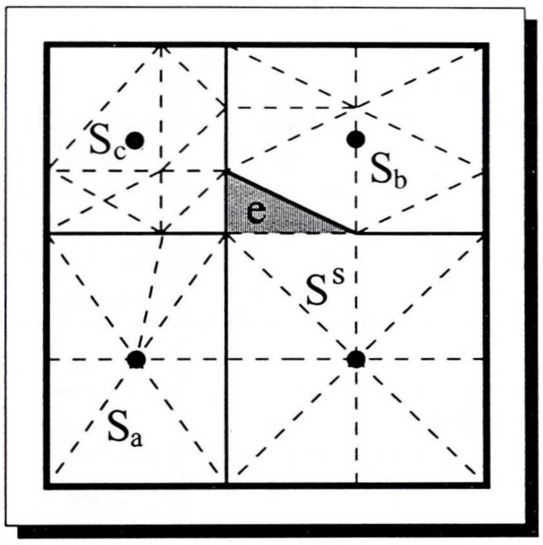


Figure 4.5: Γ_{dist} and Γ_{angle} are not satisfactory

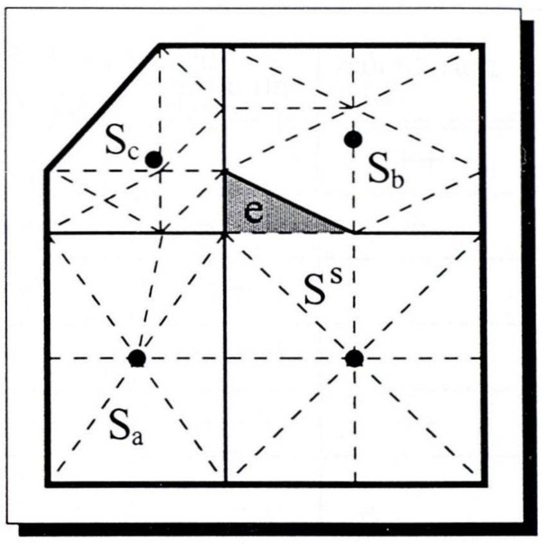


Figure 4.6: Γ_{rel_dist}, Γ_{dist} and Γ_{angle} are not satisfactory

to the global costs, which Γ_{AR} and Γ_{border} aim to optimise. Γ_{angle} does not have a global strategy or an implied impact on the local cost change. It simply rates if the move of e from S_s to S_t is suitable.

If an element is adjacent to more than one target subdomain, the cost function should distinguish between them when assigning the qualities. Figure 4.5 gives an example. If e moved to S_c, the global Aspect Ratio increases, whereas if moved to S_b, it optimises the solution. Γ_{dist} (and Γ_{avg}) will calculate only one value for both cases. This shows the superiority of the relative distance functions Γ_{rel_dist} and Γ_{rel_avg}, which will compute different values for each of the two target subdomains (S_a is not a candidate subdomain). Since it produces different costs for each of the two moves Γ_{AR} and Γ_{border} will also differentiate very well. However Γ_{angle} considers the target subdomain it will make an unsuitable decision in this example. Since e is located between C_s and C_c, Γ_{angle} will prefer S_c. This shows once again the weakness of Γ_{angle}, because the differentiation between different subdomains is only important for those elements that are adjacent to more than two subdomains. Unfortunately however those situations are very often similar to the ones shown in figures 4.5 and 4.6.

The example in figure 4.6 demonstrates that the functions that made the same decision in the latter case ($\Gamma_{rel\_dist}, \Gamma_{rel\_avg}, \Gamma_{AR}$ and Γ_{border}) can behave differently for very similar cases. The functions $\Gamma_{rel\_avg}, \Gamma_{AR}$ and Γ_{border} will still prefer the move to S_b, the better choice. However Γ_{angle} and Γ_{rel_dist} will assign the best quality to the move to S_c. These two examples show that integrating the target

	target partition	local quality	global strategy	local minimum escape	calculation cost
Γ_{dist}	$--$	$-$	$+$	$+$	$++$
Γ_{rel_dist}	$+$	$-$	$++$	$++$	$+$
Γ_{rel_avg}	$++$	$-$	$++$	$++$	$-$
Γ_{AR}	$++$	$++$	$--$	$--$	$--$
Γ_{border}	$++$	$++$	$--$	$--$	$++$
Γ_{angle}	$+$	$--$	$-$	$-$	$+$

Table 4.1: Comparison of the base quality functions

subdomain does not automatically guarantee a well distinguished assessment for all scenarios.

Table 4.1 gives a comparison of the functions according to detailed experience gained during their usage. The presented quality functions are compared in different aspects that were discussed above. The following symbols are used to rate the different aspects and are sorted from bad to excellent: $--$, $-$, $+$ and $++$. The first column shows the ability to differentiate between different target subdomains. Here, Γ_{AR} and Γ_{border} tend to produce the best results and Γ_{dist} fails completely. The second column rates the coherence of the results of the function and the actual improvement of AR. Again Γ_{AR} and Γ_{border} are best whereas Γ_{angle} is not related to any expected change in Aspect Ratio. The distance functions are slightly more coherent to AR but are not directly related to it. A global strategy on the contrary is not followed at all by the two local functions, but Γ_{rel_dist} is a global strategy by minimising the distances of the elements to their subdomain centres. The two local functions have a very poor ability to escape local minima whereas the others may be able to escape due to their global strategy. Furthermore it is very unlikely that several distances of elements to their subdomain centre and additionally the distance to the target subdomains are equal. The last column rates the cost for computing the qualities. Γ_{AR} is the most expensive function but Γ_{border} can be calculated very rapidly and since they behave in a similar fashion, this table suggests that Γ_{AR} is redundant.

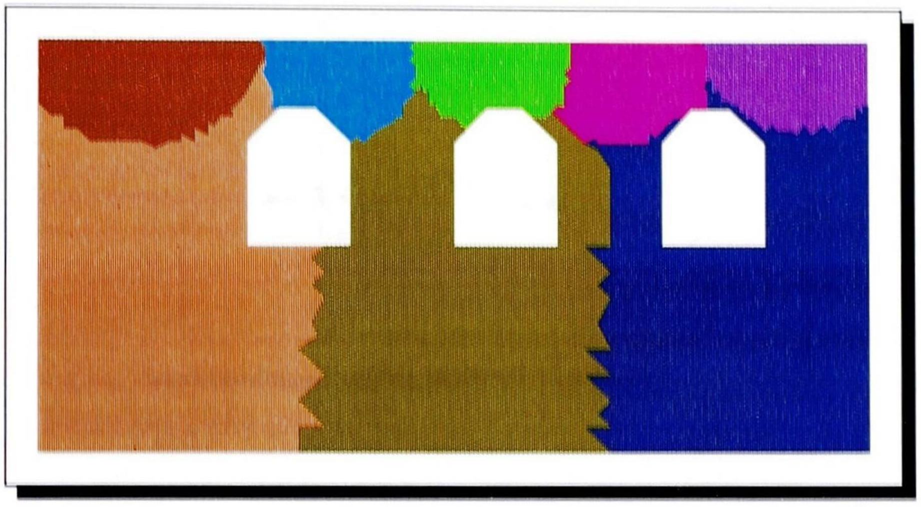


Figure 4.7: Use of Γ_{rel_avg} alone

Combinations

The above observed strengths and limitations of the proposed functions suggest combining them to sum up their strengths and at the same time eliminate their weaknesses. In this section the need for and the usefulness of such aggregate functions will be presented with an expressive example and the use of combinations will be generalised.

Since the size of a fixed element changes the Aspect Ratio of a small subdomain much more than the shape of a larger one, the scaling was introduced to Γ_{rel_avg} to meet this requirement. The scaling helps to weight the influence of the smaller subdomain higher and to decide which border is to be preferred. Although the function is capable of choosing the proper neighbouring subdomain, the effect of a (migrating) element on the shapes of both subdomains becomes somewhat unbalanced. The unsightly side effect is that during repeated load balancing steps using Γ_{rel_avg} the smaller subdomains tend to get optimal shapes, whereas larger subdomains become slightly degenerated. This usually happens on borders of two subdomains of very different sizes. In figure 4.7 this occurs for example on the two subdomains on the left hand, where the smaller (upper) subdomain became almost a semi circle but the larger one is concave at this side. Notice that the borders between pairs of almost equally sized subdomains usually are nearly optimal. They build almost a straight line, perpendicular to the connection of the two subdomain centres. This result was produced by using Γ_{rel_avg} and by only balancing the load in each step without optimising the solutions. This isolated the behaviour of the function much better and explains the jags on the borders.

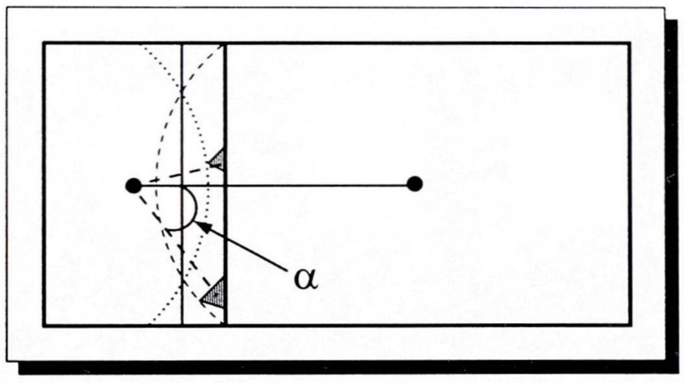


Figure 4.8: Combination of Γ_{rel_avg} and Γ_{angle}

In order to counteract this intuitively unnecessary development, a second function can be added which corrects this misbehaviour without removing the positive effect of scaling. Figure 4.8 shows two subdomains of unequal size. Using Γ_{rel_avg} to move elements from the large to the small subdomain, the resulting border of the smaller subdomain will become the indicated semi-circle (dotted). Adding another function that on its own produces a semi-circle (dashed) on the large subdomain, the resulting function hopefully creates a border indicated as the thinner line between the two subdomains. The function that will do so by definition is Γ_{angle}. Applying the compound function $\Gamma_{angle} + \Gamma_{rel\_avg}$ (with some suitable weighting factor which is discussed below) to the same scenario as in figure 4.8 the resulting partition shown in figure 4.9 looks better: the borders of all neighbouring subdomains of unequal size are now closer to a line than to an arc. Again the many jags occurred because only balancing but no optimisation was done.

Generally any combination of the functions presented in section 4.2.2 are possible. Similarly to [94, 110] the 'base' qualities can be simply weighted by some factor and added:

$$\Gamma_\Sigma = \sum(\Gamma_i(e) \cdot \zeta_i).$$

The problem here is to find suitable ζ_is. This is not a trivial task, since some of the functions are dependent on the geometric size of the mesh (i.e. Γ_{dist}) and others are not (i.e. Γ_{AR}). A solution is to split ζ_i into two parts: a *normalisation* factor κ_i and the actual weighting factor λ_i. The normalisation factors κ_i are determined such that for any element e $-1 \leq \kappa_i \cdot \Gamma_i(e) \leq 1$ holds:

$$\kappa_i = \frac{1}{\max_j(|\Gamma_i(e_j)|)}.$$

The final weighting factor ζ_i then combines the normalisation κ_i with a weight

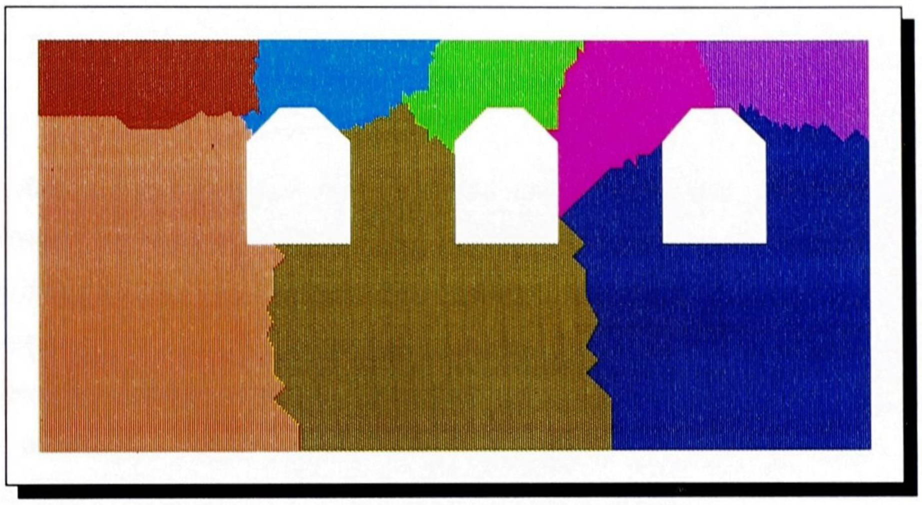


Figure 4.9: Combined quality function

λ_i:

$$\zeta_i = \kappa_i \cdot \lambda_i.$$

4.2.3 Test results

The new heuristic as a whole is a quite complex algorithm which makes it very difficult to assess and compare the different quality functions. Many parameters influence the behaviour and thus the results of the heuristic. A detailed discussion of the test results of the element functions is presented in section 6.2.3. However as an overview, it can be stated that these evaluations do seem to exhibit the intended attributes of the constructed functions. The tests show that the distance functions are most important for achieving partitions with good Aspect Ratios. The relative distance function significantly improves the basic Γ_{dist} and the scaling improves Γ_{rel_dist} further.

Generally Γ_{border} – combined with a suitable weighting factor – positively influences the distance functions and an additional Γ_{angle} often shows even better results. Therefore the good results produced using $\Gamma_{rel\_avg+border+angle}$ indeed suggest a combination of the basic functions. In chapter 6 a detailed description of the tests and their evaluation is presented.

4.2.4 Executing migration

Halo

Each border element which is a candidate for migration needs all its neighbouring elements to compute its qualities. More precisely it needs to be known to which subdomain(s) the element is adjacent and the type of all its edges (border, boundary or inner). Therefore a halo is stored on each processor that even holds enough information to migrate non-local elements. A minimum width of 1 element is needed for each halo and it is possible to enlarge this which possibly gives the algorithm greater freedom for optimisation.

To keep the overhead as small as possible an incremental update of the halos is desirable. Building the halo from scratch after each migration would involve large amounts of unnecessary communication. Hence each processor additionally sends copies of objects neighbouring the migrating elements during the migration phase as well. These adjacent objects are sent even if they are located in the halo. By keeping the migrated objects as copies the halo is enlarged which therefore improves the possibilities of the algorithm. For heavily imbalanced scenarios the halo therefore might grow even larger than the local subdomain. Some intelligent control over the halo (size) would be desirable and could be a topic for further research.

Migration

In section 2.5.1 several problems have been discussed that arise when the optimisation is done in parallel. The main problem is that the data cannot be guaranteed to be consistent without at least two communication steps, [100]. In one of them the new subdomain of the migrated elements has to be transmitted to their neighbours and to any other subdomain holding a copy of these objects.

If the depth of the halo is limited to one element this can be done without too much overhead, since an object is not duplicated on subdomains other than those adjacent to the object. In the case such a restriction is not true it becomes a lot more complicated.

Following an object-oriented approach, each object is responsible for keeping its copies up-to-date. Thus, whenever an object moves, it has to tell all its copies about this change. To be in the position to do this, each original object needs the information of the locations of all its copies. Thus, whenever a subdomains sends a new copy of an non-local object the subdomain actually owning this object must be informed about the new duplicate. This approach was chosen since it conforms best with the idea of object orientation and gives good control over the migrations and locations of copies.

In algorithm 4.2 each subdomain individually optimises all its borders. Thus each border is concurrently dealt with on two processors and the two processes might find different distributions of elements on their border. These different solutions might collide and the inconsistency has to be resolved.

Several approaches have been suggested in the literature to avoid collisions, for example by assigning the responsibility of one border to only one process [100] or using a 2-phase migration where each border is optimised successively in both directions [63]. In this work collisions are not avoided but resolved in order to preserve the full capability of the optimisation algorithm. The main idea here is to assign the responsibility of the migration of elements to the 'owning' subdomain rather than limiting the number of candidates to be moved. Whenever a subdomain has determined the elements that it wishes to migrate the physical migration phase starts which is logically divided into four parts:

1. Request elements
2. Send new copies
3. Migrate objects
4. Update

In the first step each process simply requests all non-local elements that it had tagged to be moved to its own subdomain and (virtually) moves them back to their original subdomain (i.e. into the halo). Each element that was requested will then be tagged for migration on the 'owning' processor if it was not scheduled for migration yet. If it was already tagged for migration and the target subdomain is different to the requesting subdomain the target will be changed only if that improves Aspect Ratio. Experiments proved that in most cases requested elements are actually received. Intuitively this observation is not very surprising since a reasonable optimisation strategy should find similar solutions for a shared border, independent of the processor it is executed on.

In step two, the original objects (elements, vertices and edges) are informed about copies that were eventually distributed by subdomains not owning the object. This could not be done beforehand since in step two requested elements could have been migrated. The actual data migration phase is represented by step three, where each subdomain receives all objects that were scheduled to be migrated to them from their neighbours including those that were successfully requested. Thus all data on borders that resulted in collisions is consistent again. Step two and the actual migration phase (3) are not dependent on each other and thus they can be combined into a single communication.

The last step is needed to resolve the inconsistency problems (previously described in section 2.5.1) caused by the development of new subdomain edges due to the migration of elements. Even elements that were adjacent to more than one (foreign) subdomain and were concurrently assigned to more than two different subdomains will finally be consistently assigned to the best subdomain.

4.2.5 Data management for rapid updates

The run time of the migration phase is highly dependent on the efficiency of the update procedure (line 19 in algorithm 4.2). If an element moves to another subdomain the centres and Aspect Ratios of the target subdomain, the source subdomain and their neighbouring subdomains change. Furthermore, this means that for most of the above introduced functions, the values for all moves of elements in these subdomains change. A sophisticated data management, including a good data structure and a well organised update procedure can save a lot of unnecessary computation.

Data structure

Generally a well designed data structure can influence the efficiency of an algorithm immensely. For algorithm 4.2 it should ensure a fast access to the element candidates. Two main functionalities have to be supported: finding the best element and, after moving it, updating the costs of the other elements. Fiduccia and Mattheyses [41] suggested the bucket list, which ensures constant time for accessing the element with the best cost and for updating its neighbours. Since their partitioning algorithm only optimises the cutsize, the quality/gain of a vertex is always an integer. All vertices with the same value are stored in a linked list or bucket. For all neighbouring subdomains an array of those lists is held on each subdomain. Each entry in an array holds the list of vertices with a certain gain and this ensures a direct access to vertices when their actual gain is known. A pointer to the best entry in the array guarantees constant time for accessing the best vertex move. It furthermore ensures constant time for inserting a vertex if its gain is known. Unfortunately this bucket structure is much harder to apply to the problem field presented here, which deals with non-integer values, [102]. A heap does not support arrays of frequently changing size and no fast partial re-sorting to save redundant work exists (see the next section below). However, one of the simplest data structures, the linked list, suits most the requirements of the updating and accessing processes described above. It can be used to guarantee constant access time for the first entry, very cheap insertion and deletion routines and furthermore it can easily be kept partially sorted.

When optimising cutsize, the move of an element causes the change of costs only of its adjacent elements, but geometric costs like the distance to the subdomain centre might change for all elements of several subdomains. They are independent from the connectivity but can be dependent on the location of the centre of the subdomain and/or its border and area, which usually change when an element leaves it. In the following sections an approach is presented, which ensures constant time for accessing the best element and which needs only little time to update the list after a move.

The moves list

In order to use a data structure that is capable of dealing with non-integer values, all possible entries are stored in one sorted list for each subdomain. Such an entry needs to specify the element, the subdomain it is scheduled for and the quality of that move. Any element can be neighbouring to more than one subdomain, thus an element can occur more than once in that list, where generally also the cost values differ. This approach simplifies the data structure and the access to the best possible move, since for each subdomain only one list exists and its first entry represents the best move. While executing algorithm 4.2 the outgoing flows are satisfied in no fixed order. Each subdomain selects the first entry of its list and if the flow for this move is not yet satisfied, the element is tagged for migration; otherwise the list is traversed until a suitable element move is found. After the move, a subset of entries is updated which guarantees that the move with the highest quality is on the top of the list. Thus in general the list might not be completely sorted.

Minimising the number of updates

Migrating an element can cause other elements to change their quality. For a fast access to the best move these qualities are stored together with the element and the target subdomain. The cost functions described above usually change the qualities of all elements in the source, target and all neighbouring subdomains. An update of all elements after each move would cause a huge overhead. Note that for the geometric functions used in this thesis, generally the cost of an incremental update is similar to a complete re-calculation. With this assumption the redundant overhead can be determined easily.

Let $e_1, e_2, ..., e_n$ be the entries in the list, where an entry represents a move of an element e, the target subdomain and the quality of this move. After move $z-1$ entry e_z will have been updated $z-1$ times without ever being used. Updating only that element that is moved next would save

$$(z-1)+(z-2)+(z-3)\cdots(z-z)=\sum_{i=1}^{z-1} i=\frac{z^2-z}{2}$$

updates for z moves. Obviously it is not possible to know the next element before the update, because determining this element is the reason of the update, but it shows that the simple update process offers a high potential for saving computation time.

Elements that are not on the border of the subdomain (e.g. they have no adjacent element which is in a different subdomain) should not be moved, since this would cause both subdomains to lose connectivity. Therefore those elements are not allowed to migrate. Hence it is legal to only add and update those elements that are on the border. If the mesh is not partitioned into very small subdomains, this immensely reduces the number of elements in the list of possible candidates and thus the number of updates.

To reduce the number of updates to the minimum a threshold can be determined. This threshold should specify which minimal original quality can – due to the update – change enough that it would climb to the top of the list, but no value smaller than this threshold can reach the top. If this threshold is known, it is sufficient to update the list only down to this value, since for the next move it is sufficient to know only the element with the best quality. For calculating this threshold the maximum change in cost Δ_{max} due to an element move has to be found. With the knowledge of this value, the list now is updated from the top down to the last entry above the threshold. Tests on meshes with up to 25000 elements show, that in the average less than 10 elements are updated per selection.

Determining the thresholds

The basic quality functions $\Gamma_{AR}, \Gamma_{border}, \Gamma_{dist}, \Gamma_{rel\_dist}$ and Γ_{angle} change different data. The following paragraphs show how Δ_{max} can be found for the individual functions.

$\mathbf{\Gamma_{border}}$ If the quality function only requires data from objects that are adjacent to the actual element, an update will not affect the qualities of non-neighbouring elements. Γ_{border} is a geometric function that only uses adjacent information. Thus moving an element e changes only the Γ_{border} part of those elements that are neighbouring to e. The border types of some of the edges of e change and thus the global border length might change as well. Since the elements are triangles an element has only three edges. Therefore only three edges can change their type. Hence the maximal change Δ_{max} for Γ_{border} can be determined in constant time. Of course it would be sufficient for this function to only update the neighbours to guarantee the list to be resorted. However in combination with other methods the more general approach – determining and using the threshold – is desirable.

The threshold for Γ_{border} can be computed very easily since only one edge per neighbouring element can change from an inner edge to a border edge. Thus Δ_{max} after moving element e is simply

$$\Delta_{max} = \max\left(length(ed_i)\right),$$

where ed_i is an edge of e.

$\mathbf{\Gamma_{AR}}$ For Γ_{AR} it is more complicated to compute Δ_{max} because the Aspect Ratio includes the areas of all elements in the mesh. Since Γ_{AR} is defined with the help of the Aspect Ratios of the target and source subdomains the values for all elements in the target, source and their neighbouring subdomains might change.

However, Δ_{max} must occur for one e_n adjacent to the migrating element e, since, similarly to Γ_{border}, it is only here, that the border changes. With this information the following equation determines Δ_{max}:

$$\Delta_{max} = \max_i \left(\Gamma_{AR}^{old}(e_i) - \Gamma_{AR}^{new}(e_i)\right), \quad e_i \text{ adjacent to } e$$

Recall from section 4.2.2 that

$$\Gamma_{AR} = (AR_s^{old} + AR_t^{old}) - (AR_s^{new} + AR_t^{new}).$$

Note that Γ_{AR} was computed before the migration of e but Δ_{max} is determined afterwards, thus the values AR_s^{new} and AR_t^{new} used for Γ_{AR} are the same as those of AR_s^{old} and AR_t^{old} used for the computation of Δ_{max}.

Although similarly to Γ_{border} only one single edge of the neighbouring element changes, it is not possible to formulate a simple incremental equation (due to the squares and the change of the subdomain areas). The explicit equation for Δ_{max} is very complicated:

$$\Delta_{max} = \max_n(\left(\frac{B_s - b_e + 2\cdot b_e^i}{4\cdot\sqrt{A_s - A_e}} + \frac{B_t + b_e - b_e^t}{4\cdot\sqrt{A_t + A_e}}\right) - \left(\frac{B_s - b_e + 2\cdot b_e^i - b_{e_n} + 2\cdot b_{e_n}^i}{4\cdot\sqrt{A_s - A_e - A_{e_n}}} + \frac{B_t + b_e - b_e^t + b_{e_n} - b_{e_n}^t}{4\cdot\sqrt{A_t + A_e + A_{e_n}}}\right)).$$

$\Gamma_{\mathbf{dist}}$ For the distance functions it is now the moves of the centres of the two participating subdomains rather than their areas that cause the same problems as for Γ_{AR}. The distance of elements to the centre of the source and target subdomain will change for all elements when an element e_m moves. The problem is that the change is not equal for all elements. When an element e_m changes subdomain, the centre of its original subdomain S_s moves a certain distance away from e_m in direction $\overrightarrow{v_s^c} = \overrightarrow{C}_{S_s} - \overrightarrow{coord(e_m)}$. Trivially the maximal change is

$$\Delta_{max} = |\overrightarrow{C_s^{new}} - \overrightarrow{C_s^{old}}|$$

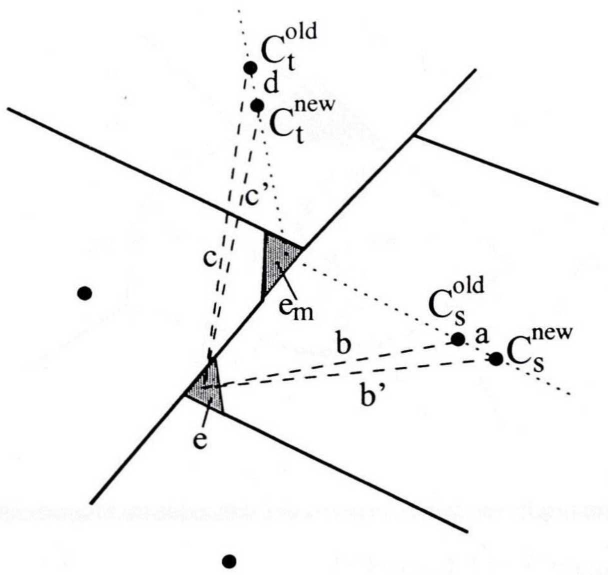
and therefore can be computed very rapidly.

In figure 4.10 it can be easily seen that with the angle between b and b' shrinking or (which is the same) an element e getting closer to the line defined by $coord(e_m)$ and C_s, $(\overline{coord(e_m), C_s})$, the change in distance for e is approximately a. Obviously, the distance of an element to the centre of its subdomain cannot change more than the centre itself. It will change by the maximal possible value only if it is located on the line $\overline{coord(e_m), C_s}$.

$\Gamma_{\mathbf{rel\_dist}}$ Similarly to the previous paragraph C_t moves additionally a certain distance towards e_m in direction $\overrightarrow{v_t^c} = \overrightarrow{C_t} - \overrightarrow{coord(e_m)}$ (see figure 4.10). Thus the maximal possible change in cost for Γ_{rel_dist} is

$$\Delta_{max} = |\overrightarrow{C_s^{new}} - \overrightarrow{C_s^{old}}| + |\overrightarrow{C_t^{new}} - \overrightarrow{C_t^{old}}|.$$

Two non-identical lines can intersect in at most one point of a plane. The maximal change therefore can only appear at the point where the two lines $\overline{coord(e), C_s}$ and



$$\begin{array}{lcl}
a & := & dist(C_s^{old}, C_s^{new}) \\
b & := & dist(C_s^{old}, e_m) \\
b' & := & dist(C_s^{new}, e_m) \\
d & := & dist(C_t^{old}, C_t^{new}) \\
c & := & dist(C_t^{old}, e) \\
c' & := & dist(C_t^{new}, e) \\
|b - b'| & \leq & a \\
|c - c'| & \leq & d
\end{array}$$

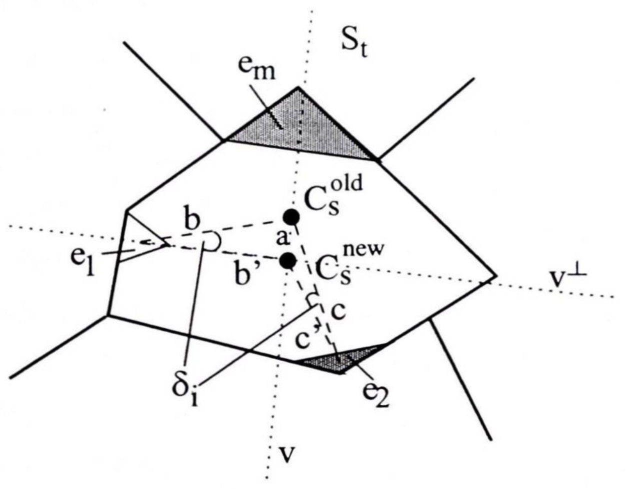
Figure 4.10: Move of subdomain centres

$\overline{coord(e), C_t}$ cross. This point is $coord(e_m)$ and thus Δ_{max} as specified above will never occur. However this threshold is on the safe side and needs no complicated calculation. Tests demonstrate, that usually this limit works very well since it causes only few updates.

Γ_{avg} and Γ_{rel_avg} Since Γ_{avg} and Γ_{rel_avg} only differ by a factor from Γ_{dist} and Γ_{rel_dist}, Δ_{max} can be determined accordingly and scaled with the appropriate factor.

Γ_{angle} It is very complicated to determine Δ_{max} for this function. The angles α_i change for all elements in the source and target subdomains since the centres of the subdomains move. In contrast to the functions discussed above, the maximal possible change cannot be determined exclusively with e_m, C_s and C_t. The maximum possible angle change δ_{max} occurs if $\overline{C_{S_s}^{old} - coord(e_i)} = v^{\perp}$ or $\overline{C_s^{new} - coord(e_i)} = v^{\perp}$. Figure 4.11 demonstrates the impact of the move of e_m from S_s to S_t on two elements e_1 and e_2. Element e_1 is positioned such that its angle will change most since $\overline{C_s^{new} - e_1} = v^{\perp}$ whereas the angle for e_2 will change less. For an arbitrary triangle $\triangle(C_s^{old}, C_s^{new}, coord(e_2))$

$$\cos\delta = \frac{a^2 - c^2 - c'^2}{-2cc'}$$



$$
\begin{aligned}
a &:= dist(C_s^{old}, C_s^{new}) \\
v &:= \overline{C_s^{old} - C_s^{new}} \\
b &:= dist(C_s^{old}, e_1) \\
b' &:= dist(C_s^{new}, e_1) \\
c &:= dist(C_s^{old}, e_2) \\
c' &:= dist(C_s^{new}, e_2)
\end{aligned}
$$

Figure 4.11: Change of angles

$(a, c, c'$ as in figure 4.11). Thus

$$\delta_{max} = \max(\arccos\left(\frac{a^2 - c^2 - c'^2}{-2cc'}\right)).$$

Since it is dependent on $dist(e_i, C_s)$ it is not possible to determine Δ_{max} without comparing all values for $dist(e_i, C_s)$s with each other. With the knowledge of $dist_{min}(e_{min}, C_s)$ a rough threshold could be specified by simply assuming that $\overrightarrow{C_s} - \overrightarrow{coord(e_{min})} = v^{\perp}$. However, $dist_{min}(e_{min}, C_s)$ cannot be calculated in constant time.

The change in the qualities for Γ_{angle} are usually very small. Above all Γ_{angle} is not suitable to be used on its own but can be very helpful as an additional assessment (see section 6.2.3). It therefore seems sufficient to update only those elements that need to be updated for the main rating function.

4.3 Calculating the flow

4.3.1 Introduction

Before elements can be migrated it has to be determined how many of them have to be moved on which borders. Several methods can be applied to solve this problem. They differ in their efficiency, in the quality of their result and in their capability for parallelisation. Two methods were used in this work: variants of the well known diffusive and Generalized Dimension Exchange methods described in sections 3.3.1. The main goal of calculating the flow is to balance the work load in the mesh. The resulting flow prescribes exactly where a subdomain has to send a specified number of elements and has an immense impact on the resulting partition. A good flow will support the generation and/or preservation of well shaped subdomains whereas a unfavourable flow might make it impossible to find a suitable set of elements to migrate without destroying the Aspect Ratios. The flow should also prescribe only as little data migration as necessary, which sometimes might conflict with its previously mentioned objective.

The AR is much more sensitive to the prescribed flow than cutsize. This is because of the extreme difficulties to find a general global strategy for optimising cutsize. There is no way to predict the development of (global) cutsize when overcoming a local optimum without looking at all possibilities. This is due to the regular structure of a element graph where almost all elements have a degree of 3 (the relatively few exceptions being those on the boundary). On the contrary it is possible to have a direct global view for optimising AR. Forcing the flow on certain subdomain edges by following a global strategy can influence the development positively. Assuming a function is known that rates subdomain edges on their suitability for optimising Aspect Ratio this knowledge can easily be used to control the flow by *weighting* the subdomain edges.

4.3.2 Border qualities

The main effort for the flow calculation is produce a weighting for the edges of the subdomain graph in a sensible way. The weighting aims to produce flows that facilitate finding sets of elements to be moved that can optimise (or at least do not worsen) the subdomain shapes. The weights on subdomain edges aim to predict on which borders migrating elements will improve Aspect Ratio. Assuming a good rating of subdomain edges is given, the advantage of this approach is that a flow calculated with this paradigm may give the global view which the local method might miss.

In conventional flow algorithms the weight of an edge expresses its cost. The cost of a path in the subdomain graph then is the sum of its edge weights. In the context of parallel flows this is not necessarily suitable. As mentioned above, weighting the subdomain edges aims to find borders on which element moves will follow the global strategy to produce good Aspect Ratios. Assuming that all subdomains have enough elements the number of edges with flow does not affect the amount of time needed to migrate elements, since the actual transportation of elements runs in parallel, independently on each of the participating borders, the few exceptions being those that transport data to the same subdomain, which are usually not within one migration path. Hence the path length should not influence the path quality and a series of 'good' edges should not be punished. This approach clearly contradicts the common aim to minimise the data flow but has shown to produce excellent results, [28]. In the following the term *border quality* will be used to distinguish it from the conventional use of *weight*. To conform with the meaning of quality, a high quality will be assigned to borders that promise to optimise global cost and low values to those that should be avoided.

Usually an edge of a subdomain graph is undirected, but the above approach allows the assignment of different qualities to the same edge depending on the direction. This is very reasonable since moving elements in different directions should produce different costs. An undirected approach could not meet the requirements of rating edges for their effect on the global Aspect Ratio. The quality of the subdomain edge pointing from S_i to S_j will be denoted as ϕ_{ij}.

Average element quality

One way to determine the border qualities ϕ_{ij} makes use of the element qualities, which express how profitable it is to migrate an element to another subdomain (see section 4.2). For each border its quality is the average quality of the elements on it. This gives an overall impression of the quality of the border. If for example the element quality is the distance from the subdomain centre, the sides of the subdomain are favoured that are furthest from the centre. This is probably a reasonable weighting, since the subdomain shape will become closer to a circle if elements are migrated that are far from the centre.

Figure 4.12 shows a subdomain S_s and its neighbouring subdomains. The border with S_a consists of only one edge. It is clear to see, that element e should be migrated in order to improve the shape of subdomain S_s. It is not obvious however which target subdomain should be preferred. Even if moving e to S_a produces a higher quality, moving it to S_b instead eliminates the border with S_a. This is very desirable since the fewer subdomain edges exist the fewer dependencies between subdomains occur [109].

Moving elements across short borders often produces extremely bad partitions. A mechanism to devalue very short borders and thus offer the elimination of such an inter-subdomain dependency is needed. Elements located on more than one border or those having a vertex that is adjacent to more than one subdomain

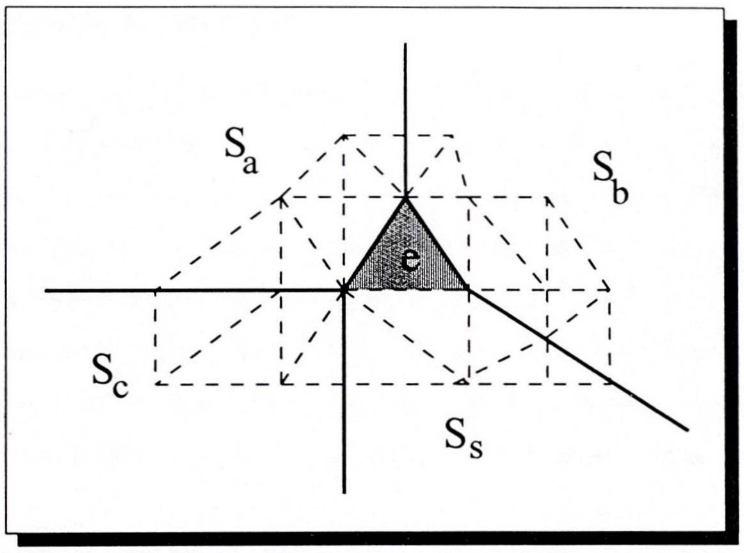


Figure 4.12: A short border

play a pivotal role in this. The mechanism proposed here is that those corner elements additionally add their negative quality for each additional border the element or (at least) one of its vertices is located on. Elements on a corner where three subdomains meet thus add zero to the border quality and those on a corner of four or more subdomains add a negative value.

Defining Λ_{ij} as the set of elements on the border of subdomain S_i with S_j, $|\Lambda_{ij}|$ as the number of elements on it, the border quality ϕ_{ij} can now be formulated as

$$\phi_{ij} = \frac{\sum_{k=1}^{k \leq |\Lambda_{ij}|} \Psi(e_k)}{|\Lambda_{ij}|}, \quad e_k \in \Lambda_{ij},$$

$$\Psi(e_k) = \begin{cases} \Gamma_j(e_k) & : \ e_k \text{ and its vertices are adjacent to 2 subdomains} \\ 0 & : \ e_k \text{ and its vertices are adjacent to 3 subdomains} \\ -\Gamma_j(e_k) & : \ \text{otherwise} \end{cases}$$

This devaluation has only very little effect on the quality of long borders but the value of short borders will decrease heavily. In figure 4.12 the border quality ϕ_{sa} between S_s and S_a will become negative since only one element contributes to the border quality but is devalued by two neighbouring subdomains (S_b and S_c). On the contrary the border between S_s and S_b consists of many edges and its average quality will balance the devaluation easily.

The border functions will be denoted with Φ with indices similar to the element functions Γ. For example the border function that represents the average distance of the elements to their subdomain centre is Φ_{dist}.

Element independent functions

Two extra functions are used that are independent from the element qualities. Φ_{ne} assigns to a subdomain edge the number of edges that are located on the respective border. Thus long borders are favoured and short borders are avoided. Although on first glance this might be a good assessment this strategy does not consider the neighbouring subdomain and as discussed in section 2.2.2 the lengths of borders are not necessarily related to the actual subdomain shape. Therefore no exact prediction of the behaviour of Φ_{ne} could be made. However, tests show in section 6.2.2 that the negative aspects overrule and using Φ_{ne} leads to bad results.

For comparison Φ_u denotes the function assigning a uniform value (1) to all edges which leads to the unweighted subdomain graph.

4.3.3 Diffusive methods

The main idea and strategies of the diffusive and GDE methods were introduced in sections 3.3.1 and 3.3.2. These two methods were enhanced by the paradigm of border qualities presented above. When calculating the current number of elements that are to be moved, the uniform α is replaced by a corresponding α_{ij}. Whenever a subdomain S_i examines S_j and their difference in load, the size x of the transferred elements is now calculated as

$$x = \alpha_{ij} \cdot (Load_i - Load_j).$$

The edge factors α_{ij} include the border qualities ϕ_{ij} and are scaled to values $\alpha_{ij} \leq 1$:

$$\alpha_{ij} = \kappa_i \cdot \phi_{ij}.$$

To assure the algorithm converges, the condition $\sum_j \alpha_{ij} \leq 1$ must be fulfilled, [44]. Thus for each subdomain S_i the scaling factor κ_i is determined as

$$\kappa_i = \frac{1}{\max_j(\phi_{ij}) \cdot degree(S_i)},$$

where $degree(S_i)$ specifies the number of neighbouring subdomains of S_i. These local κ_is did not significantly improve the original algorithm, [28, 86]. Since the geometries of the subdomains can differ heavily the values of α_{ij} were not globally scaled and thus could not properly direct the flow through the subdomain graph.

However, the introduction of a single, global

$$\kappa_i = \kappa = \frac{1}{\max_{jk}(\phi_{jk} \cdot degree(S_i))}$$

solved the problem by providing a more global perspective. Although this takes away the purely local nature of the diffusive method one global communication at the beginning of the algorithm was found to improve the behaviour that much, that this is tolerated. In algorithm 4.3 the resulting weighted diffusive method is presented.

Algorithm 4.3 Weighted_Diffusion

```
1: FOR ALL (subdomains $S_i$) DO IN PARALLEL
2:   $\kappa = \frac{1}{\max_{jk}(\phi_{jk} \cdot degree(S_i))}$;
3:   WHILE not converged DO
4:     $Load_i^{new} = Load_i$;
5:     FOR ALL neighbouring subdomains $S_j$ DO
6:       IF ($Load_i > Load_j$) THEN
7:         $\beta = \phi_{ij}$;
8:       ELSE
9:         $\beta = \phi_{ji}$;
10:      END IF
11:      $x = \kappa \cdot \beta \cdot (Load_i - Load_j)$;
12:      $Flow_{ij} = Flow_{ij} + x$;
13:      $Load_i^{new} = Load_i^{new} - x$;
14:    END FOR
15:    $Load_i = Load_i^{new}$;
16:  END WHILE
17: END FOR
```

Similarly to section 3.3.2 the GDE method simply updates the loads after each communication instead of updating it at the end of each outer loop.

Directed graph

As mentioned in section 4.3.2 the border qualities ϕ_{ij} and ϕ_{ji} are generally not equal. It is however vital to the diffusive algorithms that the diffusion matrix is symmetric and each process computes the same flow as its neighbours. To ensure this the weighting factors α_{ij} are determined in each iteration individually. Practically each process knows the qualities of all its edges in both directions and decides which factor to use as follows.

$$\beta_{ij} = \beta_{ji} = \begin{cases} \alpha_{ij} & : \quad Load_i > Load_j \\ \alpha_{ji} & : \quad Load_i < Load_j \\ 0 & : \quad Load_i == Load_j \end{cases}$$

$$x = \beta_{ij} \cdot (Load_i - Load_j)$$

Note that for $Load_i == Load_j$ β_{ij} is irrelevant anyway. It is clear to see that the above strategy achieves the aim of ensuring a unique and symmetric diffusion matrix for each iteration whilst considering the suitable border qualities in the directed subdomain graph.

Test results

During comprehensive tests which are presented in detail in section 6.2.2 edge weighting showed to meet the expectations and improves the basic diffusive methods. Selected functions ($\Phi_{rel\_avg+border+angle}$) show results that are on the average up to 7% better than conventional diffusion. Although most of the proposed functions produce better results than the unweighted version certain weighting strategies counteract the generation of good Aspect Ratios. For example Φ_{ne} generally does not produce better results even though in certain examples Φ_u is worse. For a comprehensive discussion of the test results see chapter 6.

4.4 Pre/Postprocessing

4.4.1 Preshaping

One main weakness of the presented strategy is that if large data movement is necessary to satisfy the flow, some subdomains might tend to degenerate during a strongly moving simulation and may even fall into several disconnected parts. For very large imbalances even weighting the subdomain edges sometimes cannot inhibit unfavourable flows that lead to bad Aspect Ratios. A preprocessing has been added that, in case of very poor subdomain shapes, hopefully provides a partition with better Aspect Ratios to start the actual load balancing algorithm with.

If the worst AR is high enough, the subdomain with the worst AR selects all elements that are beyond the average distance of its border and outer elements (see Γ_{avg} in section 4.2.2). The selected elements are now migrated to the best subdomain that is chosen with the selected element quality function and afterwards this subdomain and all its neighbours are locked. If at least one of the remaining unlocked subdomains' Aspect Ratios is above the threshold, this will be repeated until no subdomain is unlocked or all remaining subdomains' ARs are below the threshold. The resulting strategy is presented in algorithm 4.4.

Optimising the worst subdomain first aims to keep the maximal Aspect Ratio low. Of course this strategy does not guarantee that the average or maximal

Algorithm 4.4 Preshape

```
1: $p = i | AR_i = \max_j(AR_j)$;
2: $ar = AR_p$;
3: WHILE $ar > threshold$ DO
4:   FOR ALL elements $e_i \in S_p$ DO
5:     IF $(dist(e, C_p) > D_p^{avg})$ THEN
6:       move $e$ to best neighbouring subdomain;
7:     END IF
8:   END FOR
9:   lock $S_p$;
10:  FOR ALL neighbours $S_i$ of $S_p$ DO
11:    lock $S_i$;
12:  END FOR
13:  $p = i | S_i$ is unlocked AND $AR_i = \max_j(AR_j)$;
14:  $ar = AR_p$;
15: END WHILE
```

AR will actually decrease. For example when the second worst subdomain receives unfavourable elements from the worst, this could increase the worst and probably also the average Aspect Ratio. However, determining the optimal set of subdomains that additionally are not adjacent to each other is NP-complete (Maximum Vertex-Cover [42] with an additional optimisation criterion). This approach seemed to be very reasonable and showed to produce good results, if not called too frequently.

The locking of the neighbouring subdomains (line 11) prevents the subdomains from loosing connectivity. If two adjacent subdomains simultaneously move some of their elements to each other they can easily become disconnected, since they do not know in advance if and where new elements will arrive. In a parallel environment each processor executes exactly the same algorithm on a local representation of the subdomain graph and possibly optimises the shape only of its own subdomain(s). In other words the loop for moving the elements (lines 4-8) is empty if the subdomain is not owned by the process. But the locking of subdomains takes place and since this needs only little time, all elements move nearly synchronously.

The local representation of the subdomain graph needs to include the Aspect Ratios of the subdomains, since they are needed to determine the subdomain(s) with the maximal Aspect Ratio(s). For the quality functions Γ_{rel_dist}, Γ_{rel_avg} and Γ_{angle} a global communication step is needed anyway to transmit the centres of the subdomains and/or the average distance of border and outer elements. Hence all the other information about the subdomains (Aspect Ratios, adjacency and loads) is transmitted at the same time which saves an extra communication step.

Tests showed that fixed thresholds do not work very well, since they invoke the pre-optimisation not often enough or too frequently. Thus the threshold is chosen dependent on the actual partition. It is determined with the average subdomain Aspect Ratio AR_{avg} and the Aspect Ratio of the (unpartitioned) mesh AR_M. During tests

$$threshold = min(1.7 \cdot AR_{avg}, 1.7 \cdot AR_M)$$

has shown to produce the best results.

4.4.2 Postprocessing

It is possible that some solvers need subdomains to be connected. The algorithm presented so far cannot guarantee this property. Thus an additional optional step is provided to ensure connectivity. At the expense of perfect balance the postprocessing step searches for disconnected sets in each subdomain. Starting with the smallest of these sets, they are dissolved until only one set of each subdomain is left. This procedure again can leave some subdomains disconnected. This is due to the way the elements of the dissolving parts are distributed. Each element in the subdomain part will move to the subdomain which produces the best quality (similar to the preprocessing step in section 4.4.1). If these neighbouring subdomains are disconnected parts themselves, again, the parallel execution might cause a concurrency problem: since the neighbouring parts are now dissolved the migrated elements build again a disconnected part. Thus the postprocessing has to be executed in a second outer loop until no more disconnected sets are left.

Figure 4.13 shows that this strategy could run into an infinite loop if the number of elements were not discrete. The loop will end eventually when only one element is left. It is even possible that subsets of elements move in circles. This can only occur if the dissolved subsets send their elements to neighbouring subsets that are dissolved at the same time. This is extremely unlikely and to prevent this the algorithm changes dynamically the quality functions Γ and Φ for selecting the elements and for rating the subdomain borders. Even now the algorithm can still run into an infinite loop and therefore it is terminated when the number of moved elements is greater than the number of elements in the mesh. For these extremely improbable cases this step fails. However, the balancing algorithm usually produces only very rarely and very few disconnected sets and

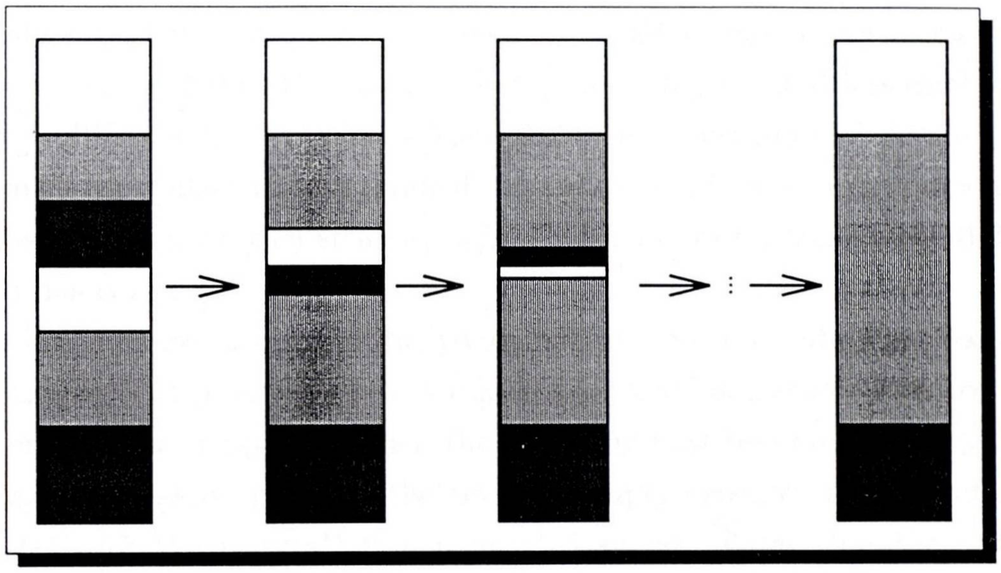


Figure 4.13: Removing of disconnected subsets

hence if the postprocessing is executed, it usually terminates successfully after one loop with no or only little element migration.

Trying to restore balance after the postprocessing would mean going through all steps again and still the connectivity could not be guaranteed. Hence the resulting imbalance is left until the load balancer is invoked again. Due to the good results of the main algorithm the imbalance usually is tolerable and another complete load balancing step would produce too big an overhead. A small imbalance will probably slow down the parallel solver only for an inconsiderable amount of time and the next run of the load balancer (after the refinement) will usually restore perfect balance.

If the main concern of the load balancing step is not the Aspect Ratio but perfect balance it might be possible to introduce a new element function which, in case of multiple target subdomains, would choose the one with lower load to send the element to.

4.5 Asynchronous Multiple Phase Diffusion

During extensive tests with the basic algorithm (4.1) presented above, it was observed that in cases where much data movement is necessary to balance the load, some subdomains are forced to send away many unfavourable elements which completely destroys their Aspect Ratio. A new strategy was developed which does not wait for the flow calculation to have computed a fully balancing flow but starts migrating when the flow has reached a certain threshold. Due to this interleaved migration it is hoped that smaller amounts of elements are moved at a time and thus the subdomains become degenerated less easily. This approach is different from the Wavefront Diffusion algorithm of Schloegel *et al*, [89], because their algorithm is applied on a completed balancing flow which is to be satisfied, whereas this strategy might start migration before the diffusive algorithm has converged.

The idea is to execute the diffusive algorithm until the outgoing flow is too large or an adjacent process requests migration. A subdomain can request from a neighbour to start migration when the incoming flow becomes too big. When the migration phase is triggered the process simply executes the migration algorithm 4.2 with the (partial) flow computed so far. After that the diffusion algorithm is continued. One main aim was to keep the algorithm as asynchronous as possible thus not all processes necessarily migrate at the same time. One processor might migrate elements while others are still only diffusing.

Generally each processor communicates in each iteration with each of its neighbours and examines their messages. These messages always contain the information needed for the diffusion but optionally might request migration or contain migration data. If migration was requested by one of its neighbours (or migration is necessary) the subdomain will start the migration before the next communication round.

As described in section 4.2.4 a consistent physical migration of elements cannot be done in one step. Thus, in a synchronous environment a process needs three iterations to complete a migration. Applying this directly to an asynchronous environment the different processes could be in several phases of the migration or diffusion, which was the motivation for naming this strategy Multiple Phase Diffusion (MPD). However, since in this asynchronous approach not all neighbours are necessarily in the same state (phase). Hence it is useless to perform the three steps (requesting elements, migrating and updating) in separate rounds, because the neighbours cannot respond in time. Thus all three communications are done at once and this method will be referred to as Asynchronous Multiple Phase Diffusion (AMPD).

In algorithm 4.5 the resulting strategy is presented. The outer loop (lines 3-30) is performed until the diffusive algorithm has converged and no element was migrated. Even if diffusion has converged elements might be moved to optimise the partition. For a description of the function `Enable_Optimising()` in line 27 see section 4.2.

Algorithm 4.5 Asynchronous Multiple Phase Diffusion

```
 1: FOR ALL (subdomains $S_i$) DO IN PARALLEL
 2:   $\kappa = \frac{1}{\max_{jk}(\phi_{jk}) \cdot degree(S_i)}$;
 3:   REPEAT
 4:     IF ($migrating$ OR $optimising$) THEN
 5:       Migrate( virtually );
 6:       Pack( requests, copies, objects, updates );
 7:       Send_Messages();
 8:       $migrating$ = off;
 9:     END IF
10:     FOR ALL neighbouring subdomains $S_j$ DO
11:       Receive_Message();
12:       IF ($Load_i > Load_j$) THEN
13:         $\beta = \phi_{ij}$;
14:       ELSE
15:         $\beta = \phi_{ji}$;
16:       END IF
17:       $x = \kappa \cdot \beta \cdot (Load_i - Load_j)$;
18:       $Flow_{ij} = Flow_{ij} + x$;
19:       $Load_i^{new} = Load_i^{new} - x$;
20:       Unpack( requests, copies, objects, updates );
21:       IF (migration_requested()) THEN
22:         $migrating$ = on;
23:       END IF
24:     END FOR
25:     $Load_i = Load_i^{new}$;
26:     IF (Balanced() AND NOT $optimising$) THEN
27:       Enable_Optimising();
28:     END IF
29:     $migrating$ = ($migrating$ OR migration needed);
30:   UNTIL (converged AND no element was migrated AND $migrating$ == off)
31: END FOR
```

The threshold for requesting or triggering migration was found to be best determined dependent on the number of adjacent subdomains. The more neighbours a subdomain has the more dangerous a large flow on one of its border is. Thus the migration and the migration request is triggered if the outgoing/incoming flow is chosen as

$$flow_{i/o} > \frac{Load}{degree()},$$

where *degree*() is the number of adjacent subdomains. More details on this algorithm are very technical and do not contribute to a better understanding. However they can be found in section 5.2.4. AMPD was tested on several scenarios and the results are presented in section 6.3.3.

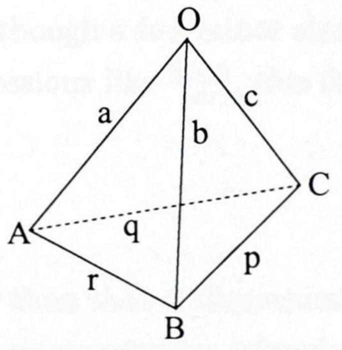
4.6 Extensions to 3D

Although this work is focused on 2-dimensional problems a few aspects on possible extensions to 3D are discussed in this section.

4.6.1 Aspect Ratio

Since Aspect Ratio in 3D considers faces rather than edges and volumes rather than areas the definition for 3 dimensions is different than in 2D and was given in section 2.2.3. It is slightly slower to compute this Aspect Ratio since tetrahedrons/bricks have more faces than triangles/rectangles have sides and the corresponding values are more costly to determine. Whilst the area of a triangle easily computed by three vector sums plus four vector multiplies and one scalar-vector multiply (see chapter 1), the volume of an arbitrary tetrahedron [13] is as complicated as

$$V = \sqrt{\frac{1}{288} \cdot \begin{vmatrix} 0 & r^2 & q^2 & a^2 & 1 \\ r^2 & 0 & p^2 & b^2 & 1 \\ q^2 & p^2 & 0 & c^2 & 1 \\ a^2 & b^2 & c^2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{vmatrix}}$$



The two definitions for the different dimensions $d \in \{2,3\}$ can both be computed in $O(c_d \cdot n)$, however since in 3D the definition of AR has two exponential operations instead of one and due to the afore-mentioned the constant for 3D c_3 is greater than c_2 for two dimensions.

Following the approach in [102] the definition of Aspect Ratio can be generalized to any dimension d to facilitate the extension to 3D. For subdomain S, let Θ_S be the boundary length in 2D or the area of surface in 3D and let Ω_S be the area in 2D or the volume in 3D the generalised formulation for $d \in \{2,3\}$ can then be formulated as

$$AR = \frac{\Theta_S^2}{\pi^{\frac{2}{d}} \cdot (2d \cdot \Omega_S)^{\frac{2d-2}{d}}}.$$

Provided elements are given in a uniform manner, i.e. that for each element e the boundary/surface area θ_e and area/volume ω_e are given, all computations can be handled similarly, whether they are for 2D or 3D.

4.6.2 Quality functions

In principal the quality functions described in section 4.2.2 can be directly mapped to 3D by replacing areas by the generalised Ω/ω and the border/edge lengths by Θ/θ. Γ_{border} is now replaced by

$$\Gamma_\theta = \theta_e^t - \theta_e^i$$

where θ^i and θ^t are defined similarly to b^i and b^t. The distance functions Γ_{dist}, Γ_{avg}, Γ_{rel_dist} and Γ_{rel_avg} and Γ_{angle} do not change at all since they do not consider the area or borders. However Γ_{AR} must be formulated as

$$\begin{aligned}\Gamma_{AR} \quad = \quad & \left[\frac{\Theta_s^2}{\pi^{\frac{2}{d}} \cdot (2d \cdot \Omega_s)^{\frac{2d-2}{d}}} + \frac{\Theta_t^2}{\pi^{\frac{2}{d}} \cdot (2d \cdot \Omega_t)^{\frac{2d-2}{d}}}\right] \\ & - \left[\frac{(\Theta_s - \theta_e + 2 \cdot \theta_e^i)^2}{\pi^{\frac{2}{d}} \cdot (2d \cdot (\Omega_s - \omega_e))^{\frac{2d-2}{d}}} + \frac{(\Theta_t + \theta_e - \theta_e^t)^2}{\pi^{\frac{2}{d}} \cdot (2d \cdot (\Omega_t + \omega_e))^{\frac{2d-2}{d}}}\right].\end{aligned}$$

This definition is extremely complicated and although a few minor simplifications could be made, e.g. substituting constant expressions like $\frac{2d-2}{d}$, this function will still be most costly to calculate.

4.6.3 Cardinality

3-dimensional objects have a higher cardinality than their 2-dimensional counterparts, e.g. tetrahedrons have 4 neighbouring elements whereas triangles have only three. This property most probably is not dangerous to the algorithm. It is even likely that the new heuristic performs better in 3D than in 2D since it has higher degree of freedom which minimises the risk of local minima. This assumption is backed with experience and literature [15] on local partitioning algorithms.

Even though the original algorithm considers only three candidate target subdomains for each element the heuristic should work equally well as if an arbitrary number of candidate subdomains was allowed. This assumption can be made because the algorithm considers not any three subdomains but those three which produce the best quality, and it is extremely unlikely – even in 3D – that an element is adjacent to more than 3 subdomains and at the same time the functions assign an unsuitable quality.

The cardinality of the subdomain graph will generally increase as well when going to 3D. Again this should be advantageous to the algorithm. As it will be observed and explained in section 6.2.2 Weighted Diffusion works better the more subdomains there are and thus the higher the cardinality is.

4.7 Summary

In this chapter a new heuristic for load balancing in adaptive Finite Element Simulations was introduced. Besides the aim of generating/preserving well shaped subdomains the following new ideas were presented

- the concept of weighting the subdomain edges for the diffusive algorithms was introduced in section 4.3.2
- in the same section different strategies for weighting the subdomain borders were presented
- in section 4.2.2 several new functions were developed to select migrating elements
- algorithm 4.2 in section 4.2 introduces the idea of optimising all borders at the same time on each subdomain (and not sequentially one after the other as common strategies do)
- a new update procedure presented in section 4.2.5 provides almost constant time complexity for updates after each move
- in section 4.2.4 a new object oriented 3-phase migration was introduced which ensures data consistency
- a simple but effective preprocessing step improves the heuristic (section 4.4.1)
- in section 4.5 a new experimental algorithm is presented which interleaves the flow computation with the actual element migration

In section 4.2.2 the introduced functions were also compared with respect to several important aspects that influence their effectivity and capabilities. Finally in section 4.6 a possible extension of the algorithm to three dimensions was discussed. The results from this section suggest that such a generalisation does not require additional concepts and could be implemented relatively easily.

Chapter 5

Implementation

The heuristics presented in chapter 4 are implemented as an integral part of the PadFEM project at the University of Paderborn, [21]. The project is written in C++ and is intended to be a development environment for FEM-related algorithms. It provides a base data structure, [83], solvers, [11, 50], an error estimator and mesh refinement strategies, [59] for parallel and sequential applications. In this chapter the most important aspects of the implementation of the load balancer are discussed. The chapter closes with a description of a generic Graphical User Interface (GUI) which was developed as part of this work.

5.1 Data structure

The base data structure provided by PadFEM makes heavy use of the object oriented aspects of C++ and offers basic functionality for data generating, data access and consistency ensurement as well as the possibility to derive more advanced and specialised structures.

The mesh is represented by a class called *mesh2d* which contains all information and objects. The mesh consists of a number of subdomains. Each of them holds its vertices, edges and elements. All the objects (vertices, edges and elements) are connected to all their neighbouring objects, e.g. a triangular element has pointers to its three vertices and its three edges and vice versa. This allows direct and thus fast access to neighbouring objects. Common data structures used elsewhere, e.g. [64, 98], do not provide such a high connectivity which often leads to large computational overheads caused by searching for objects. On the other hand ensuring data consistency often implies computation overhead which could be omitted by using a more specialised data structure.

If the applications are executed in parallel each process(or) holds one mesh with at least one local subdomain. Any calculation is performed on these local subdomains and their sub-objects. Depending on the requirements of the application an extra arbitrary sized border of objects can be stored. The objects belonging to this halo are stored in the corresponding subdomains which are tagged as non-local. Thus duplicated data can be accessed in the same manner as local objects.

An infrastructure is provided to easily communicate with other processes. Also a mechanism is provided to duplicate elements, vertices and edges and to migrate them virtually and physically. For communication a class is provided that hides the implementational details of the underlying message passing software. Since originally only very rudimentary functionality had been implemented and only PVM was supported they were substantially enhanced to meet the requirements of this work. During this work so called *wrappers* were developed to be able to use MPI, [72], and files as communication channels. In addition a thread wrapper was implemented to use shared memory on a appropriate systems. All these wrappers now support standard communication tasks such as gathering and global reduction with an uniform interface.

For the implementation of the algorithms of this work new classes for the mesh, the partitions, elements, vertices and edges were derived from the base data structure. These classes include all functionality needed as methods and additional members that were needed to deal with data that is not found in the base structure, such as the subdomain graph, the centres of elements and many more. For the candidate moves a new class was derived from a base list structure of the Standard Template Library (STL) which is heavily used in PadFEM.

5.2 Load balancer

5.2.1 Termination

Ideally Algorithm 4.1 terminates when the load is balanced and no more elements have been migrated. Unfortunately the local nature of the heuristic might not reach a fully balanced state at all. Thus a mechanism was introduced to prevent the algorithm from an endless loop.

Algorithm 4.2 allows the prescription of not only the flow and a maximal load imbalance but also a tolerance in the flow change (see section 4.2.1). An initial mechanism to accelerate the attainment of a balanced state is to decrease the latter tolerance whenever one iteration did not restore balance. If fractions of elements could be moved this would be sufficient. Since this is obviously not possible another more rigorous strategy was added.

Theoretically the global flow (defined as the sum of all outgoing flows) should decrease with each loop of Algorithm 4.1. The discrete nature of the load balancing problem however often inhibits such a development. Nevertheless it is highly improbable that, if the flow increases too often, a balanced partition might be found at all. Therefore the loop is terminated when the flow increased a certain number of times although the maximal tolerated flow change was smaller than 0.4. A similar strategy is also applied within Jostle [97, 98] and showed to perform very well, since the execution time often is only a fraction of the time needed without this limit but the results usually are equally good.

5.2.2 Diffusion

For efficiency reasons the diffusive algorithms were implemented in a non-parallel fashion (except AMPD). For this each process creates the complete subdomain graph and computes the flow locally. The convergence of diffusion tends to require most of the time for the last 5% imbalance and therefore would need many communications if executed in parallel.

Although computing the flow only once should theoretically be sufficient, the discrete nature of balancing the number of elements means a single flow computation might not be in the position to create perfect balance for the following reason. The computed flow could prescribe a flow where no subdomain has an outgoing flow >= 0.5 even though the (discrete) load is not balanced. Therefore no elements are migrated and the imbalance remains. If on the other hand the flow is computed from scratch it might prescribe a more practical flow. In order to save computation time the flow is only updated most of the time. Note that moving elements might change the flow and thus the flow might not be a perfect balancing flow after migration took place. Thus only every four iterations of the outer loop in Algorithm 4.1 the diffusion algorithm is started from scratch (i.e. with no flow on any edge). Most of the time however the current (eventually not perfectly balancing) flow is used as the initial solution.

5.2.3 Neighbourhood

Internally, not only the subdomains that actually share at least one edge are stored as neighbours, but also those subdomains that hold at least one copy of one of the local objects. The neighbouring subdomains are also included in local communications.

When the locations of newly generated copies are sent to the original (see section 4.2.4) or while receiving objects from disconnected subdomains (for incrementally updating the halo) new adjacencies can be generated. Although the new edges might be generated while receiving data from a neighbour but after sending own messages, the new neighbour might have known about this edge before and thus might wait for a message of this neighbour. Therefore an extra dummy message step for those new neighbourhoods is necessary to keep the program from getting trapped in a deadlock.

5.2.4 Asynchronous Multiple Phase Diffusion

Requested elements

Due to the asynchronous nature of the algorithm elements might be requested by neighbouring subdomains after eventually the migration was completed. Those requests cannot be satisfied in the same iteration. Therefore the algorithm will not terminate until the next iteration has completed, in which the requested elements are virtually migrated no matter how (dis)advantageous this move may be, since this is done before the local migrations are considered. Nevertheless, if a request was followed, the element might be moved back again during the optimisation.

Data consistency

Another difficulty is that the asynchronousity of the algorithm makes it hard to keep the data consistent. As described in section 4.2.4 each original object is in charge of keeping its copies up to date which is done by sending update messages to all subdomains holding one of its copies (only necessary after being migrated). For this it needs to know where the duplicates are and this information is appended to the migration messages. Since this is done asynchronously a message containing the information of a new copy could arrive after the original object was migrated. In such a case the subdomain receives the information about a new copy of an object that it does not own any more. Fortunately this object must be up-to-date on this processor since it must have been sent away in the same round. Thus the new location can be passed to the actual owner in the next iteration. Obviously this could theoretically continue until all subdomains have been 'visited' by the object before a consistent state is reached. This can however happen only if the object is never kept for one consecutive iteration at one subdomain. Fortunately this is extremely unlikely and was never observed.

Termination detection

Since diffusion tends to converge very slowly an extra termination detection was not implemented but after each round a global communication was used to detect balance and the global state. Although this might not match the idea of asynchronousity it does not change the behaviour of the algorithm and was sufficient for experiments with the idea of AMPD.

Diffusion and GDE

The implementation of AMPD resembles the idea of diffusion and theoretically an adjustment to GDE should be possible. However the asynchronousity makes the actual implementation difficult. It is vital to the algorithm that the flow on each border is stored as the inverse on each of the neighbouring subdomains and furthermore β in lines 7 and 9 of algorithm 4.3 is dependent on the direction of the new flow. To be in the position to properly handle newly generated adjacencies all messages which are sent in each round need therefore to be explicitly stored.

5.2.5 Speed

The algorithms were implemented mainly to practically prove the power of the heuristic and not with special concern on the effective speed. Flexibility was preferred to speed, therefore any parameters can be set at run time and their change does not need a recompilation of the program. Thus the implementation of the heuristics presented in this thesis cannot, in terms of speed, compete with partitioners/load balancers like Jostle and Metis. These have been developed on a data structure that had been designed especially for the needs of partitioning. Although applying them might involve some transformation of data into this structure they are therefore still faster.

Two main reasons are responsible for the limited speed of the actual program, which both are due to the underlying data structure. Firstly a specialised data structure involves a much lower computation and communication overhead. When objects are migrated from one subdomain to another PadFEM moves all data attached to this object. A specialised data structure would transfer only a fraction of this amount and the attached data would be moved only after the final partition has been determined. The second slowing down factor is the pointer based approach of PadFEM, [20, 83]. All objects are created and stored individually and not – like in Jostle or Metis – held in large arrays. Thus cache hits are extremely unlikely which is well known as a main factor of speeding up programs. Furthermore the flexible approach mentioned above makes it a lot harder for the compiler to optimise the code, since for example function pointers cannot be inlined.

If all those disadvantages were eliminated, it should be possible to reach an efficiency comparable with those of Jostle or Metis, although optimising Aspect Ratio is naturally slower than optimising cutsize. This is because Aspect Ratio is floating point based and cannot be handled with integer arithmetic as optimising cutsize allows, [102]. As discussed in section 6.2.9 the actual total number of swaps during optimisation of the new heuristic is lower than for a conventional state-of-the-art multi-level partitioner which is another argument suggesting that a very fast implementation should be possible.

5.3 The graphical user interface *XFem!*

As part of the project a graphical user interface was implemented to display, analyse, verify and animate algorithms and meshes in Finite Element applications. It is implemented for the X-Windows System and can be compiled for any common Unix operating system supporting X. The design follows a strict object oriented (OO) approach, except that some restrictions for technical reasons had to be made (C++/X does not allow methods to be used as call-back procedures).

Basic functionality such as displaying the whole mesh or a single object (element, vertex, edge) is supported as well as more sophisticated features like different types of zooming and linear shading. The implementation as a library allows any application that is using the mesh2d data structure or any of child classes to use this tool.

The software interface provides functionalities to display information about any object in the mesh. It also allows the application to provide functions to replace or augment the default information functions. These are colour, label and detailed information functions for each type of object. Furthermore it provides the possibility to easily inherit the full functionality of the tool and derive a new subclass. This is very useful but only necessary to visualise information which is not part of the base mesh2d structure, e.g. the geometric centres of subdomains. Additionally the possibility for two different types of shading is provided, for example to visualize results. The actual appearance of the interface can be suppressed by specifying a command line parameter even if the module was linked into the program. ***XFem!*** which is shown in figure 5.1, also allows the user to halt and continue the program during run time at any point.

5.3.1 Basic usage

Navigation

Buttons provide basic functionalities to navigate through the mesh: zoom in and out, move up, down, left and right. The mouse supports dragging, zoom-in-and-centre, zoom-out-and-centre and selecting a region to be fully displayed. The desired function is selected with the mouse button and the key modifiers 'shift' and 'control'.

Debugging facility

If the implementing application wants to make use of the debugging facilities, ***XFem!*** provides buttons to step through the algorithm. A hierarchy is implemented defining a theoretical unlimited number of levels, which identify the priority to stop. Next to the step button a menu button is placed allowing the user to select the priority. Whenever the algorithm stops a status label shows the current state or position in the algorithm. When the application is not halted, the step button turns to a stop button which, if pressed, causes the application to stop at the next provided breakpoint. A special button is displayed which forces the program to continue until the application has finished.

A very special feature allows one to set breakpoints on objects. The info boxes provide an extra button to set this breakpoint at run time. The application then stops whenever it comes across a point where this object is given.

The menu bar

The menu bar provides functionality to deal with files, viewing and searching. The file menu allows the user to load and save files in the native file format. These only store/restore the mesh structure. PostScript output can be generated which will result in the same output as displayed on the screen (except the circles for vertices), even in shading mode and with enabled labels. In linear shading mode a file in ppm format (a common UNIX graphics file format) will be generated instead. The view menu allows the user to activate/deactivate labels for elements, vertices and edges. Additionally the displaying of edges, vertices and elements can be switched on and off. Furthermore the diameter of the circles which represent the vertices can be changed. The search menu provides the possibility to search for elements, vertices and edges. The found object is centred, marked and the information box shows.

Object information

By clicking on an element an information window will show. The information displayed is by default the element number and basic element information stored in the mesh2d structure. If the underlying application provides a function which returns other information, this will be displayed optionally, additionally or exclusively. Even if several parts in an inheritance hierarchy defined such functions, all the information can be presented. By pressing one of the modifier keys (shift or control) information for the selected vertex or edge will appear in the same way. Figures 5.2 and 5.3 show the basic information appearing in the info box and the info box after providing additional data. The use of the break button was briefly described above.

5.3.2 The software interface

Information functions

The three main types of objects (elements, vertices and edges) are the heart of information handling. For each of them three information functions can be specified by the application for colours, labels and detailed information. Their purpose is to return the desired information for a given instance of an object. The colour functions should return the colour number for the given object. Whenever the object is drawn it will be assigned the colour returned by this function. The label functions should give a string that will be shown for every object of that type when the labels for that type are activated. By default this is the number. Detailed information about an object can be provided by the application with the help of the third information function. The returned data will be displayed in the information window showing when selecting an object. Such detailed information consists of pairs: the name of the information (e.g. "Id") and a function providing this data for a given object (e.g. give_id()). The interface to ***XFem!*** supports the declaration of these functions and the decision whether the information should be displayed exclusively or if other parts of the project should share the info box (e.g.

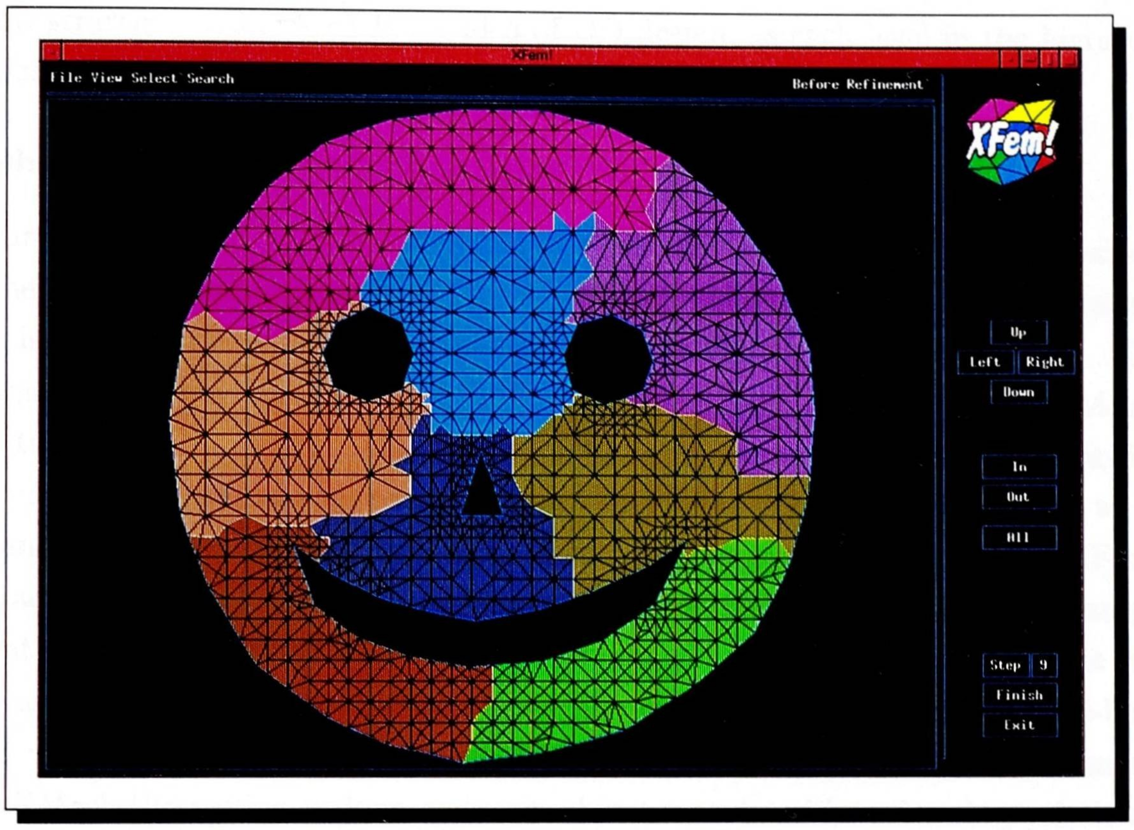
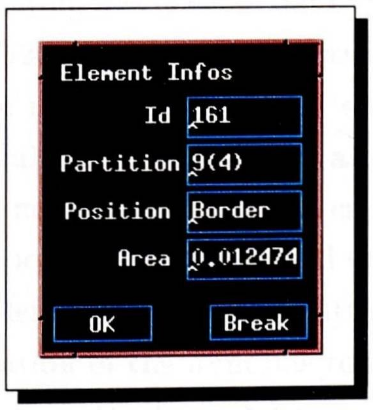


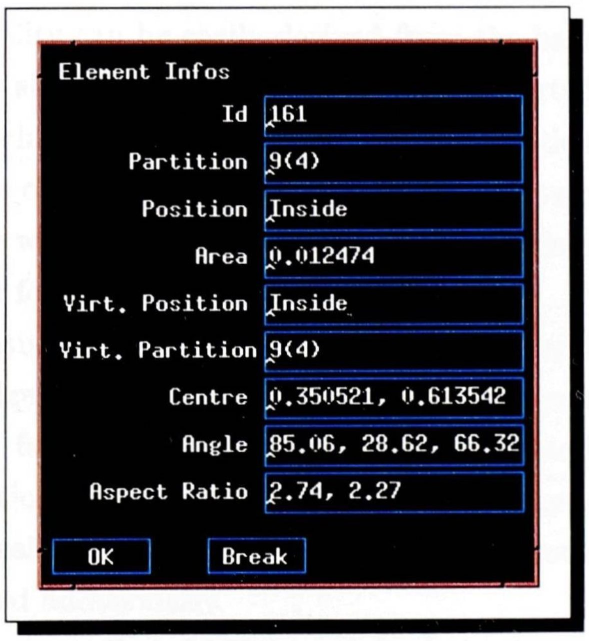
Figure 5.1: ***XFem!***



Element Infos

Id	161
Partition	9(4)
Position	Border
Area	0.012474

Figure 5.2: ***XFem!*** standard info box



Element Infos

Id	161
Partition	9(4)
Position	Inside
Area	0.012474
Virt. Position	Inside
Virt. Partition	9(4)
Centre	0.350521, 0.613542
Angle	85.06, 28.62, 66.32
Aspect Ratio	2.74, 2.27

Figure 5.3: ***XFem!*** enhanced info box

the mesh refinement might be interested in the data of the error estimator, and this strategy conforms with the idea of OO design, as each level in the hierarchy is completely responsible for its own data).

Inheritance

Three basic classes were implemented that represent different levels of abstraction. The class 'GWindow' represents the layer that is closest to the operating system. It is in charge of the actual communication with the X-window system. The interface between the mesh2d structure and GWindow is a class called 'XMesh'. It transforms geometric information of the mesh2d structure into the drawing primitives GWindow supports (e.g. the continuous coordinate system has to be transformed into the discrete pixel system). A child class from XMesh is 'XFem'. It adds the GUI to the functionality supported by XMesh. A typical example that shows this hierarchy is the user's command to draw the mesh. He/she will press on the 'All' button which causes the class XFem to invoke the call-back function for this event. It instructs XMesh to rescale and draw the whole mesh. In XMesh the scaling is done and each object translated into drawing primitives like drawing a line and passed to GWindow. Finally GWindow will execute these basic commands and when all these primitives are executed, the whole mesh is displayed on the screen.

A new class with additional functionality can be easily derived from the base class XFem. In particular the powerful shading functionality is not supported with buttons or menu items in the base class. This is due to lack of information and the aim for flexibility, since the base class works with the mesh2d structure and shading needs some data to visualise which usually is located in a child class of mesh2d. However all functions needed for shading are provided.

The menu can be easily enhanced using the methods `MakeMenu(...)` and `AddMenuItem(...)` without any knowledge of X programming. The programmer simply needs to specify the caption, the function to be executed when selected and optionally a value passed to the function when called. Almost equally simply is the definition of a new button via `MakeButton(...)`. Only some data about the location of the item has to be specified additionally.

Three methods must be overloaded to properly initialize a newly derived new child class. Four virtual draw methods can optionally be overloaded to display additional data, one for the mesh, elements, vertices and edges each. For an easy implementation of additional drawings basic procedures are provided to draw

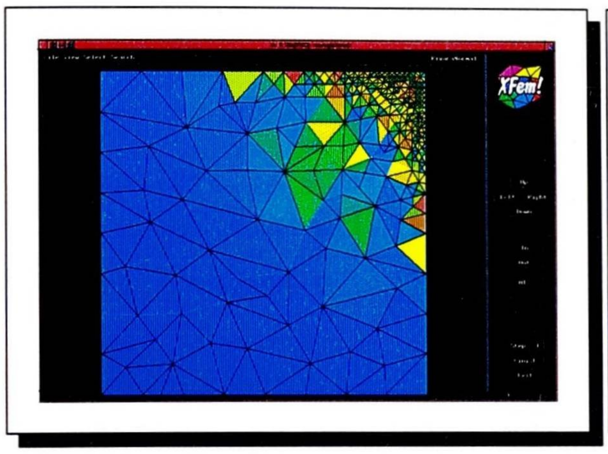


Figure 5.4: ***XFem!***: element shading

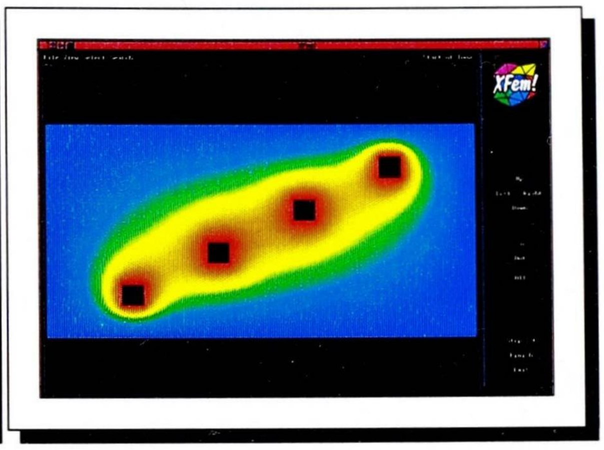


Figure 5.5: ***XFem!***: linear shading

and fill polygons and circles and to draw text. To simplify this task a text-based program is provided to automatically create the class framework and definitions.

Shading

XFem! supports two different kinds of shading: element shading and linear shading. Both use as many shades as possible but the number can be fixed with a parameter. The colours run from blue to red via green, yellow and orange. Element shading assigns one colour shade to each element. Therefore the application needs to specify a function that returns any rational value for the given element. The necessary scaling is done by ***XFem!***. This feature is very useful where an overall impression of some element data is appreciated, e.g. results of an error estimator (figure 5.4). The second method shades the whole mesh linearly according to some rational data on the vertices, which again must be supplied by the application. This can be used to display the results of a solver, figure 5.5 shows an example. To enhance flexibility up to 256 different values per element and vertex can be displayed. To make them accessible through the menu only a virtual method needs to be overloaded simply calling the parent class' method with the necessary number of entries.

Breakpoints

Wherever the programmer thinks is a suitable place to eventually halt the program, simply the call `XFemStep(priority,message,object)` has to be inserted, optionally specifying the priority of this breakpoint, a lable to be shown in case of stopping and an object of interest. If an object is specified the application will

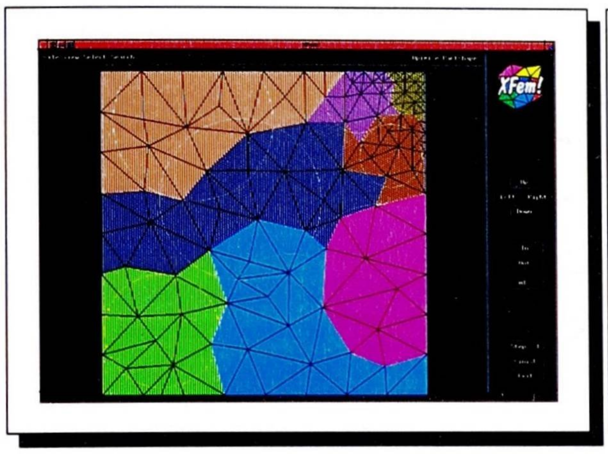


Figure 5.6: Visualisation of the average element distance with ***XFem!***

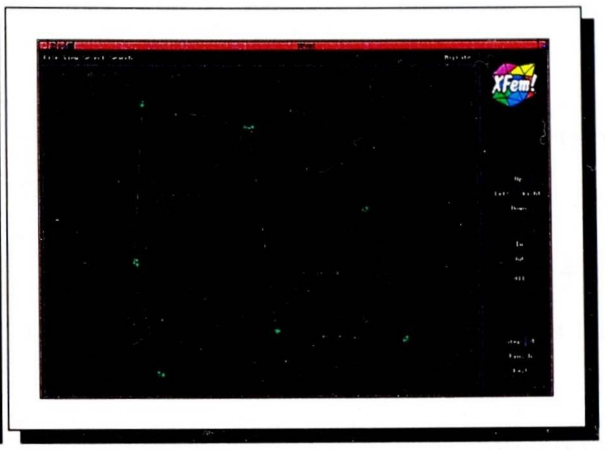


Figure 5.7: Visualisation of the flow and border qualities with ***XFem!***

be halted there when at runtime a breakpoint was set on the given object (see section 5.3.1).

Interaction

In order to be able to able to fully use the advantages of a GUI ***XFem!*** offers two methods to communicate with the user, a function to send a message and one to ask for an input. Both pop up an extra window.

5.3.3 Enhancements for the load balancer

XFem! was heavily enhanced for better support of the needs for analysing the algorithms proposed in this thesis. The information box which appears when selecting an object, displays the data, which is interesting for the load balancer, e.g. border information and the different element qualities. To gain an overall impression of the quality distribution the element shading functionality was used to display these values. The object colours were adapted in a way that virtually moved objects are displayed in the new colour. The mean distance of border elements to the centre of their partition can be displayed by white circles (figure 5.6). To do this, the menu was augmented with the necessary entries. Two items were added to the file menu to save and load a partition of a mesh. The element shading function can be activated and the desired value index can be selected. Finally the user is offered the possibility to visualise the subdomain graph with the flow that has been calculated and the border qualities (figure 5.7).

5.4 Summary

In this chapter several details of the implementation of the new heuristic presented in chapter 4 were discussed. The key issues in this chapter were

- a dynamic threshold for ensuring termination, introduced in section 5.2.1
- the incremental flow computation presented in section 5.2.2 accelerates the algorithm
- efficiency limits of the actual implementation are due to the underlying generic data structure, as discussed in section 5.2.5

Section 5.3 presents a powerful, generic and flexible GUI for dynamic FEM simulations which was developed as part of this work. A brief description of its capabilities is given in respect of its usage (section 5.3.1) and its inclusion into a software project (section 5.3.2). Also the use of ***XFem!*** for this work was demonstrated in section 5.3.3.

Chapter 6

Tests and Comparisons

6.1 Experimental setup

Several characteristics of Finite Element simulations challenge a dynamic load balancer, the most important ones are

- Focused and fixed refinement
- Focused and moving refinement
- Complicated domain shapes
- Disadvantageous triangulations

The first case resembles problems with singularities at certain places, where an error estimator usually computes a large error. Hence the refinement will be very unbalanced and very few subdomains become heavily overloaded. Thus it might be necessary to route the load from the heavy to under-weight subdomains over other vertices of the subdomain graph. This can make difficulties when the ratio of outgoing and incoming flow of some subdomains becomes large.

In applications that simulate dynamically changing systems such as shock waves the point of interest might move between two refinement phases. Heavy mesh refinement might therefore take place in non-stationary regions. The difficulty which arises from such movements is that long term developments are hard to control.

If the domain shape is very complicated the element migration can be obstructed. This can be caused by relatively small convexities or holes in the inner mesh. These places can easily become element migration bottlenecks which cause similar problems as heavily focused refinement.

Often complicated domain shapes also imply very irregular triangulations. At some points meshes can be very fine whereas other places might be extremely coarse. In particular if the differences are relatively close together the generation/preservation of good Aspect Ratios might get very difficult or even impossible. As a measure for the complexity of the triangulation the Aspect Ratio and the mesh grading can be used, [102]. Here the grading of a mesh is defined as the maximal element area over the minimal element area. Another assessment criteria for the quality of the triangulation is the Aspect Ratios of the elements. Very degenerated triangles can make it very hard to obtain a good subdomain shape. In the following the same definition of Aspect Ratio is used for elements as is used for domain shapes.

6.1.1 Test cases

To test the behaviour of the new heuristic on typical but demanding FEM scenarios five synthetic sequences of adaptively refined meshes were selected. During 10 adaptive phases all test scenarios applied a certain refinement strategy which resemble different demands of FEM simulations. The main issues of the following detailed description of the individual scenarios are summarised in table 6.1. The Aspect Ratio of the mesh, AR, and the numbers of subdomains, S, for each example are given. To express the complexity of the scenarios the number of elements N, the maximal and average element Aspect Ratios ar_{max} and ar_{avg} are listed as well as the mesh gradings. The characteristics that change during the simulation are given for the starting mesh and also for the final stage to give an impression of the dynamic behaviour of the simulations. The meshes can be viewed in the appendix (pages 156-161) where the graph and the partition at the start of the simulation and the final graph after the simulations are displayed.

The underlying PadFEM data structure provides functions for almost all possible needs of FEM simulations. It therefore consumes huge amounts of memory and tends to slow down significantly if the mesh becomes large. For this reason the test cases are chosen not too large.

Square

A unit square was used to provide a very simple domain shape (figures B.5-B.7). The type of refinement applied in this example however is very challenging. In the upper right corner a singularity is simulated such that almost all refinement takes place at this point. During the simulation therefore one subdomain covers most of

		AR	S	N	grading	ar_{max}	ar_{avg}
square	first	1.27	6	115	7.96e+01	3.07	1.90
	last			14839	9.89e+03	3.07	1.90
arrow	first	2.80	16	559	5.85e+02	2.27	1.90
	last			11805	5.33e+00	2.86	1.91
plate	first	2.35	32	1270	2.67e+01	2.18	1.90
	last			13140	1.60e+02	2.86	1.93
shock	first	3.36	32	3302	5.32e+00	2.56	1.74
	last			16396	1.02e+02	2.62	1.92
uk	first	14.58	64	4824	1.43e+06	67.51	1.93
	last			24721	5.99e+04	67.51	1.98

Table 6.1: Tested scenarios

the square and heavy data migration takes place in the remaining 5 subdomains. Almost half of the final elements are generated during the last refinement step where 8820 elements are refined into 14839 triangles.

Arrow

With this example the effect of a severe bottleneck was tested (figures B.8-B.10). At the beginning the larger part of the mesh is still very coarse and on the left hand an arrow-like indentation leaves only a very small connection between the left and the right parts. During the simulation on 16 subdomains the largest 33% of the elements are refined in each step. Since the original mesh is very fine at the arrow mainly the elements on the left hand are refined and thus the subdomains in the left part have to 'move' through the bottleneck to the other side.

Plate

This mesh represents a rectangular metal bar, which is used as a cooling device (figures B.11-B.13) and was partitioned into 32 subdomains. Even though the refinement takes place mainly on four small holes, of all the examples this simulation involves the least subdomain movement. The holes are very small and well positioned, such that the refinement is fairly balanced. The relatively uniform refinement is reflected in the fact that the grading changes less than a factor of 10 whereas all the other scenarios change the grading by two orders of magnitude.

Shock

In dynamic simulations the focus of refinement rarely jumps arbitrarily around the mesh. For example shock waves move in a more or less regular fashion. Even though, for instance, turbulence might show a more chaotic behaviour such a scenario was not included into the test series. To resemble a shock-wave-like behaviour a very regularly triangulated mesh with particularly good element shapes was partitioned into 32 subdomains. It is refined on a moving focus which starts in the lower left corner and is directed diagonally over the mesh (figures B.14-B.16). A circular hole in the middle of the domain adds another complication.

UK

A refinement technique similar to the arrow example was applied on a mesh representing the British mainland (figures B.4, B.17 and B.18). It was chosen for its very intricate boundary – and therefore a high Aspect Ratio – and its high difference in sizes of its triangles. Table 6.1 clearly shows that this scenario has not only a difficult shape but also a very unfavourable triangulation since during the complete simulations element Aspect Ratios up to 67.5 exist. During the tests the largest 12% of all elements are refined in each step. It is the largest of the test examples and was also tested with the highest number of subdomains (64).

6.1.2 Machines

For the sequential timing and all parallel tests presented later in this chapter the PSC Scali Cluster at the PC^2 in Paderborn was used. It consists of 32 machines with 2 Pentium-II-300 processors and 256 Mb of memory each, connected via SCI double LC2 PCI interfaces and Fast Ethernet.

Since PadFEM only compiles using the GNU g++ compiler version 2.8.1 and higher this was the only suitable machine available. Unfortunately most parallel machines do not support this compiler. Therefore only tests up to 64 subdomains could be sensibly made. As discussed later some other difficulties occurred while using this machine.

6.1.3 Test structure

The above described scenarios were extensively tested in a sequential simulation. In chapter 4 different cost functions for assessing subdomain borders and element moves are presented. Obviously not all possible combinations could be tested.

Therefore experience from previous work ([86]) was used to preselect the most promising settings. For each mesh 14832 different tests were made which are evaluated in the following sections. These setting do not vary the optimising function, but previous experiences showed that the chosen function Γ_{ar} usually produces the best results. Section 6.2.5 backs this assumption with additional tests. All initial tests were done without the preshaping step (see section 4.4.1) in order to demonstrate the effects of the individual functions more clearly. The preshaping algorithm is tested in section 6.2.6.

Before parallel tests are analysed in section 6.3 the sequential algorithm is compared to an Aspect Ratio optimising load balancing tool. A parallel load balancer is compared to the new heuristic in section 6.3.4 and finally the convergence of a DD-PCGS applied on final partitions is discussed.

6.1.4 Assessment

The main assessment method naturally measures the resulting Aspect Ratios. Since 10 different values are retrieved (one for each refinement stage) some kind of 'global' assessment is needed. The mesh grows during the simulation and therefore the quality of the refinement is more important the larger the mesh grows. Simply computing the average Aspect Ratio does not consider this fact. Defining N_i the number of elements and AR_i the Aspect Ratio in stage i a 'weighted' Aspect Ratio can be defined as

$$ARW = \frac{\sum_{i=0}^{i<n} AR_i \cdot N_i}{\sum_{i=0}^{i<n} N_i}$$

where n is the number of refinement steps. Since the complexity of the problem rises with the number of elements $AR_i \cdot N_i$ weights the contribution of AR_i with the size of the actual problem. This definition is particularly good due to the independence of the definition of Aspect Ratio from the mesh size. Thus ARW indeed gives a good impression of the partitions during the whole simulation and it is used whenever Aspect Ratios of simulations are presented.

A second important property of a partition is its imbalance. In this thesis for the imbalance a percentage notation is used. It expresses the difference of the largest number of elements in a subdomain N_{max} to the average size N_{avg}:

$$imb = \frac{(N_{max} - N_{avg}) \cdot 100}{N_{avg}}$$

Again a fixed imbalance of a larger mesh might slow down the simulation much more than the same imbalance on a smaller mesh. Therefore the imbalance of simulation is weighted similarly to *ARW* (see above).

Cutsize is sometimes presented as well and it is weighted in the same fashion as the balance and Aspect Ratio. Although cutsize is dependent on the mesh sizes the resulting value can be used as a good comparison of different strategies.

6.2 Sequential tests

6.2.1 Evaluation methods

Four main parameters were tested:

- The flow algorithm – i.e. diffusion or GDE
- The border function – i.e. the function which assigns the border qualities
- The element function – i.e. the function which assigns the element qualities during migration
- The optimisation function – i.e. the function which assigns the element qualities during optimisation

To compare different possible settings two strategies were applied. The first method fixes one (or more) parameter(s) and computes the average resulting AR over the remaining combinations. Often a few combinations of all possibilities for one value of a parameter behave completely differently from the majority of settings. This can easily produce misleading results. Therefore a second subset of tests is generated for each mesh, which includes only the best 100 settings for each example respectively. This aims to exclude those misleading values from the evaluation and at the same time shows which settings perform best, which is the most interesting issue. The presented data usually includes the share of each parameter in these top 100 settings and the best result produced using this method/value.

6.2.2 Border functions

Eight single border functions and seven base combinations were tested:

- Φ_{dist}
- Φ_{avg_dist}
- Φ_{border}
- Φ_{angle}
- Φ_{rel_dist}
- Φ_{rel_avg}
- Φ_{ne}
- Φ_u

- $\Phi_{dist} + \Phi_{border}$
- $\Phi_{dist} + \Phi_{angle}$
- $\Phi_{border} + \Phi_{angle}$
- $\Phi_{rel\_dist} + \Phi_{border}$
- $\Phi_{rel\_avg} + \Phi_{border}$
- $\Phi_{rel\_dist} + \Phi_{border} + \Phi_{angle}$
- $\Phi_{rel\_avg} + \Phi_{border} + \Phi_{angle}$

The weighting factors λ_i (see section 4.2.2) for the combined functions were varied by adjusting λ_{border} and λ_{angle}. Since at most three functions were combined, fixing the factors for the distance functions is not too restrictive. The factors were set to 0.2, 0.6, 1.0, 1.4 and 1.8 and all possible combinations were used where applicable. This makes a total of 103 tested border quality functions.

Table 6.2 lists the average results for all meshes and functions. The last column additionally shows the average Aspect Ratio for each function. The values in brackets represent the *average deviation* δ. It is the average of the individual *deviations* which are defined as $Dev_{mesh} = \sqrt{\frac{AR^{best}_{mesh}}{AR^{f}_{mesh}}}$, where AR^{best}_{mesh} is the best average result of a given scenario (identified by *mesh*) and AR^{f}_{mesh} the average of function f on this example. Using M as the number of test cases the average deviation δ for function f is then computed as follows.

$$\delta_f = \left(\left(\frac{\sum_{i=0}^{i<M} Dev_{mesh_i}}{M} \right)^2 - 1 \right) \cdot 100$$

For example the unweighted strategy (Φ_u) is – in the average – almost 6% (5.77) worse than the weighted case using Φ_{rel_avg} (0.21).

This table suggests that Φ_{rel_avg} is the best choice whilst Φ_{border}, $\Phi_{dist+angle}$, Φ_u, $\Phi_{border+angle}$, Φ_{angle} and Φ_{ne} seem not to be good choices since their average quality is more than 5% worse than the best. However, Table 6.2 is not specific enough to be discussed in more detail because all results of combined functions represent several methods in one. The weighting factors are not considered and therefore a more detailed listing is given in table 6.3 which is only an extract from the complete table A.4 on page 140. The first three columns define the method. The first one specifies the base function of which the weighting factor is set to

	square	arrow	plate	shock	uk	avg (δ)
Φ_{rel_avg}	1.84	2.04	1.89	2.42	2.63	2.17 (0.21)
$\Phi_{rel\_avg+border+angle}$	1.95	2.08	1.92	2.43	2.63	2.20 (2.15)
$\Phi_{rel\_avg+border}$	1.89	2.08	1.92	2.43	2.69	2.20 (1.98)
Φ_{avg_dist}	1.90	2.14	1.97	2.51	2.65	2.24 (3.55)
$\Phi_{rel\_dist+border}$	1.82	2.17	1.97	2.47	2.74	2.24 (3.33)
$\Phi_{rel\_dist+border+angle}$	1.84	2.17	1.98	2.46	2.74	2.24 (3.54)
Φ_{dist}	1.91	2.13	1.97	2.53	2.66	2.24 (3.73)
$\Phi_{dist+border}$	1.97	2.09	1.98	2.52	2.63	2.24 (3.91)
Φ_{rel_dist}	1.86	2.20	1.97	2.43	2.75	2.24 (3.74)
Φ_{border}	1.92	2.17	1.99	2.52	2.76	2.27 (5.10)
$\Phi_{dist+angle}$	1.99	2.16	2.00	2.54	2.70	2.28 (5.71)
Φ_{u}	1.86	2.20	2.06	2.53	2.78	2.29 (5.77)
$\Phi_{border+angle}$	2.01	2.19	2.02	2.53	2.78	2.30 (6.75)
Φ_{angle}	2.01	2.19	2.04	2.55	2.77	2.31 (7.25)
Φ_{ne}	2.02	2.20	2.09	2.52	2.75	2.32 (7.48)

Table 6.2: Average results of all tested border functions

1.0. λ_b and λ_a represent the weighting factors for the Φ_{border} and Φ_{angle} parts. An empty field always means that this part is not included in the function.

This more realistic presentation puts a $\Phi_{rel\_avg+border+angle}$ slightly on top of Φ_{rel_avg} (shown the other way round in table 6.2). The advantage of weighting the subdomain graph is more significant in the more detailed listing: the unweighted average (base function Φ_u) is more than 7% worse than the best weighted one. In particular from the average deviation it can be inferred that Φ_{rel_avg} optionally combined generally outperforms the other functions. In table A.4 the best 26 functions use Φ_{rel_avg} whereas the topmost function not including Φ_{rel_avg} shows a δ that is almost 3 times worse (4.41) than the one for the best function (1.53).

The square example shows deviation values > 2.6 for the four functions that performed best on average. However, the best result for the square was achieved using $\Phi_{rel\_dist+border}$ where the deviations for the other examples are much higher: 7.4%, 4.5%, 3.8% and 6.5%. This special role of the square could be explained by its low number of subdomains combined with a very high rate of refinement, in particular in the last stage. Some disadvantageous subdomain border qualities may destabilise diffusion by building a badly conditioned diffusion matrix. An-

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
rel_avg	0.2	0.6	1.89	2.05	1.89	2.39	2.57	2.16 (1.53)
rel_avg			1.84	2.04	1.89	2.42	2.63	2.17 (1.74)
rel_avg	0.2		1.83	2.04	1.91	2.39	2.65	2.17 (1.77)
rel_dist	0.6		1.81	2.17	1.97	2.44	2.73	2.22 (4.41)
rel_dist	0.6	1	1.81	2.17	1.99	2.45	2.72	2.23 (4.51)
dist	1.8		1.95	2.09	1.98	2.52	2.62	2.23 (5.12)
avg_dist			1.90	2.14	1.97	2.51	2.65	2.24 (5.12)
dist			1.91	2.13	1.97	2.53	2.66	2.24 (5.31)
rel_dist			1.86	2.20	1.97	2.43	2.75	2.24 (5.31)
dist		0.2	2.00	2.15	1.98	2.53	2.64	2.26 (6.40)
	1	0.2	1.93	2.15	2.00	2.48	2.76	2.26 (6.43)
	1		1.92	2.17	1.99	2.52	2.76	2.27 (6.70)
u			1.86	2.20	2.06	2.53	2.78	2.29 (7.37)
		1	2.01	2.19	2.04	2.55	2.77	2.31 (8.88)
ne			2.02	2.20	2.09	2.52	2.75	2.32 (9.10)

Table 6.3: Best average results of weighted border functions

other peculiarity of the square can be observed by looking at the ranges of the individual deviations. They are not given in a table, however they can be inferred from table A.4. Where the values for the other meshes are all $\leq 10\%$ the deviations are steadily rising up to 21.65% for the square. This is another indication of the risk of applying border weighting when the number of subdomains is small.

The values investigated above only reflect a very rough idea of the general behaviour of the border functions. There are still too many unknowns and unfavourable influences which balance the results. Table A.5 on page 143 aims to eliminate any of those effects. To fill this table only the best 100 runs for each scenario were used individually, thus the set of runs for one example may include parameter settings which are not found in another (here the expression parameter setting means a fixed combination of all parameters, i.e. the flow method, the border function, the element function, the optimisation function and the weighting factors). For each function table A.5 extracts from this data the number of settings (%) out of those 100 best runs that used the respective function. Additionally the best result (AR) that was achieved with a single setting including this function is given respectively. Note that values (AR) for the same border function but different meshes might have been achieved using different settings for the remaining parameters. Some of the fields have a zero and a dash which means that no setting including this function is under the best 100 runs for this scenario. The table is sorted by the average number of settings that used each method and by the average Aspect Ratio.

Basically this table does not disclose any conclusive facts. Only uk clearly prefers $\Phi_{rel\_avg+[0.2-0.6]\cdot border+[0.6-1.0]\cdot angle}$ which make 50% of the best 100 runs. Widening the range of λ_{border} to $[0.2 - 1.0]$ and the range of λ_{angle} to $[0.6 - 1.4]$ even increases this share to 74%. The other meshes do not show such a significant preference for particular functions but all of them produce at least 2 excellent results with $\Phi_{rel\_avg+[0.2-0.6]\cdot border+[1.0-1.4]\cdot angle}$. In unison with the previous observations on table A.4 the scaled distance functions produced more and better results in table A.5 than the unscaled versions.

As a summary the results disclose that a carefully chosen border weighting function significantly improves the results and $\Phi_{rel\_avg+border+angle}$ seems to generally outperform the other suggested methods. The scaled distance functions appear on average to work better than the unscaled versions. All presented results suggest that $\Phi_{rel\_avg+[0.2-0.6]\cdot border+[1.0-1.4]\cdot angle}$ as a default setting for the border function promises to produce very good results independent of the example.

6.2.3 Element functions

The element qualities were determined using eight different functions:

- Γ_{rel_dist}
- Γ_{rel_avg}
- $\Gamma_{rel\_dist} + \Gamma_{border}$
- $\Gamma_{rel\_avg} + \Gamma_{border}$
- $\Gamma_{rel\_dist} + \Gamma_{angle}$
- $\Gamma_{rel\_avg} + \Gamma_{angle}$
- $\Gamma_{rel\_dist} + \Gamma_{border} + \Gamma_{angle}$
- $\Gamma_{rel\_avg} + \Gamma_{border} + \Gamma_{angle}$

Similar to section 6.2.2 the weighting factors λ_{border} and λ_{angle} were set to 0.2, 0.6, 1.0, 1.4 and 1.8 and all possible combinations were used where applicable.

	chaos	arrow	plate	shock	uk	avg (δ)
Γ_{rel_avg}	1.76	2.00	1.96	2.38	2.67	2.15 (0.13)
Γ_{rel_dist}	1.84	2.00	1.96	2.37	2.67	2.17 (0.95)
$\Gamma_{rel\_avg+angle}$	1.86	2.06	1.99	2.37	2.71	2.20 (2.39)
$\Gamma_{rel\_dist+angle}$	1.92	2.07	1.99	2.38	2.70	2.21 (3.20)
$\Gamma_{rel\_avg+border+angle}$	1.92	2.16	1.97	2.49	2.71	2.25 (4.97)
$\Gamma_{rel\_dist+border}$	1.92	2.14	1.96	2.54	2.72	2.26 (5.16)
$\Gamma_{rel\_dist+border+angle}$	1.95	2.17	1.97	2.50	2.72	2.26 (5.50)
$\Gamma_{rel\_avg+border}$	1.98	2.15	1.96	2.51	2.72	2.26 (5.57)

Table 6.4: Average results of all tested element functions

Thus 72 different methods calculating the element qualities were tested.

Table 6.4 (which is the equivalent to table 6.2) shows one peculiarity: the average results of the plate example are all extremely similar. Its deviations (which are to be inferred from table 6.4) are all $< 6\%$ whereas the largest differences occur again for the square example (up to 17.8%). In particular the good result for Γ_{rel_avg} is due to the outstandingly good average Aspect Ratio for the square. Although (similar to section 6.2.2) this table shows rather unspecific data the general superiority of the normalised distance function Γ_{rel_avg} to its unscaled counterpart Γ_{rel_dist} can be observed.

Table A.6 shows a similar listing including the weighting factors. The most significant fact which can be extracted from this table is that methods that differ only in their weighting factors can produce very different results. For example $\Gamma_{rel\_avg+0.2\cdot border+0.6\cdot angle}$ gets the best average result (2.12 (0.65)) whilst $\Gamma_{rel\_avg+1.8\cdot border+0.2\cdot angle}$ behaves very poorly (2.35 (11.48)) and is ranked almost at the bottom of the table. It can be observed that all combinations using Γ_{border} appear to be almost sorted by λ_{border}. This leads to two major conclusions: Firstly the task of selecting elements is much more sensitive to the weighting factors than using the functions for assessing the border quality. Certainly this is not particularly surprising since the border functions only represent an average value over many elements whereas the element functions are applied on individual elements and actually decide the Aspect Ratio development by selecting the migrating elements. Secondly adding Γ_{border} with a $\lambda_{border} < 1.0$ in general significantly improves Γ_{rel_avg} and Γ_{rel_dist}.

Again the effect of an unfavourable function appears to be much stronger for the square than for any other mesh whereas the difference is much less for the plate example. The reason for this is that in the square scenario much more data is migrated and thus the element function is used more heavily. The plate example does not include much element migration and therefore most of the time the algorithm is in optimisation mode where the optimisation function is active (in contrast to the migration phase where the element function is used, see section 4.2).

Table A.7 in the appendix (which is similar to table A.4, see section 6.2.2) gives a very similar picture. The top of both tables (tables A.7 and A.6) list almost the same functions. This means that the same functions that show good results whatever the other parameters are set to also produce the absolute best results. However, Γ_{rel_avg} tends to occur slightly more frequently in table A.7 and to produce slightly better results than its unscaled counterpart Γ_{rel_dist}.

At first glance the observations made above on table A.6 might contradict the fact that functions using Γ_{border} with $\lambda_{border} > 1.0$ appear in the top settings at all: for the plate these methods make even 6% of the top 100 runs. On second thoughts this becomes clearer: most of these candidates were used on the plate example where less element migration takes place. As mentioned above this makes the element function less important and is more dependent on the optimisation function. This explanation is backed with the fact that for square and uk there is only one occurrence each and for the other two scenarios no function using $\lambda_{border} > 1.0$ achieved a top result.

The tests most significantly suggest that both relative distance functions can be improved by adding Γ_{border} with $\lambda_{border} \leq 0.6$. The addition of Γ_{angle} often improves the combined method further. Although the results for λ_{angle} are not as clear as for λ_{border}, setting $\lambda_{angle} \geq \lambda_{border}$ seems to make a safe choice. Finally the more data migration occurs, the more carefully the element function must be selected.

6.2.4 Flow methods

The tests showed that there is no significant difference between the two weighted diffusive methods (see tables A.10 and A.11). On average GDE and conventional diffusion almost equally share the top 100 runs, their best results are almost identical and the average produced Aspect Ratio as well as their average deviation is very similar. Also combining the flow method with border functions does not disclose any significant differences, thus those tables are omitted. Only the comparison of individual runs show a slight better performance of diffusion. Even though in theory GDE tends to converge faster, diffusion might therefore be preferred if run times are not the most important factor.

6.2.5 Optimisation functions

When in section 4.2 Algorithm 4.2 calls in lines 3 and 16 `enable_optimising()`, the function assessing the element qualities is set to the selected optimisation function. As the default Γ_{ar} was used in the previous tests. To back experimental experience additional tests were performed varying this function. Eight candidates were chosen:

- Γ_{ar}
- Γ_{border}
- Γ_{rel_dist}
- Γ_{rel_avg}
- $\Gamma_{rel\_dist} + \Gamma_{border}$
- $\Gamma_{rel\_avg} + \Gamma_{border}$
- $\Gamma_{rel\_dist} + \Gamma_{border} + \Gamma_{angle}$
- $\Gamma_{rel\_avg} + \Gamma_{border} + \Gamma_{angle}$

They were combined with a set of parameter settings (i.e. combinations of flow method, border function and element function) that was obtained by adding the best 300 settings of each scenario. If no parameter combination performed well enough to be within the best 300 settings for more than one scenario this set would contain 1500 combinations. However since some settings performed well on more than one example this set contains 1101 different parameter settings (which means that at most 399 combinations are among the best 300 settings for at least two scenarios). Combined with the above 8 optimisation functions this made a total of 8808 tests per mesh.

The average deviations in table A.8 on page 149 shows clearly that using Γ_{ar} is the most stable function. A value $\delta < 1$ clearly outperforms the second best function $\Gamma_{rel\_avg+0.2\cdot border+0.2\cdot angle}$ with an average deviation of almost 4% (four times worse). The bottom part of the table demonstrates that a bad optimisation function cannot do its job properly. Even though in section 2.2.4 a similarity

	square	arrow	plate	shock	uk	avg.
average	0.96	0.99	0.96	0.83	0.96	0.95
best	0.93	0.99	0.95	0.83	0.96	0.93
worst	1.13	0.83	0.98	0.51	0.96	0.88

Table 6.5: Results of the preshaping step

between Γ_{ar} and Γ_{border} had been supposed, table A.8 cannot back this equivalence. However, although Γ_{border} shows an average deviation of almost 7%, it is the second most frequent optimisation function appearing in the best 100 tests. Table A.9 again lists the share of each function and the best results for each mesh. It discloses that Γ_{ar} and Γ_{border} make more than 60% of the top 100 runs which they almost equally share. This suggests that Γ_{border} works reasonably well if it is assisted by parameters.

The results presented in tables A.8 and A.9 back previous experimental experience that Γ_{ar} is the clearly preferred function in the optimisation phase. Of course this is not particularly surprising since it directly addresses the global cost function the algorithm tries to optimise.

6.2.6 Preprocessing

To investigate the effect of the preprocessing step a similar set of parameter settings as used in section 6.2.5 was tested on the preshaping algorithm (see section 4.4.1). This time a larger set could be used and hence the best 2000 parameters of each scenario were put together and made a total of 7438 different parameter combinations.

In table 6.5 for each mesh three values are listed. The first one is the average Aspect Ratio produced by the heuristic using the preprocessing step divided by the average results without the preshaping (of course the same set of parameter settings was used). The second value is the best result using the preprocessing over the outcome without preshaping. Similarly, the last line shows the same ratio for the worst results. The last column gives the average for each row.

The table shows clearly the positive effect of the preshaping algorithm. Both the average and the top results are always improved from 1% up to 17%. Only the worst result for the square shows a negative effect. However, the average improvement for the worst settings is much better than for the first two rows. An

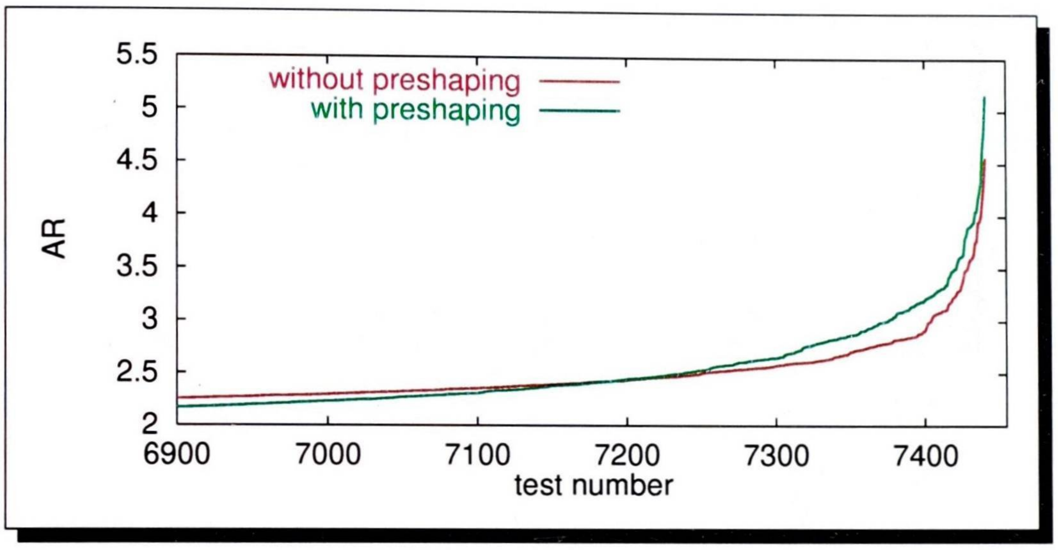


Figure 6.1: The worst runs with and without preshaping

average improvement of 12% and a top value of 49% suggests that the 13% worsening is one of the exceptional values that easily occur for discrete optimisation problems. This might be the case, nevertheless figure 6.1 shows that the worst 3.3% of the runs without preshaping produced better results than the worst 3.3% of the test including the preprocessing. In this diagram the x-axis shows the rank of the runs which are sorted by their resulting AR, and the y-axis shows their AR. However there are cases for which the preshaping algorithm has a negative effect, the difference decreases to less than 8% after the last three runs and then the preprocessing version quickly turns out to be superior. Another explanation for this might be again the low number of subdomains used for the square scenario. If the subdomain which is currently preshaped has only few neighbours these adjacent subdomains can quickly get heavily overloaded which causes a high imbalance. As discussed earlier heavy element migration causes severe problems which in turn can destroy the positive effect of the preprocessing step.

Generally the preshaping step improves the heuristic significantly. Not only the top results are improved but it also stabilises the algorithm which is reflected in an average improvement of 5% and in an average improvement of the worst setting of 12%.

6.2.7 Complete parameter settings

In the above discussion extensive tests of a sequential simulation were used to extract good parameter settings. The analysis of a very large number of tests suggest the use of $\Phi_{rel\_avg+border+angle}$ as the function to weight the subdomain edges for

S	square	arrow	plate	shock	uk	avg.
4	1.70	1.76	2.02	2.49	6.02	2.80
8	2.06	2.07	1.98	2.44	5.04	2.72
16	1.84	1.88	1.99	2.25	3.34	2.26
32	2.05	2.06	2.03	2.17	3.03	2.27
64	2.42	2.03	1.92	2.41	2.51	2.26

Table 6.6: Results on different numbers of subdomains

diffusive methods. Values < 1 for λ_{border} promise good results, optionally combined with a $\lambda_{border} < \lambda_{angle} \leq 1.4$. For selecting elements $\Gamma_{rel\_avg+border+angle}$ seems to be the best choice. The results suggest that λ_{border} should be set to 0.2 and $\lambda_{angle} \leq 1$ promises to work well. The diffusive methods work equally however diffusion seems to perform slightly better. The preprocessing should clearly be used and Γ_{ar} proved to work best as the optimisation function.

From these observations it could be inferred that a parameter setting conforming all the above settings should – on average – perform best. Indeed the settings which can be derived from the above conclusions produce the best results. Computing the average Aspect Ratio for all test cases, using complete parameter settings (flow,border function, element function, optimisation function) most of the best 30 settings are variations of the proposed values. The best average result was achieved by using diffusion, $\Phi_{rel\_avg+0.2\cdot border+0.2\cdot angle}$, $\Gamma_{rel\_avg+0.2\cdot border+0.2\cdot angle}$ and Γ_{ar} as optimisation function with a $\delta = 5.68$. As an absolute value this might be less impressive but the relation to the worst deviation of 87.11 gives a much clearer picture. Note that the deviation of 87.11 is the worst only out of the set with the top 100 settings per scenario and thus far worse settings were tested.

6.2.8 Number of subdomains

The test cases where also tested on different numbers of subdomains. For each scenario and number of subdomains the 250 parameter settings where used which on average produced the best results. Table 6.6 summarises the results by listing the average results.

Only the square and the uk example show a significant correlation between the number of subdomains and the resulting average Aspect Ratios. The new heuristic has difficulties when the number of subdomains rises on the square whereas it favours a great number of subdomains on the uk example. The latter can be explained by the intricate shape: with a high number of subdomains the algorithm can place entire subdomains within problematic areas of the domains such that the coarse structure of the resulting subdomains is almost convex. If the number of elements per subdomain exceeds the number of elements in such areas there is no possibility to generate more or less convex subdomains, which results in bad Aspect Ratios.

The difficulties with the square example could be explained with the generally demanding problem this example represents. Since the refinement takes place mainly at a single small point only a few subdomains will cover most of the global area. This imbalance is hard to deal with, in particular since in every refinement stage large load imbalances forced huge amounts of data migration. This can easily lead to poor subdomain shapes.

The algorithm performs generally stable on variable numbers of subdomains. Even though two examples seem to react sensitively on this parameter, this observation is more likely to be due to the nature of the examples themselves to be caused by the function of the algorithm.

6.2.9 Comparison with Jostle

The results of the new heuristic were compared with those produced by the (re)partitioning and load balancing tool Jostle, [98] (see section 3.2.2). This is the only state-of-the-art load balancer which has an option for optimising Aspect Ratio (although the Aspect Ratio optimising version has not been previously tested for dynamic repartitioning). It was linked to the PadFEM project and Jostle was tested on the same scenarios introduced above. To achieve comparable results the following parameter settings were used:

parameter	value
data	partitioned
matching	local
threshold	10
imbalance	0

Table 6.7 compares Jostle with the new heuristic. For each scenario and balancer the achieved Aspect Ratio and final imbalance of the best runs are given. Additionally the total number of element swaps during the optimisation and the time in seconds the balancing needed throughout the entire simulation are

		square	arrow	plate	shock	uk
Aspect Ratio	PadFEM	1.39	1.68	1.62	1.96	2.13
	Jostle	1.76	1.79	1.78	2.05	2.12
% imbalance	PadFEM	0.2	0.6	1.4	0.9	1.3
	Jostle	0.2	1.2	2.4	1.0	1.8
no migrations	PadFEM	13556	21756	7957	65756	33685
	Jostle	19725	37076	18824	204903	68917
time in sec.	PadFEM	8.66	12.26	10.25	36.32	46.46
	Jostle	2.74	3.31	3.13	9.84	15.13

Table 6.7: Comparison with Jostle

listed. Even though Jostle executes 3-4 times faster the quality of the partitions are usually worse. The Aspect Ratios generated by the new heuristic can be up to almost 27% better than the results by Jostle. In particular the square example with heavily imbalanced refinement shows the most significant difference (26.6%). Although Jostle achieved an equally good result for uk an average of over 9% worse Aspect Ratios shows the strength of the new method.

Both heuristics achieve good a balances (see section 6.1.4 for the definition of imbalance), however PadFEM tends to balance slightly better and its produced imbalance always stays $< 1.5\%$ whereas Aspect Ratio optimising Jostle produces imbalances up to 2.4%.

The times for Jostle include the time needed for generating the data structures used by the external software as well as the time for the actual migration of objects. This approach reflects a more realistic comparison, since usually data structures that are used for numerical simulations are not suitable for such high performance balancing as the ones by specialised libraries. Furthermore the transformation is needed anyway if Jostle is to be used and the data needs to be migrated.

The very good performance of Jostle is due to the implementational reasons previously described in section 5.2.5. The total number of element swaps was included to prove that the new heuristic could be implemented with a comparable performance. The number shows all migrations that were done during the optimisation of Jostle and PadFEM. It usually is more than an order of magnitude greater than the minimal number of migrations needed to actually achieve the computed partition. As a library call Jostle internally does not need to actually

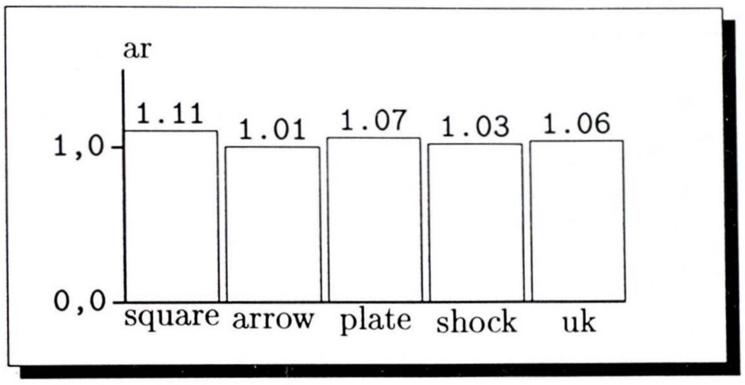


Figure 6.2: Results of parallel runs

migrate all data attached to the elements, hence one of those swaps takes significantly less time than one swap performed inside PadFEM (where each migration includes all element data and additionally its vertices, edges and neighbouring elements). Table 6.7 shows that Jostle needs 45-211% more swaps than PadFEM for its optimisation. An average of 114% suggests that the new algorithm needs (internally) less than half the number of element swaps for its optimisation than Jostle and at the same time produces results that are on average more than 9% better. From this it can be inferred that the new heuristic is more effective and it does furthermore not include expensive computations needed for graph reduction/expansion (although this is not that simple since the multi-level approach might save work by clustering elements). Thus the fast execution of Jostle seems to be achievable with a careful implementation that replaces several physical swaps by virtual migrations.

The results presented in this section clearly show that the new heuristic is competitive to a state-of-the-art load balancer. Although Jostle might produce better results if the parameters were chosen more carefully the general conclusion would certainly remain.

6.3 Parallel tests

6.3.1 Quality

Each scenario was tested in parallel for 30 different parameter settings which were the most promising chosen from the results of the sequential simulations. Figure 6.2 shows for each mesh the average Aspect Ratio of the top 100 runs in sequential divided by the best parallel result. The diagram shows that the parallel version never achieves as good results as the sequential simulations promised.

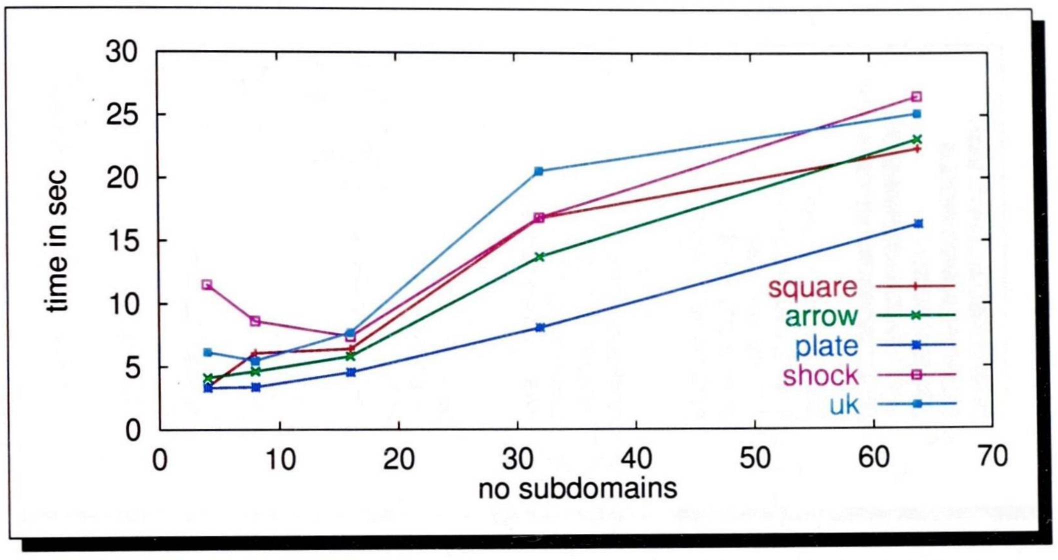


Figure 6.3: Time versus number of subdomains

Although the base algorithm is on average 5.5% worse the results for square, arrow and shock are still better than the sequential multi-level software Jostle (see table 6.7). In particular for the square, which shows the largest deviation of 11%, the new heuristic performed still 2.4% better than Jostle. Since also the shock example is treated 1% better than by Jostle it is assumed that the new heuristic performs particularly well in scenarios where the subdomain shapes have to change frequently.

Certainly the difference between the two versions of the new heuristic is caused by the advantage of global data locality in sequential. Collisions cannot occur and the optimisation has much greater freedom since a 'complete halo' is provided. The parallel version might use the entire halo without achieving comparable changes (see also [14]). This general problem of a parallel execution is also reflected in other software tools. ParMetis [61] for example seems to have similar difficulties (see section 6.3.4).

6.3.2 Scalability

To investigate the scalability of the algorithm the performance was timed on the above 5 examples. Each scenario was decomposed into 4, 8, 16, 32 and 64 subdomains and the usual simulations performed over 10 stages each. Unfortunately the program could not be compiled using the implementation of MPI [72] taking advantage of the SCI interfaces (SCAMPI [84]). Thus PVM [92] over Fast Ethernet was used which is significantly slower than SCAMPI. Furthermore PVM did

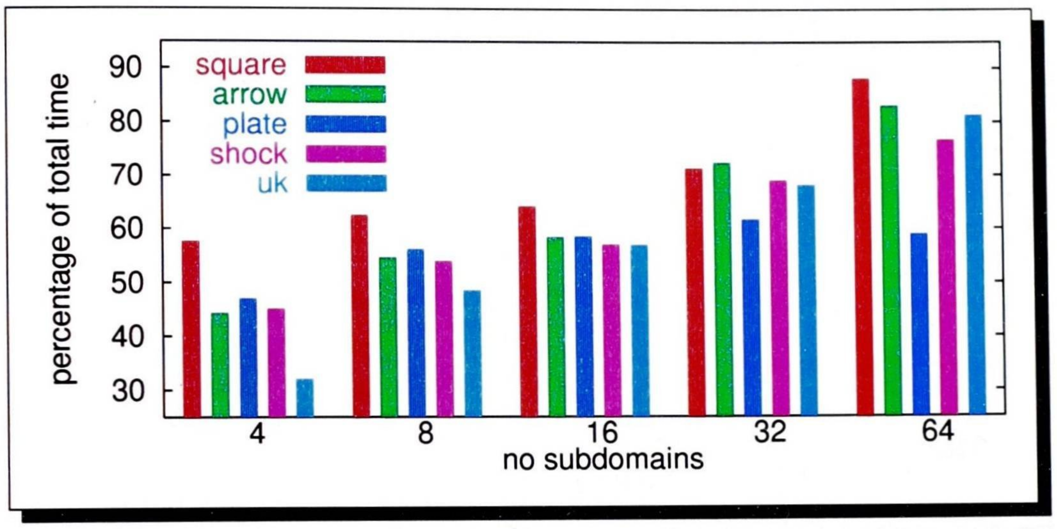


Figure 6.4: Percentage of diffusion and communication times

not distribute the processes evenly over the provided processors. Frequently one dual processor machine was allocated three processes and another one got only one. Therefore the results would most probably be better in a more optimised environment.

Figure 6.3 shows the time used for the load balancing for each mesh and number of subdomains. The values represent average times of 5 different runs each. It can be observed that the program does not scale well. When the number of processors increases the algorithm slows down. Even though the shock example shows a gain in speed up to 16 processors the time increases on average almost linearly with the number of subdomains. This is not surprising since the complexity of the diffusion algorithm is dependent on the number of subdomains and it is called frequently.

More detailed analysis discloses that the fraction of time needed by communication, additional data consistency enforcement and flow calculation rises with the number of subdomains. Figure 6.4 shows the percentage of communication and diffusion of the complete run times. Usually the flow calculation takes around 80-90% of the time the communication needs and the time needed to keep the data about vertices and edges up-to-date only 1-2%. The time for those updates was included as well since a dedicated implementation would not need to consider vertices and edges at all. For 64 subdomains the program is clearly dominated by diffusion and communication which take on average more than 77% of the run time whereas for 4 subdomains only 45% of the time are spent in these parts. Only the plate example shows a less steep increase which could be explained with

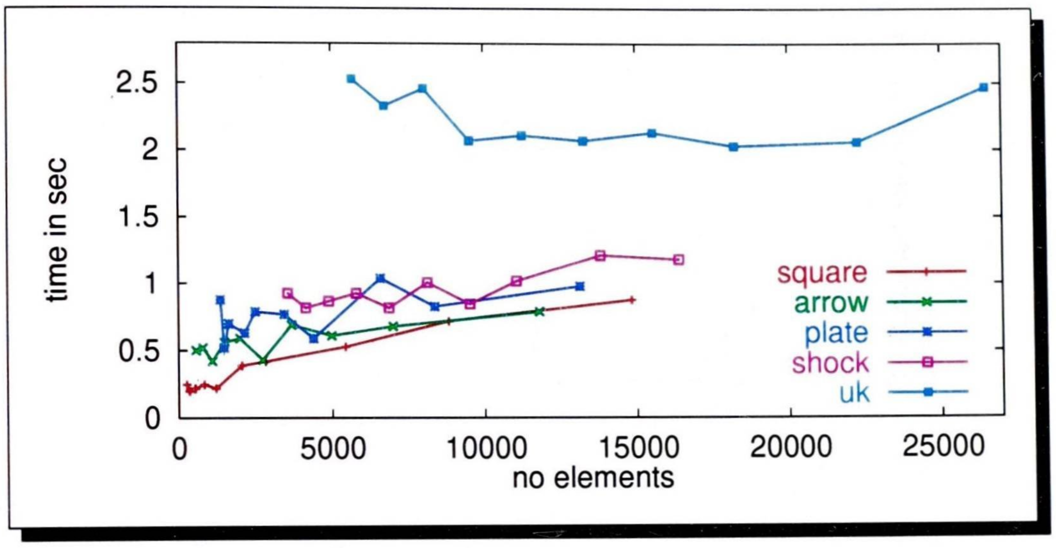


Figure 6.5: Time versus number of elements

its relative uncomplicated nature.

One explanation for the bad scalability of the communication could be its implementation. The migration of objects is provided by the PadFEM data structure and needs for every migration step one synchronisation point and a global all-to-all communication. In this communication step all subdomain pairs exchange messages to migrate objects even if the two subdomains are not adjacent. Since migration is only allowed between neighbouring subdomains the majority of these messages are not necessary. During the optimisation many of those migration steps are performed and thus the time needed for communication increases with the number of subdomains and not – as it should be – with the cardinality of the subdomain graph. A dedicated communication implementation that omits those redundant communications should scale a lot better.

One way of reducing the effect of the diffusion algorithm is implemented in ParMetis [61], which calculates the flow only once. The negative aspect of this approach is that it makes it more difficult to achieve perfect balance. This problem is discussed in section 5.2.2 and is also reflected by the imbalances generated by ParMetis in section 6.3.4.

Figure 6.5 shows average times the parallel executions needed for each scenario and stage. The x-axis shows the the number of elements in each stage and on the y-axis the time is given. It can be observed that only the square example shows a significant increase in time when the number of elements rises. Although the other examples also need more time on larger meshes the increase is much less.

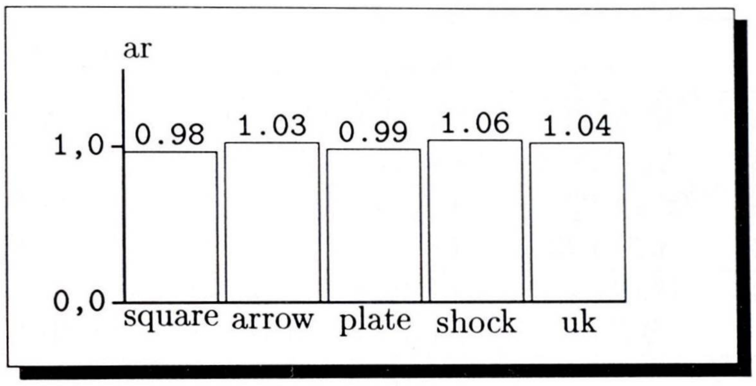


Figure 6.6: Results of AMPD

Sometimes the time even decreases when the number of elements increases. In particular the uk shows on average to consume almost constant time for each level. Only the last stage shows a significant increase in time which can be explained with the particular high number of new elements. If many elements are generated during the refinement high imbalances can occur and therefore more time might be needed. This behaviour can also be observed for the square example where the number of elements and the times synchronously increase significantly only from step 7 onwards.

The discussions in this section suggest that the algorithm scales very well with the mesh size but does not take advantage of an increasing number of processors. However the latter could possibly be relaxed with a careful implementation this is a general problem of parallel load balancing strategies, since the number of subdomains play a major role in this task and load balancing is a very communication-intensive application.

6.3.3 Asynchronous Multiple Phase Diffusion

Asynchronous Multiple Phase Diffusion was tested on all examples. Unfortunately figure 6.6, which shows the best results of AMPD divided by the best results of the normal algorithm in parallel, does not reflect the intended effect. Although AMPD performed 2% better on the square it produced worst results on shock. This is particularly surprising and disappointing since both examples are the ones which resemble the type of simulation best for which this method was devised. However, the plate also shows slightly better a result than the conventional strategy. On average AMPD performs 2% worse which is still acceptable accounting for the experimental state of this method.

		square	arrow	plate	shock
ar	ParMetis	2.02/2.27	3.21/4.21	6.08/6.78	6.46/7.33
	PadFEM	1.68/1.65	1.77/1.83	1.82/1.79	2.18/2.23
cut	ParMetis	69.4/77.1	49.9/60.2	59.52/62.20	79.46/85.75
	PadFEM	71.8/71.8	35.7/36.5	33.9/35.8	46.5/44.1
imb.	ParMetis	13.8/5.8	41.0/19.5	5.54/10.21	4.98/5.05
	PadFEM	0.2/0.1	11.8/12.4	1.6/1.6	2.6/1.3
time	ParMetis	4.0/4.6	8.4/11.8	–	–
	PadFEM	3.0/4.4	6.33/18.7	–	–

Table 6.8: Comparison with ParMetis

The diffusive methods tend to consume most of the time to balance the last 2 or 3 percent of the imbalance. Therefore the first tests showed run times which where 10-100 times slower than the tests on the original algorithm. Allowing an imbalance tolerance of 5% decreased the run time by a factor of 10 to 50. This improvement sometimes made AMPD even faster than its counterpart, but in general the times needed by AMPD vary a lot.

Even though the above results do not demonstrate an improvement, tests showed that on the square example AMPD performs generally better. Further research could perhaps find more suitable settings for the cost functions that cooperate better with the needs of AMPD. Furthermore a detailed analysis of the thresholds that trigger migration might help to meet the expectations in this experimental algorithm.

6.3.4 Comparison with ParMetis

The partitioning/load balancing software ParMetis [61] which is freely available for academic purposes was tested on the same scenarios. As a comparison the two diffusive methods (global and local diffusion, see section 3.2.3) provided by ParMetis were tested and are shown shown in table 6.8. For an comparison with the new heuristic the results for the normal method and AMPD are given as well. Unfortunately it was not possible to test the uk example with ParMetis, since the code crashed. It is not known whether this is a problem with the data supplied to ParMetis, the machines it was tested on or with ParMetis itself.

The table shows at the top that the diffusion based algorithms implemented in ParMetis produced Aspect Ratios that are almost 4 times worse than the ones produced by the new heuristic. Since ParMetis optimises cutsize this difference is not very surprising. The next two rows list the cutsize for each scenario which was weighted in a similar fashion as Aspect Ratio. These values for cutsize show that PadFEM surprisingly generated partitions with lower or only slightly worse cutsize than ParMetis. An explanation for this could be that optimising AR keeps the subdomain shapes compact and thus it is unlikely that during dynamic changes the subdomains become too degenerated or disconnected.

The values for the imbalance show an average weighted imbalance per mesh. The table shows that ParMetis produced relatively poor balance (although the scratch/remapping option, which was not tested, might have produced better results). Comparing the times discloses that the new heuristic even performed slightly faster than ParMetis, despite the problems mentioned in section 5.2.5. The run times for plate and shock are not listed since for 32 subdomains an external program had to be used with files as the interface. Note that ParMetis was tested using MPI and the new heuristic using PVM, what should give another advantage to ParMetis. Certainly ParMetis itself is faster without including the transformation and migration work. But this is an overhead any application using ParMetis has to consider – if not using an integrated data structure like PadFEM.

As a summary the new parallel heuristic shows excellent performance compared to a state-of-the-art parallel diffusion based load balancer:

- far better Aspect Ratio
- lower cutsize
- better balance
- comparable run times

The difference between the sequential and parallel versions suggest that even better performance might be possible if further research were made on the new method.

6.4 PadFEM and ParMetis on a preconditioned solver

To demonstrate that on more realistic examples good subdomain shapes improve the convergence of a domain decomposition based preconditioned conjugate gradient solver (DD-PCGS), the same solver (PETSC, CG, [5]) and preconditioner (ParPre, DD, [35]) as used in in section 2.2.5 were applied to the square, arrow,

	square	arrow	plate	shock
PadFEM	1516	1574	3489	5563
ParMetis	2016	3663	9542	8623

Table 6.9: No. iterations of a DD-PCGS (PadFEM vs. ParMetis)

plate and shock examples. Again a simple heat distribution problem was applied and tested on the last level of each scenario (after 10 levels of refinement and load balancing). This is a meaningful test because in these tests (and frequently in real-world simulations) the final mesh is the largest and hence the one which dominates the solution time. The solver was tested using the final partitions which were achieved from the adaptive runs with the new heuristic and ParMetis respectively.

Table 6.9 shows the number of iterations the solver needed to converge on each scenario after load balancing with PadFEM or ParMetis. Conforming with the better Aspect Ratios in table 6.8, in all examples the solver needed significantly less iterations on the partitions generated by the new heuristic. On average the solver needed almost twice as many (1.99) iterations for ParMetis' decomposition.

These figures back the results from section 2.2.5 and suggest that indeed the subdomain Aspect Ratio should be optimised for this type of solver. Together with the results of section 6.3.4 the tests furthermore underline that the conventional criterion of minimising the number of cut edges often does not imply good subdomain shapes and thus an explicit shape optimising should be preferred to achieve good Aspect Ratios.

6.5 Summary

In this chapter extensive tests of the heuristic described in chapter 4 were presented and analysed. In section 6.2 sequential simulations were used to extract good parameters, different values were analysed individually (sections 6.2.2, 6.2.3, 6.2.4, 6.2.5 and 6.2.6) and in section 6.2.7 their suggested use was confirmed with comparisons of combined settings.

In comparison to another Aspect Ratio optimising load balancing tool, Jostle [98], the sequential version of PadFEM produced partitions with better Aspect Ratios. Although Jostle executed faster, the low number of global migrations which the new heuristic needs during optimisation suggests (section 6.2.9) that a dedicated implementation might perform similarly well.

In section 6.3 the parallel version of the new heuristic was tested. The analysis of the tests disclose (section 6.3.1) that the algorithm does not perform equally well in sequential and in parallel. However for examples which are considered to be particularly difficult, comparable results were achieved. Although the scalability of the program with the number of processes stays below expectations, in section 6.3.2 the heuristic appears to be almost independent of the number of elements.

Despite these problems with the parallel version, the new heuristic demonstrates (section 6.3.4) an excellent performance in comparison to a parallel state-of-the-art diffusion based load balancer provided by ParMetis [61]. Even though the selected algorithms of ParMetis were not faster nor produced lower cutsizes, the new heuristic generated partitions with much better Aspect Ratio and significantly lower imbalance. These impressive results of the load balancer developed in this work are also reflected in the test on a Domain Decomposition Preconditioned Conjugate Gradient Solver (section 6.4). In all tested examples the solver needed significantly less iterations to converge on the partitions generated by the new heuristic than on the partitions produced by its opponent.

Chapter 7

Conclusion and future directions

This thesis was motivated by the need for a load balancer with a new objective. Modern parallel solvers often react more sensitively to the shapes of the subdomains than to their cutsize which is the optimisation criterion of conventional partitioning and load balancing. A new load balancing heuristic that directly addresses the optimisation of subdomain shapes was derived and implemented. Different ways to assess subdomain shapes that were suggested in the literature were presented and compared with a new and more consistent definition of Aspect Ratio.

The latest results of research in the field of load balancing were applied and additional strategies were developed to meet the requirements of the new objective. The well known diffusive method and the Generalised Dimension Exchange algorithm were enhanced by weighting the subdomain edges, which improved the results significantly. Cost functions were applied that were recently suggested in literature to select suitable elements to be migrated. Additionally new geometric functions were developed that help the heuristic to produce and preserve good subdomain Aspect Ratios. Finally comprehensive tests were made to extract the best parameters. This was successfully achieved for the element as well as the border and optimisation functions. Although no single setting could be suggested that always produces the best results, a generally stable and good combination was found.

The powerful flow algorithms combined with a simple optimisation strategy and the use of sophisticated optimisation functions made it possible to achieve better results than state-of-the-art load balancing tools. Although no complicated computations such as graph reduction, relative gain or extra optimisation steps were included into the heuristic, it avoids degenerated subdomain shapes. In comparison to a conventional parallel cutsize optimising load balancer the new parallel heuristic proved to be superior in terms of all major aspects. It generates partitions with much better Aspect Ratios, better balance, lower cutsize and seems even to run faster. The good results for the number of cut edges can be explained with its capability to keep the subdomains very compact.

Parts of this work appeared in the proceedings of the EUROPAR '98 [27], IRREGULAR '98 [31], ALV '98 [30] and the 3rd Euro Conference on Parallel & Distributed Computing for Computational Mechanics [87]. Others appeared in the International Journal for Supercomputer Applications [102] and were accepted for publication in the Journal for Parallel Computing [28].

In possible future work the heuristic should be enhanced to 3 dimensions and its practicability for 3D should be investigated. To be more flexible the heuristic could be extended to deal with weighted meshes as well, e.g. to be applied on hierarchical meshes. Further research should be done on Asynchronous Multiple Phase Diffusion which did not meet the expectations due to its experimental state but its excellent results on one of the test examples suggests the possibility of improvements. Also the effect of halos of different sizes and a control over the halo sizes give opportunities for more investigation. Due to the implementation of PadFEM the efficiency of the new heuristic could not be examined properly, thus a dedicated implementation of the proposed ideas would be necessary. Furthermore the multi-level idea might be helpful to improve the heuristic. Another field for further research might be to transfer the idea of weighting the subdomain edges to related algorithms [56, 70]. Finally the objective of generating/preserving subdomains with good Aspect Ratios could be generalised to generating/preserving subdomains with particular shapes or geometries.

Acknowledgments

First of all I would like to thank my supervisor C. Walshaw for his constructive suggestions, invaluable support, helpful discussions and frequent motivation. Without his help I would not have achieved this piece of work. The Parallel Processing Group at the University of Greenwich is thanked for providing a pleasant working environment. I would also like to acknowledge the financial support of the School of Computing and Mathematical Sciences, in particular M. Cross, which made this work possible. The work was also supported by the German Research Foundation (DFG-SFB 673, "Massive Parallelität", TP-A3) in the beginning of my research. My colleagues in Paderborn working on the PadFEM project are thanked as well for assisting me, in particular R. Diekmann who contributed a lot to this work by his valuable support and helpful discussions. I would also like to acknowledge the technical support of the PC^2 and V. Eijkhout's spontaneous help on ParPre. Finally I would like to thank my friends and family who suffered from my work on this thesis.

List of Figures

List of Tables

List of Algorithms

Appendix A

Tables

		t60k		uk		whitaker		crack	
		AR	Its	AR	Its	AR	Its	AR	Its
ar	AR_0	1.76	2509	3.56	406	1.60	1061	1.55	1570
	AR_1	3.05		1.45		1.89		253.44	
	AR_2	2.75		1.12		1.50		12.25	
	AR_3	3.98		6.31		2.49		2.12	
	AR_4	483.08		9.90		727.38		45.93	
	AR_5	2.32		2.58		2.13		1.76	
	cut	569		109		380		475	
cut	AR_0	1.79	2635	3.81	571	1.68	877	1.79	1804
	AR_1	2.95		2.23		2.82		7.09	
	AR_2	2.62		2.00		2.50		3.38	
	AR_3	2.54		8.47		2.49		2.54	
	AR_4	760.80		14.52		310.47		52.35	
	AR_5	2.06		2.69		1.95		1.98	
	cut	539		99		383		329	

Table A.1: Iterations and ARs/Cutsize for a DD-PC-CG (8 subdomains)

		t60k AR	t60k Its	uk AR	uk Its	whitaker AR	whitaker Its	crack AR	crack Its
ar	AR_0	1.79	2376	2.73	814	1.73	1583	1.68	–
	AR_1	2.40		2.38		4.61		14.04	
	AR_2	2.06		1.75		4.12		3.56	
	AR_3	3.19		4.33		2.77		2.63	
	AR_4	751.85		14.73		172.16		4.86	
	AR_5	2.30		2.47		2.14		1.97	
	cut	1031		197		662		742	
cut	AR_0	1.92	2907	2.90	873	1.70	1424	1.90	–
	AR_1	4.94		4.60		2.00		11.87	
	AR_2	4.25		4.19		1.50		6.56	
	AR_3	2.87		4.10		3.19		3.15	
	AR_4	463.75		10.62		96.52		6.01	
	AR_5	2.24		2.44		1.94		2.07	
	cut	1016		182		629		555	

Table A.2: Iterations and ARs/cutsize for a DD-PC-CG (16 subdomains)

		t60k AR	t60k Its	uk AR	uk Its	whitaker AR	whitaker Its	crack AR	crack Its
ar	AR_0	1.65	4451	2.20	1029	1.75	2008	1.65	3931
	AR_1	4.83		3.39		3.49		6.80	
	AR_2	4.28		2.62		3.12		3.34	
	AR_3	2.88		3.66		3.16		2.44	
	AR_4	273.13		10.57		48.71		35.08	
	AR_5	2.15		2.15		2.15		1.94	
	cut	1607		332		1029		1042	
cut	AR_0	1.72	5481	2.47	1303	1.71	2216	1.78	4166
	AR_1	2.94		3.08		4.57		5.67	
	AR_2	2.41		2.81		4.16		3.22	
	AR_3	2.40		3.94		2.42		2.51	
	AR_4	171.07		13.37		57.73		13.65	
	AR_5	1.92		2.36		1.89		2.01	
	cut	1552		305		971		823	

Table A.3: Iterations and ARs/cutsize for a DD-PC-CG (32 subdomains)

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
rel_avg	0.2	0.6	1.89	2.05	1.89	2.39	2.57	2.16 (1.53)
rel_avg	0.2	0.2	1.83	2.02	1.90	2.43	2.62	2.16 (1.40)
rel_avg			1.84	2.04	1.89	2.42	2.63	2.17 (1.74)
rel_avg	0.2		1.83	2.04	1.91	2.39	2.65	2.17 (1.77)
rel_avg	0.2	1.4	1.96	2.04	1.90	2.42	2.57	2.18 (2.53)
rel_avg	0.2	1	1.95	2.04	1.89	2.44	2.57	2.18 (2.61)
rel_avg	0.6	0.2	1.86	2.04	1.91	2.44	2.66	2.18 (2.52)
rel_avg	1	1.4	1.96	2.06	1.90	2.41	2.59	2.18 (2.89)
rel_avg	0.6	0.6	1.93	2.07	1.90	2.42	2.63	2.19 (2.95)
rel_avg	0.6	1	1.98	2.05	1.90	2.45	2.56	2.19 (3.13)
rel_avg	0.6		1.86	2.06	1.91	2.43	2.68	2.19 (2.82)
rel_avg	0.6	1.4	1.98	2.08	1.90	2.43	2.59	2.20 (3.55)
rel_avg	1		1.87	2.08	1.92	2.42	2.70	2.20 (3.27)
rel_avg	1	0.2	1.89	2.07	1.93	2.41	2.70	2.20 (3.43)
rel_avg	1	0.6	1.93	2.08	1.91	2.44	2.66	2.20 (3.60)
rel_avg	0.6	1.8	1.96	2.07	1.91	2.47	2.61	2.20 (3.74)
rel_avg	1	1	1.96	2.07	1.93	2.46	2.60	2.20 (3.83)
rel_avg	0.2	1.8	2.00	2.06	1.91	2.48	2.61	2.21 (4.10)
rel_avg	1.4	1	1.96	2.08	1.94	2.43	2.64	2.21 (4.10)
rel_avg	1.4	1.4	1.96	2.10	1.94	2.43	2.62	2.21 (4.17)
rel_avg	1	1.8	1.98	2.10	1.93	2.43	2.63	2.21 (4.24)
rel_avg	1.4	0.2	1.93	2.11	1.93	2.41	2.69	2.21 (4.13)
rel_avg	1.8	0.6	1.94	2.11	1.93	2.43	2.69	2.22 (4.50)
rel_avg	1.4	1.8	1.99	2.08	1.93	2.46	2.65	2.22 (4.61)
rel_avg	1.8	1.8	1.96	2.09	1.94	2.45	2.68	2.22 (4.60)
rel_avg	1.4		1.96	2.08	1.93	2.43	2.72	2.22 (4.59)
rel_dist	0.6		1.81	2.17	1.97	2.44	2.73	2.22 (4.41)
rel_dist	0.6	1	1.81	2.17	1.99	2.45	2.72	2.23 (4.51)
rel_dist	1.4		1.78	2.17	1.97	2.48	2.73	2.23 (4.42)
rel_avg	1.8	0.2	1.92	2.13	1.94	2.45	2.70	2.23 (4.74)
rel_dist	1.8	1.4	1.84	2.15	1.98	2.43	2.74	2.23 (4.61)
rel_avg	1.8	1.4	2.01	2.12	1.95	2.41	2.64	2.23 (5.01)
rel_dist	0.2	0.6	1.82	2.15	1.99	2.45	2.74	2.23 (4.60)
rel_dist	1.4	0.2	1.81	2.19	1.97	2.43	2.76	2.23 (4.70)
rel_dist	1	1	1.81	2.14	1.98	2.47	2.76	2.23 (4.67)
rel_dist	0.6	0.2	1.82	2.16	1.98	2.43	2.77	2.23 (4.78)
dist	1.8		1.95	2.09	1.98	2.52	2.62	2.23 (5.12)
rel_avg	1.4	0.6	2.01	2.12	1.94	2.42	2.68	2.23 (5.19)
rel_dist	0.2	0.2	1.83	2.18	1.97	2.45	2.75	2.23 (4.87)
dist	0.6		1.96	2.08	1.98	2.53	2.62	2.23 (5.19)
rel_dist	1	0.6	1.81	2.18	1.98	2.45	2.75	2.24 (4.88)

continued on next page

continued from previous page

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
avg_dist			1.90	2.14	1.97	2.51	2.65	2.24 (5.12)
rel_dist	0.2	1	1.82	2.17	1.97	2.49	2.72	2.24 (4.90)
rel_dist	1.8	0.2	1.83	2.17	1.97	2.45	2.76	2.24 (4.91)
rel_dist	1.8	0.6	1.83	2.16	1.98	2.45	2.76	2.24 (4.92)
rel_dist	0.6	1.4	1.82	2.19	1.98	2.47	2.72	2.24 (4.98)
rel_dist	1	0.2	1.82	2.16	1.98	2.47	2.75	2.24 (4.95)
rel_dist	0.2	1.4	1.83	2.17	1.97	2.50	2.71	2.24 (5.04)
rel_avg	1.8	1	2.00	2.11	1.95	2.44	2.69	2.24 (5.34)
dist	1.4		1.99	2.10	1.98	2.50	2.62	2.24 (5.44)
rel_dist	0.6	0.6	1.85	2.15	1.97	2.48	2.74	2.24 (5.06)
rel_avg	1.8		1.92	2.14	1.95	2.46	2.72	2.24 (5.23)
rel_dist	0.2		1.85	2.17	1.98	2.45	2.75	2.24 (5.14)
rel_dist	1	1.8	1.83	2.16	1.98	2.48	2.74	2.24 (5.13)
dist			1.91	2.13	1.97	2.53	2.66	2.24 (5.31)
rel_dist	1		1.84	2.18	1.98	2.47	2.74	2.24 (5.16)
rel_dist	1.4	0.6	1.82	2.19	1.98	2.47	2.75	2.24 (5.12)
rel_dist	1.8	1	1.85	2.18	1.99	2.42	2.76	2.24 (5.27)
rel_dist	1.4	1	1.85	2.19	1.98	2.46	2.73	2.24 (5.29)
rel_dist			1.86	2.20	1.97	2.43	2.75	2.24 (5.31)
rel_dist	1	1.4	1.82	2.20	1.98	2.50	2.73	2.24 (5.34)
dist	1		1.98	2.11	1.99	2.52	2.64	2.24 (5.73)
rel_dist	1.4	1.8	1.88	2.17	1.98	2.48	2.72	2.25 (5.49)
rel_dist	1.8		1.84	2.17	1.97	2.49	2.76	2.25 (5.37)
rel_dist	1.4	1.4	1.85	2.16	1.98	2.51	2.74	2.25 (5.51)
dist	0.2		2.00	2.10	1.98	2.51	2.66	2.25 (5.95)
rel_dist	0.6	1.8	1.92	2.17	1.98	2.46	2.72	2.25 (5.91)
rel_dist	1.8	1.8	1.89	2.21	1.99	2.47	2.72	2.26 (6.02)
rel_dist	0.2	1.8	1.95	2.16	1.99	2.47	2.71	2.26 (6.22)
dist		0.2	2.00	2.15	1.98	2.53	2.64	2.26 (6.40)
	1	0.2	1.93	2.15	2.00	2.48	2.76	2.26 (6.43)
	1.8	0.2	1.89	2.16	2.00	2.52	2.75	2.26 (6.39)
	1.4	0.2	1.93	2.17	2.00	2.50	2.75	2.27 (6.73)
	1		1.92	2.17	1.99	2.52	2.76	2.27 (6.70)
	0.6	0.2	1.95	2.18	2.00	2.47	2.76	2.27 (6.85)
dist		0.6	2.00	2.14	1.99	2.54	2.70	2.27 (6.93)
dist		1.4	2.00	2.17	2.01	2.51	2.73	2.28 (7.43)
	1.4	0.6	1.95	2.19	2.01	2.51	2.76	2.28 (7.35)
	1.8	0.6	1.94	2.21	2.00	2.50	2.77	2.28 (7.44)
u			1.86	2.20	2.06	2.53	2.78	2.29 (7.37)
	1.8	1	1.98	2.17	2.01	2.54	2.76	2.29 (7.70)

continued on next page

continued from previous page

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
dist		1	1.99	2.17	2.01	2.56	2.73	2.29 (7.87)
dist		1.8	1.99	2.18	2.03	2.54	2.73	2.29 (7.93)
	1.8	1.4	1.98	2.15	2.01	2.54	2.78	2.29 (7.84)
	1.4	1	2.01	2.18	2.01	2.52	2.77	2.30 (8.07)
	1	0.6	1.97	2.21	2.02	2.54	2.77	2.30 (8.30)
	0.6	1.4	1.98	2.20	2.03	2.54	2.77	2.30 (8.34)
	1.4	1.8	1.99	2.19	2.02	2.55	2.78	2.30 (8.36)
	0.2	0.2	2.01	2.19	2.01	2.53	2.78	2.31 (8.44)
	1.8	1.8	2.01	2.21	2.01	2.54	2.77	2.31 (8.53)
	0.6	0.6	2.00	2.19	2.02	2.54	2.79	2.31 (8.59)
	1.4	1.4	2.01	2.18	2.02	2.55	2.79	2.31 (8.63)
	1	1.4	1.99	2.20	2.02	2.55	2.79	2.31 (8.64)
	1	1	2.01	2.20	2.02	2.53	2.79	2.31 (8.71)
		1	2.01	2.19	2.04	2.55	2.77	2.31 (8.88)
	0.6	1	2.02	2.21	2.02	2.54	2.78	2.31 (8.88)
	1	1.8	2.01	2.19	2.02	2.55	2.80	2.31 (8.84)
ne			2.02	2.20	2.09	2.52	2.75	2.32 (9.10)
	0.2	1.4	2.08	2.17	2.03	2.53	2.77	2.32 (9.14)
	0.2	1	2.11	2.18	2.02	2.56	2.77	2.33 (9.75)
	0.2	1.8	2.10	2.18	2.03	2.55	2.79	2.33 (9.75)
	0.6	1.8	2.13	2.23	2.04	2.53	2.79	2.34 (10.51)
	0.2	0.6	2.17	2.21	2.03	2.57	2.78	2.35 (10.97)

Table A.4: All tested weighted border functions

			square		arrow		plate		shock		uk	
base	λ_b	λ_a	%	best	%	best	%	best	%	best	%	best
rel_avg	0.2	0.6	2	1.49	3	1.73	9	1.73	5	2.04	14	2.36
rel_avg	0.2	1.0	1	1.50	3	1.73	7	1.76	5	2.04	14	2.37
rel_avg	0.6	1.0	1	1.53	2	1.78	3	1.76	1	2.10	22	2.37
rel_avg	0.2	1.4	1	1.53	5	1.76	4	1.74	3	2.07	10	2.38
rel_avg	0.6		2	1.50	7	1.75	4	1.75	3	2.08	0	–
rel_avg	0.6	1.4	0	–	1	1.75	6	1.74	3	2.06	6	2.41
rel_avg			0	–	5	1.76	5	1.73	6	2.04	0	–
rel_avg	1.0	1.0	1	1.54	3	1.79	5	1.76	2	2.04	4	2.38
rel_avg	0.2		2	1.51	1	1.76	5	1.71	7	2.04	0	–
rel_avg	0.6	0.6	0	–	7	1.78	6	1.76	2	2.09	0	–
rel_avg	0.2	0.2	1	1.49	7	1.72	2	1.74	3	2.04	0	–
rel_avg	1.0	1.4	1	1.52	1	1.81	3	1.75	2	2.03	4	2.39
rel_avg	0.6	0.2	1	1.52	2	1.79	4	1.74	3	2.07	0	–
rel_avg	1.4	1.4	0	–	5	1.75	2	1.76	3	2.09	0	–
rel_avg	0.6	1.8	0	–	1	1.80	6	1.72	1	2.10	1	2.42
rel_avg	1.0	0.6	0	–	3	1.78	2	1.75	4	2.08	0	–
rel_avg	1.0		1	1.52	2	1.78	1	1.77	4	2.07	0	–
rel_avg	1.4	0.6	0	–	2	1.78	3	1.74	2	2.07	1	2.43
rel_avg	1.4	1.8	1	1.53	2	1.77	1	1.77	1	2.08	2	2.41
rel_avg	1.0	0.2	1	1.52	2	1.75	1	1.74	3	2.01	0	–
rel_dist	1.0	0.2	4	1.51	3	1.78	0	–	0	–	0	–
rel_avg	1.4		1	1.52	1	1.80	1	1.77	3	2.05	0	–
rel_avg	1.0	1.8	1	1.54	1	1.81	1	1.75	0	–	3	2.41
dist			1	1.54	0	–	2	1.77	0	–	3	2.39
rel_dist	1.4	0.2	3	1.51	1	1.80	0	–	1	2.10	0	–
rel_avg	1.4	1.0	0	–	1	1.78	2	1.75	2	2.07	0	–
dist		0.2	2	1.49	1	1.77	0	–	0	–	2	2.42
rel_dist	0.6	1.4	1	1.50	3	1.74	0	–	0	–	1	2.43
avg_dist			1	1.54	0	–	1	1.77	0	–	3	2.39
rel_dist	0.6	0.2	3	1.52	0	–	0	–	2	2.04	0	–
rel_dist	0.2	1.4	3	1.51	0	–	0	–	2	2.07	0	–
rel_dist	1.4		3	1.51	0	–	0	–	2	2.08	0	–
rel_dist	1.4	0.6	5	1.51	0	–	0	–	0	–	0	–
rel_dist	1.4	1.0	1	1.54	0	–	1	1.75	1	2.06	1	2.42
rel_avg	1.8	0.6	2	1.52	0	–	1	1.77	1	2.08	0	–
rel_dist	1.8	1.4	1	1.51	1	1.79	0	–	2	2.08	0	–
rel_avg	1.8		1	1.54	2	1.80	0	–	1	2.10	0	–
rel_avg	1.4	0.2	0	–	1	1.81	1	1.75	2	2.10	0	–
rel_dist	1.0	1.4	3	1.50	0	–	0	–	1	1.96	0	–
rel_avg	0.2	1.8	0	–	0	–	2	1.77	0	–	2	2.41

continued on next page

continued from previous page

			square		arrow		plate		shock		uk	
base	λ_b	λ_a	%	best	%	best	%	best	%	best	%	best
rel_avg	1.8	1.8	0	–	3	1.79	0	–	0	–	1	2.41
rel_dist	1.8	1.8	1	1.54	1	1.77	1	1.77	0	–	0	–
rel_avg	1.8	1.4	1	1.54	1	1.81	1	1.77	0	–	0	–
rel_avg	1.8	0.2	1	1.53	1	1.75	0	–	1	2.08	0	–
rel_dist	1.0	1.0	1	1.54	1	1.81	0	–	1	2.05	0	–
rel_dist			1	1.54	0	–	1	1.77	1	2.10	0	–
rel_dist	0.2		1	1.53	1	1.81	0	–	1	2.09	0	–
dist	1.4		0	–	1	1.80	0	–	1	2.10	1	2.43
rel_dist	1.8	0.2	2	1.52	0	–	1	1.75	0	–	0	–
rel_dist	1.0	0.6	2	1.51	0	–	1	1.78	0	–	0	–
rel_dist	0.6	1.0	1	1.50	2	1.80	0	–	0	–	0	–
	1.0	0.6	2	1.53	1	1.78	0	–	0	–	0	–
rel_dist	1.0		2	1.51	1	1.80	0	–	0	–	0	–
rel_dist	1.0	1.8	2	1.50	0	–	0	–	1	2.08	0	–
rel_dist	1.8	0.6	1	1.53	0	–	0	–	2	2.08	0	–
rel_dist	0.6	0.6	2	1.52	0	–	0	–	1	2.10	0	–
rel_avg	1.8	1.0	0	–	0	–	1	1.70	2	2.06	0	–
rel_dist	1.8		0	–	0	–	1	1.77	2	2.07	0	–
dist	0.6		1	1.51	0	–	0	–	0	–	2	2.42
dist	1.0		0	–	0	–	0	–	1	2.10	2	2.41
rel_dist	1.4	1.8	3	1.51	0	–	0	–	0	–	0	–
rel_dist	0.2	0.2	3	1.53	0	–	0	–	0	–	0	–
rel_dist	1.8	1.0	1	1.50	0	–	1	1.77	0	–	0	–
rel_dist	1.4	1.4	1	1.54	1	1.77	0	–	0	–	0	–
dist		1.0	1	1.52	1	1.81	0	–	0	–	0	–
rel_dist	0.2	0.6	1	1.52	0	–	0	–	1	2.10	0	–
rel_dist	0.6		0	–	0	–	1	1.76	1	2.08	0	–
ne			2	1.52	0	–	0	–	0	–	0	–
	1		2	1.53	0	–	0	–	0	–	0	–
	0.6	1.4	0	–	2	1.75	0	–	0	–	0	–
dist		1.8	1	1.50	0	–	0	–	0	–	0	–
		1	1	1.50	0	–	0	–	0	–	0	–
u			1	1.52	0	–	0	–	0	–	0	–
	1.4	0.2	1	1.52	0	–	0	–	0	–	0	–
	0.6	0.6	1	1.52	0	–	0	–	0	–	0	–
	1.4	1.4	1	1.52	0	–	0	–	0	–	0	–
	1.8	1.8	1	1.52	0	–	0	–	0	–	0	–
	1.0	1.0	1	1.52	0	–	0	–	0	–	0	–
	0.2	0.2	1	1.52	0	–	0	–	0	–	0	–

continued on next page

continued from previous page

			square		arrow		plate		shock		uk	
base	λ_b	λ_a	%	best	%	best	%	best	%	best	%	best
	0.6	0.2	1	1.53	0	–	0	–	0	–	0	–
	1.8	0.6	1	1.53	0	–	0	–	0	–	0	–
rel_dist	0.2	1.0	1	1.53	0	–	0	–	0	–	0	–
	1.0	1.8	1	1.53	0	–	0	–	0	–	0	–
	1.8	0.2	0	–	1	1.78	0	–	0	–	0	–
dist		0.6	0	–	1	1.78	0	–	0	–	0	–
	1.4	1.8	0	–	1	1.79	0	–	0	–	0	–
dist	0.2		0	–	1	1.81	0	–	0	–	0	–
	1.4	0.6	0	–	0	–	0	–	1	2.07	0	–
dist	1.8		0	–	0	–	0	–	0	–	1	2.44

Table A.5: Top 100: weighted border functions

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
rel_avg	0.2	0.6	1.81	2.01	1.92	2.28	2.61	2.12 (0.65)
rel_dist	0.2	0.2	1.80	2.00	1.92	2.29	2.64	2.13 (0.91)
rel_avg	0.2		1.82	2.00	1.92	2.28	2.65	2.13 (1.03)
rel_avg	0.2	0.2	1.83	1.99	1.92	2.30	2.63	2.13 (1.09)
rel_dist	0.2		1.84	2.00	1.92	2.31	2.65	2.14 (1.57)
rel_avg	0.2	1.0	1.81	2.05	1.94	2.29	2.64	2.15 (1.69)
rel_dist	0.2	0.6	1.83	2.01	1.93	2.31	2.65	2.15 (1.80)
rel_avg			1.76	2.00	1.96	2.38	2.67	2.15 (1.83)
rel_dist	0.2	1.0	1.88	2.04	1.95	2.29	2.65	2.16 (2.60)
rel_avg		0.6	1.79	2.03	1.97	2.34	2.69	2.16 (2.44)
rel_dist			1.84	2.00	1.96	2.37	2.67	2.17 (2.65)
rel_avg		0.2	1.82	2.01	1.95	2.36	2.70	2.17 (2.66)
rel_dist		0.2	1.81	2.04	1.97	2.35	2.70	2.17 (2.87)
rel_avg	0.2	1.4	1.85	2.08	1.95	2.33	2.69	2.18 (3.28)
rel_avg		1.0	1.84	2.04	1.99	2.36	2.71	2.19 (3.64)
rel_dist		0.6	1.86	2.04	1.99	2.36	2.69	2.19 (3.72)
rel_avg	0.6	0.2	1.89	2.08	1.95	2.37	2.66	2.19 (3.84)
rel_dist	0.6	0.6	1.91	2.10	1.96	2.33	2.66	2.19 (4.01)
rel_avg	0.6	1.0	1.89	2.07	1.97	2.38	2.65	2.19 (3.98)
rel_avg	0.6	0.6	1.86	2.09	1.96	2.38	2.68	2.19 (4.01)
rel_avg	0.6		1.93	2.08	1.94	2.37	2.66	2.20 (4.23)
rel_dist	0.2	1.4	1.96	2.05	1.96	2.33	2.69	2.20 (4.34)
rel_dist	0.6	1.0	1.93	2.09	1.95	2.37	2.64	2.20 (4.36)
rel_dist		1.0	1.90	2.07	1.99	2.37	2.69	2.20 (4.46)
rel_dist	0.6		1.90	2.11	1.95	2.41	2.66	2.21 (4.72)
rel_dist	0.6	0.2	1.89	2.10	1.94	2.42	2.69	2.21 (4.60)
rel_avg		1.4	1.88	2.08	2.00	2.37	2.71	2.21 (4.73)
rel_avg	0.2	1.8	1.93	2.09	1.98	2.34	2.70	2.21 (4.85)
rel_avg	0.6	1.4	1.89	2.12	1.95	2.42	2.68	2.21 (4.87)
rel_dist	0.6	1.4	1.98	2.13	1.95	2.37	2.67	2.22 (5.40)
rel_avg	0.6	1.8	1.95	2.12	1.98	2.38	2.67	2.22 (5.41)
rel_dist	0.2	1.8	2.02	2.10	1.98	2.33	2.70	2.23 (5.81)
rel_avg	1.0	0.6	1.86	2.13	1.96	2.48	2.70	2.23 (5.45)
rel_dist		1.4	1.96	2.09	2.01	2.40	2.69	2.23 (5.90)
rel_avg	1.0	1.0	1.92	2.15	1.97	2.43	2.70	2.23 (5.82)
rel_dist	1.0	0.6	1.91	2.17	1.96	2.47	2.69	2.24 (6.16)
rel_dist	0.6	1.8	2.02	2.13	1.97	2.40	2.69	2.24 (6.44)
rel_dist	1.0		1.88	2.14	1.96	2.55	2.71	2.25 (6.36)
rel_avg		1.8	1.97	2.14	2.02	2.42	2.74	2.26 (7.09)
rel_avg	1.0	1.8	1.98	2.18	1.99	2.46	2.70	2.26 (7.42)
rel_dist		1.8	2.06	2.12	2.01	2.41	2.73	2.27 (7.64)

continued on next page

continued from previous page

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
rel_dist	1.0	0.2	1.92	2.16	1.98	2.55	2.74	2.27 (7.34)
rel_avg	1.0	1.4	1.96	2.18	1.97	2.50	2.72	2.27 (7.52)
rel_dist	1.0	1.0	1.99	2.16	1.96	2.49	2.73	2.27 (7.57)
rel_avg	1.0	0.2	1.98	2.17	1.99	2.49	2.72	2.27 (7.72)
rel_dist	1.0	1.4	2.03	2.17	1.97	2.49	2.73	2.28 (8.01)
rel_avg	1.0		2.03	2.17	1.95	2.51	2.73	2.28 (8.08)
rel_dist	1.4	0.6	1.90	2.22	1.98	2.57	2.76	2.28 (8.13)
rel_avg	1.4	1.0	1.93	2.20	1.98	2.55	2.76	2.28 (8.21)
rel_dist	1.0	1.8	2.06	2.21	1.98	2.48	2.71	2.29 (8.65)
rel_avg	1.4	0.6	1.95	2.21	1.99	2.57	2.74	2.29 (8.67)
rel_avg	1.4	1.4	1.94	2.28	1.98	2.57	2.70	2.30 (8.83)
rel_dist	1.4	1.4	1.97	2.24	2.00	2.56	2.77	2.31 (9.43)
rel_avg	1.4	0.2	1.97	2.24	1.98	2.63	2.74	2.31 (9.60)
rel_dist	1.4	1.0	1.93	2.24	1.98	2.67	2.75	2.31 (9.49)
rel_dist	1.4		1.98	2.22	1.98	2.66	2.77	2.32 (9.93)
rel_dist	1.4	1.8	2.01	2.24	2.01	2.60	2.78	2.33 (10.37)
rel_avg	1.8	1.0	1.93	2.24	2.02	2.67	2.78	2.33 (10.23)
rel_avg	1.4	1.8	2.01	2.27	2.01	2.60	2.76	2.33 (10.48)
rel_avg	1.4		2.03	2.25	1.98	2.63	2.78	2.33 (10.63)
rel_dist	1.4	0.2	1.96	2.24	2.00	2.70	2.77	2.33 (10.47)
rel_dist	1.8	1.4	1.98	2.27	1.98	2.67	2.78	2.33 (10.53)
rel_dist	1.8	0.6	1.94	2.27	1.99	2.75	2.72	2.34 (10.54)
rel_avg	1.8	1.8	2.02	2.31	2.01	2.64	2.72	2.34 (10.96)
rel_avg	1.8	0.6	1.94	2.26	1.99	2.73	2.78	2.34 (10.65)
rel_dist	1.8	1.8	2.01	2.28	1.99	2.65	2.77	2.34 (10.89)
rel_dist	1.8	1.0	1.93	2.29	2.00	2.76	2.75	2.35 (11.07)
rel_avg	1.8	1.4	2.00	2.27	2.01	2.74	2.73	2.35 (11.33)
rel_avg	1.8	0.2	2.01	2.27	1.98	2.71	2.81	2.35 (11.48)
rel_dist	1.8		2.00	2.24	2.00	2.77	2.83	2.37 (12.05)
rel_dist	1.8	0.2	1.97	2.28	2.02	2.77	2.84	2.38 (12.42)
rel_avg	1.8		2.08	2.26	1.99	2.77	2.79	2.38 (12.70)

Table A.6: All tested weighted element functions

base	λ_b	λ_a	square		arrow		plate		shock		uk	
			%	best	%	best	%	best	%	best	%	best
rel_avg	0.2		11	1.50	10	1.75	6	1.74	15	1.96	2	2.42
rel_avg	0.2	0.2	2	1.52	10	1.73	9	1.72	9	2.01	5	2.38
rel_dist	0.2		6	1.52	6	1.75	8	1.74	6	2.07	9	2.37
rel_avg	0.2	0.6	6	1.49	5	1.77	4	1.73	10	2.04	6	2.41
rel_avg		0.2	10	1.50	10	1.74	3	1.70	4	2.08	3	2.42
rel_dist	0.2	0.2	6	1.50	8	1.72	6	1.75	7	2.04	3	2.43
rel_avg			15	1.49	8	1.76	3	1.75	1	2.10	1	2.42
rel_avg	0.2	1.0	1	1.52	2	1.80	3	1.76	10	2.04	5	2.39
rel_dist	0.2	0.6	1	1.54	7	1.76	3	1.77	6	2.05	4	2.40
rel_dist	0.2	1.4	0	–	4	1.80	3	1.74	7	2.03	4	2.37
rel_avg	0.6	0.2	1	1.53	1	1.81	5	1.73	2	2.05	5	2.39
rel_dist		0.2	8	1.50	2	1.77	2	1.75	0	–	2	2.39
rel_dist	0.6	0.2	1	1.52	3	1.79	2	1.73	0	–	7	2.41
rel_dist		0.6	4	1.52	5	1.77	2	1.76	1	2.06	0	–
rel_dist	0.2	1.0	1	1.54	0	–	3	1.76	6	2.04	2	2.38
rel_avg	0.6		2	1.49	1	1.76	4	1.75	1	2.06	3	2.39
rel_avg		0.6	4	1.52	3	1.79	2	1.77	1	2.04	0	–
rel_avg	0.6	0.6	0	–	1	1.77	4	1.74	0	–	4	2.40
rel_avg	0.6	1.0	5	1.52	0	–	0	–	2	2.06	2	2.41
rel_avg		1.0	5	1.50	0	–	1	1.76	1	2.09	1	2.43
rel_avg	0.6	1.4	1	1.54	0	–	0	–	4	2.10	3	2.40
rel_avg	0.2	1.4	1	1.51	1	1.81	1	1.77	2	2.07	2	2.38
rel_avg	1.0	0.6	2	1.53	0	–	3	1.76	0	–	2	2.42
rel_dist			0	–	4	1.73	1	1.78	0	–	2	2.41
rel_dist	0.6		1	1.54	2	1.78	1	1.77	0	–	2	2.41
rel_dist	0.6	1.0	1	1.54	1	1.81	2	1.75	1	2.08	0	–
rel_dist	1.0	0.6	2	1.50	0	–	2	1.76	0	–	0	–
rel_avg	1.0	1.4	0	–	0	–	1	1.76	0	–	3	2.42
rel_dist		1.0	0	–	2	1.79	0	–	0	–	2	2.40
rel_avg	0.2	1.8	0	–	0	–	0	–	3	2.09	1	2.42
rel_dist	0.2	1.8	0	–	0	–	0	–	0	–	4	2.42
rel_dist	0.6	1.4	0	–	0	–	1	1.77	1	2.10	1	2.43
rel_avg	1.4	1.0	1	1.51	0	–	2	1.77	0	–	0	–
rel_dist	1.0		1	1.54	2	1.78	0	–	0	–	0	–
rel_avg	1.0	0.2	0	–	0	–	1	1.75	0	–	2	2.36
rel_dist	0.6	1.8	0	–	0	–	2	1.71	0	–	1	2.42
rel_avg		1.8	1	1.53	0	–	0	–	0	–	1	2.42
rel_avg		1.4	0	–	0	–	1	1.76	0	–	1	2.41
rel_dist	0.6	0.6	0	–	1	1.80	0	–	0	–	1	2.42
rel_dist	1.0	0.2	0	–	0	–	2	1.74	0	–	0	–

continued on next page

continued from previous page

			square		arrow		plate		shock		uk	
base	λ_b	λ_a	%	best	%	best	%	best	%	best	%	best
rel_avg	1.0		0	–	0	–	1	1.75	0	–	0	–
rel_dist	1.0	1.0	0	–	0	–	1	1.76	0	–	0	–
rel_avg	1.8	1.4	0	–	0	–	1	1.77	0	–	0	–
rel_avg	0.6	1.8	0	–	0	–	1	1.77	0	–	0	–
rel_avg	1.4	0.2	0	–	0	–	1	1.77	0	–	0	–
rel_avg	1.8	0.2	0	–	0	–	1	1.77	0	–	0	–
rel_dist	1.8	0.2	0	–	0	–	1	1.77	0	–	0	–
rel_avg	1.0	1.8	0	–	1	1.80	0	–	0	–	0	–
rel_dist	1.0	1.4	0	–	0	–	0	–	0	–	1	2.40
rel_avg	1.4		0	–	0	–	0	–	0	–	1	2.43
rel_dist	1.0	1.8	0	–	0	–	0	–	0	–	1	2.43
rel_dist		1.4	0	–	0	–	0	–	0	–	1	2.43

Table A.7: Top 100: weighted element functions

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
ar			1.75	1.99	1.78	2.18	2.46	2.03 (0.98)
rel_avg	0.2	0.2	1.81	1.93	1.88	2.29	2.55	2.09 (3.91)
rel_avg	1.0	0.2	1.89	1.92	1.87	2.35	2.55	2.11 (5.06)
rel_avg	0.2	0.6	1.87	1.95	1.88	2.33	2.55	2.12 (5.19)
rel_avg	0.6	0.2	1.89	1.95	1.85	2.35	2.57	2.12 (5.33)
rel_avg	0.2		1.84	1.96	1.89	2.34	2.57	2.12 (5.37)
rel_avg	1.0	0.6	1.91	1.96	1.89	2.31	2.59	2.13 (5.89)
rel_avg	1.0		1.90	1.99	1.88	2.36	2.58	2.14 (6.38)
rel_avg	1		1.91	1.97	1.88	2.38	2.59	2.15 (6.62)
rel_dist	0.2	0.2	1.93	1.98	1.93	2.35	2.57	2.15 (7.01)
rel_dist	0.2	0.6	1.92	2.02	1.94	2.32	2.57	2.15 (7.06)
rel_dist	1.0	0.2	1.93	1.98	1.91	2.36	2.58	2.15 (7.06)
rel_avg	0.2	1.0	1.91	1.99	1.90	2.36	2.61	2.15 (6.99)
rel_dist	0.2		1.91	1.98	1.96	2.37	2.56	2.15 (7.16)
	1		1.88	2.01	1.86	2.41	2.62	2.16 (6.97)
rel_avg	0.6	0.6	1.92	1.99	1.87	2.44	2.56	2.16 (7.14)
rel_avg	1.0	1.0	1.90	1.99	1.89	2.42	2.59	2.16 (7.13)
rel_dist	0.6	0.2	1.97	2.02	1.90	2.37	2.54	2.16 (7.46)
rel_avg	1	1	1.94	1.98	1.89	2.38	2.59	2.16 (7.29)
rel_dist	0.2	1.0	1.94	2.00	1.95	2.35	2.57	2.16 (7.58)
rel_avg	0.6		2.02	1.97	1.86	2.40	2.59	2.17 (7.82)
rel_avg	0.6	1.4	2.05	2.02	1.89	2.30	2.59	2.17 (8.07)
rel_avg	0.6	1.0	2.01	2.05	1.85	2.39	2.56	2.17 (8.14)
rel_dist	1.0		2.00	2.00	1.93	2.38	2.60	2.18 (8.53)
rel_dist	1		1.99	2.00	1.93	2.42	2.59	2.18 (8.62)
rel_dist	0.6	0.6	2.00	2.01	1.95	2.32	2.65	2.18 (8.71)
rel_dist	1.0	1.0	1.96	2.02	1.92	2.43	2.60	2.19 (8.69)
rel_avg	0.2	1.4	1.96	2.06	1.94	2.35	2.62	2.19 (8.83)
rel_dist	1	1	2.00	2.02	1.94	2.40	2.61	2.19 (9.11)
rel_dist	0.6	1.0	2.05	2.05	1.91	2.35	2.62	2.20 (9.33)
rel_dist	1.0	0.6	1.96	2.00	1.94	2.46	2.64	2.20 (9.19)
rel_avg	0.2	1.8	2.09	2.02	1.95	2.42	2.53	2.20 (9.78)
rel_dist	0.6		2.06	1.99	1.89	2.48	2.65	2.21 (9.94)
rel_dist	0.6	1.4	2.05	1.98	1.91	2.49	2.65	2.22 (10.19)
rel_dist	0.2	1.8	2.05	2.11	1.99	2.37	2.60	2.23 (10.93)
rel_avg	1.0	1.4	2.05	2.00	1.86	2.64	2.59	2.23 (10.54)
rel_dist	0.2	1.4	2.01	2.07	1.97	2.43	2.67	2.23 (10.86)
rel_dist	1.0	1.4	2.17	2.04	1.94	2.47	2.57	2.24 (11.52)
rel_avg	0.6	1.8	2.10	2.00	1.90	2.59	2.60	2.24 (11.24)
rel_dist	1.0	1.8	2.09	2.04	1.96	2.37	2.74	2.24 (11.31)
rel_avg			1.98	2.09	1.98	2.48	2.70	2.24 (11.50)

continued on next page

continued from previous page

base	λ_b	λ_a	square	arrow	plate	shock	uk	avg (δ)
rel_dist	1.4	1.8	2.19	2.12	1.82	2.35	2.75	2.25 (11.81)
rel_dist	1.4	1.4	2.02	2.04	1.92	2.81	2.48	2.25 (11.94)
rel_dist			2.02	2.10	2.02	2.50	2.69	2.27 (12.74)
rel_avg	1.0	1.8	2.06	2.05	1.94	2.56	2.79	2.28 (12.98)
rel_dist	0.6	1.8	2.16	2.12	1.92	2.46	2.74	2.28 (13.52)
rel_avg	1.8		2.22	2.05	1.88	2.47	2.82	2.29 (13.61)
rel_avg	1.8	1.8	2.16	2.05	1.77	3.09	2.47	2.31 (14.04)
rel_avg	1.8	1.4	1.92	2.04	1.94	2.37	3.29	2.31 (13.58)
rel_avg	1.8	1.0	2.49	2.06	2.02	2.58	2.43	2.32 (15.97)
rel_avg	1.8	0.2	2.21	1.93	1.81	2.45	3.22	2.33 (14.37)
rel_avg	1.4	1.8	2.11	2.09	2.00	2.27	3.16	2.33 (15.04)
rel_avg	1.4	1.4	1.85	2.02	1.79	3.02	2.96	2.33 (13.93)
rel_dist	1.4	1.0	2.28	2.02	1.98	2.53	2.85	2.33 (15.92)
rel_avg	1.4		2.37	2.01	1.90	2.87	2.57	2.35 (16.53)
rel_avg	1.4	0.2	2.52	2.03	1.90	2.76	2.61	2.36 (17.51)
rel_avg	1.4	1.0	2.52	2.07	1.86	2.64	2.80	2.38 (18.15)
rel_dist	1.4		2.47	2.14	2.00	2.65	2.64	2.38 (18.83)
rel_avg	1.4	0.6	2.44	2.10	1.84	2.75	2.78	2.38 (18.22)
rel_dist	1.4	0.6	2.26	2.04	1.93	3.02	2.71	2.39 (18.37)
rel_dist	1.8		2.36	2.04	1.87	2.72	3.02	2.40 (18.75)
rel_dist	1.4	0.2	2.56	2.06	1.90	2.82	2.97	2.46 (21.91)
rel_dist	1.8	1.0	2.59	2.21	2.16	2.41	2.97	2.47 (23.11)
rel_dist	1.8	1.4	2.77	2.03	1.83	3.00	2.94	2.51 (24.21)
rel_dist	1.8	1.8	2.72	2.53	1.90	3.05	2.50	2.54 (26.45)
rel_dist	1.8	0.6	3.21	2.01	1.79	3.22	3.03	2.65 (30.46)
rel_dist	1.8	0.2	3.18	2.06	2.05	3.37	2.97	2.73 (34.83)
rel_avg	1.8	0.6	3.57	2.04	1.98	3.30	3.02	2.78 (37.41)

Table A.8: All tested weighted optimisation functions

			square		arrow		plate		shock		uk	
base	λ_b	λ_a	%	best	%	best	%	best	%	best	%	best
ar			16	1.48	20	1.69	63	1.62	34	1.96	30	2.25
	1		58	1.48	17	1.72	23	1.64	19	1.97	23	2.27
rel_avg	1		20	1.47	25	1.68	9	1.68	27	1.95	14	2.25
rel_avg	1	1	6	1.52	20	1.72	5	1.67	16	1.96	9	2.27
rel_dist	1		0	–	12	1.73	0	–	3	1.99	18	2.26
rel_avg	1.0		9	1.47	3	1.72	3	1.68	1	2.02	1	2.28
rel_avg	0.2		3	1.49	6	1.72	0	–	7	1.95	1	2.29
rel_dist	1	1	0	–	6	1.72	0	–	1	1.98	6	2.26
rel_avg	0.2	0.2	0	–	6	1.73	0	–	5	1.96	1	2.29
rel_avg	0.6		0	–	2	1.74	2	1.69	4	1.99	3	2.25
rel_avg	0.6	0.2	0	–	3	1.74	2	1.69	1	2.03	4	2.27
rel_avg	1.0	1.0	2	1.54	3	1.72	1	1.67	1	2.00	0	–
rel_avg	1.0	0.6	3	1.52	1	1.73	0	–	1	2.04	0	–
rel_avg	0.2	0.6	1	1.57	2	1.76	0	–	2	2.03	0	–
rel_avg	0.6	1.0	0	–	1	1.76	0	–	3	2.00	1	2.30
rel_avg	1.0	0.2	0	–	0	–	1	1.67	1	1.99	2	2.27
rel_avg	0.2	1.0	0	–	4	1.75	0	–	0	–	0	–
rel_dist	0.6		0	–	2	1.74	0	–	0	–	1	2.26
rel_dist	0.2		0	–	1	1.73	0	–	0	–	2	2.29
rel_dist	0.6	0.2	0	–	1	1.75	0	–	0	–	2	2.29
rel_avg	0.6	0.6	0	–	0	–	0	–	2	1.98	1	2.29
rel_dist	0.2	0.2	0	–	1	1.76	0	–	0	–	1	2.29
rel_dist	0.6	0.6	0	–	0	–	0	–	1	1.98	1	2.29
rel_dist	0.2	1.0	0	–	2	1.76	0	–	0	–	0	–
rel_avg	1.4	0.6	0	–	0	–	1	1.68	0	–	0	–
rel_avg	1.4		0	–	0	–	1	1.69	0	–	0	–
rel_dist	0.2	0.6	0	–	1	1.72	0	–	0	–	0	–
rel_dist	1.0	0.6	0	–	1	1.77	0	–	0	–	0	–
rel_dist	1.0	1.0	0	–	0	–	0	–	0	–	1	2.26
rel_dist	1.0		0	–	0	–	0	–	0	–	1	2.27
rel_dist	1.0	0.2	0	–	0	–	0	–	0	–	1	2.29

Table A.9: Top 100: weighted optimisation functions

	square	arrow	plate	shock	uk	avg (δ)
Diff	1.93	2.12	1.98	2.47	2.69	2.24 (0.12)
GDE	1.92	2.16	1.97	2.49	2.73	2.26 (0.82)

Table A.10: Tested flow algorithms

	square		arrow		plate		shock		uk	
	%	best	%	best	%	best	%	best	%	best
Diff	58	1.49	59	1.73	38	1.70	44	2.01	58	2.36
GDE	42	1.49	41	1.72	62	1.71	56	1.96	42	2.37

Table A.11: Top 100: flow algorithms

Appendix B

Illustrations of meshes

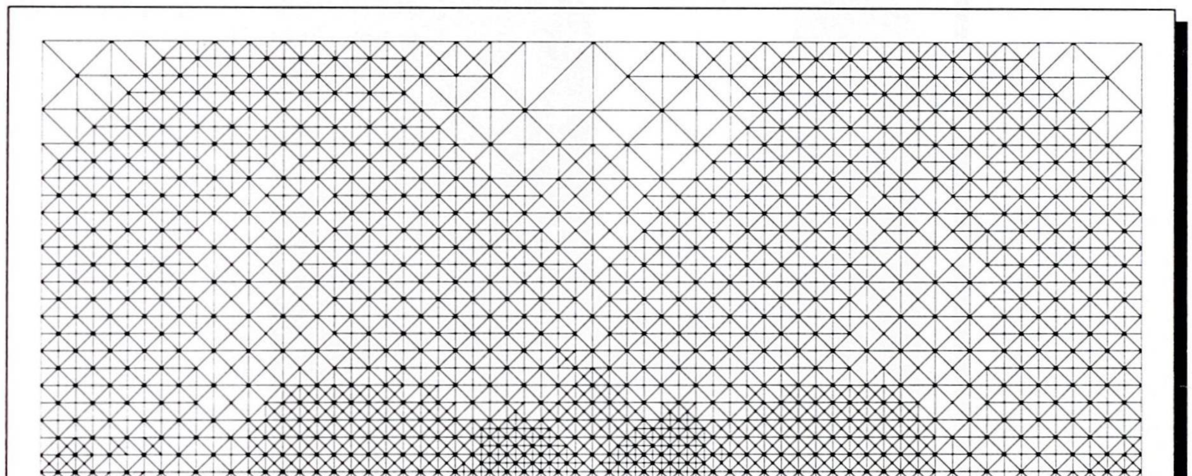


Figure B.1: Crack

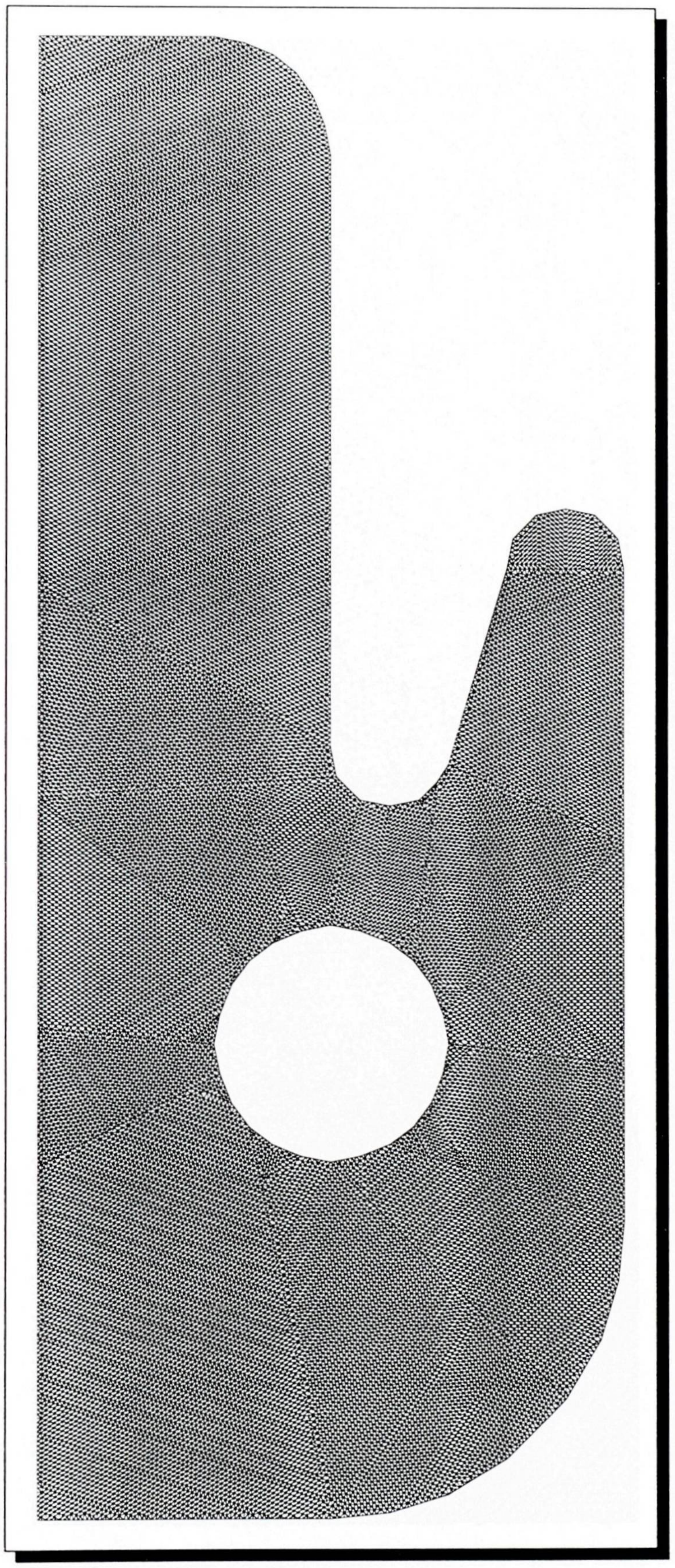


Figure B.2: T60k

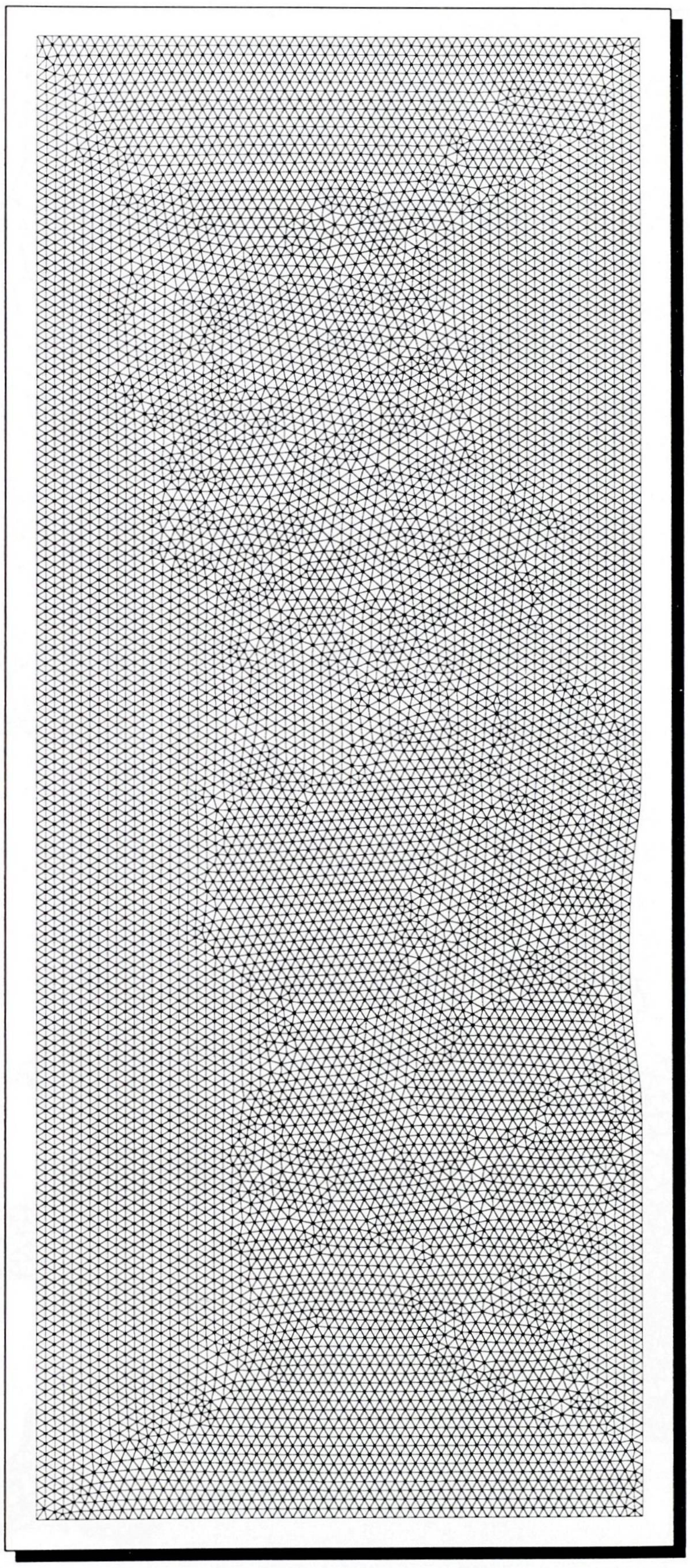


Figure B.3: Whitaker

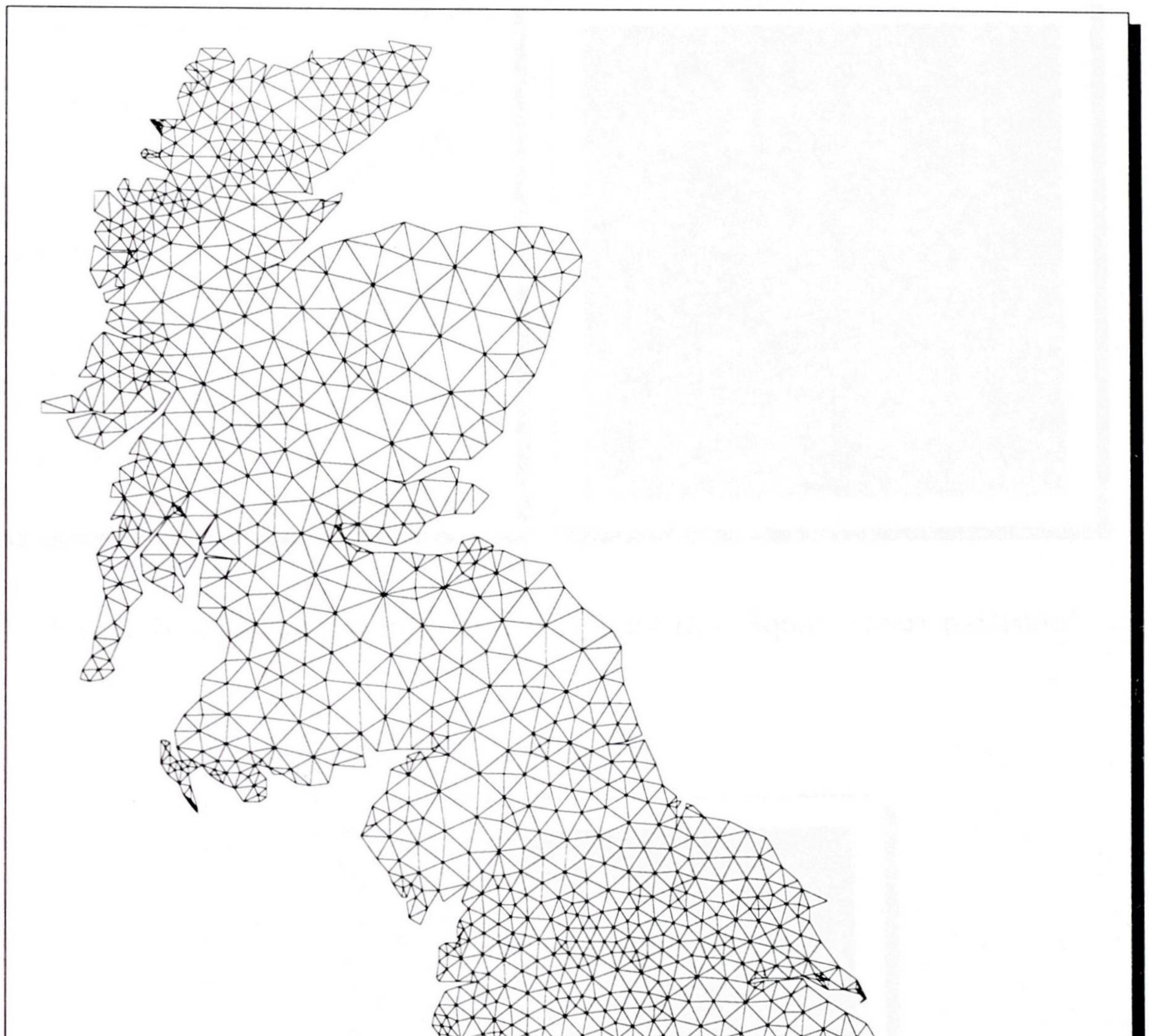


Figure B.4: UK

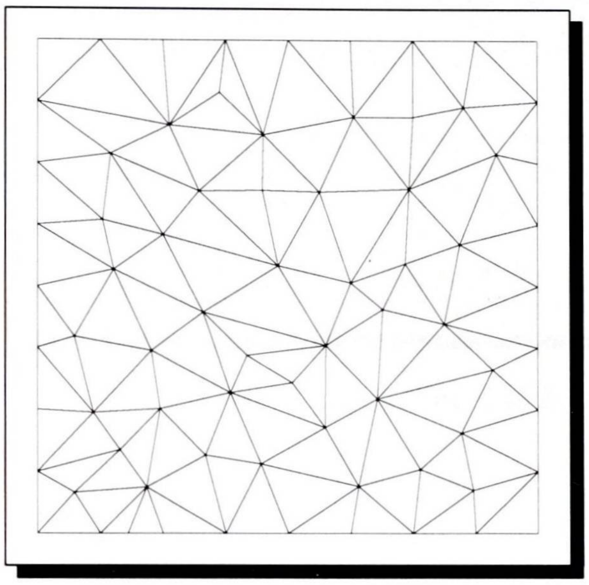


Figure B.5: Square (graph)

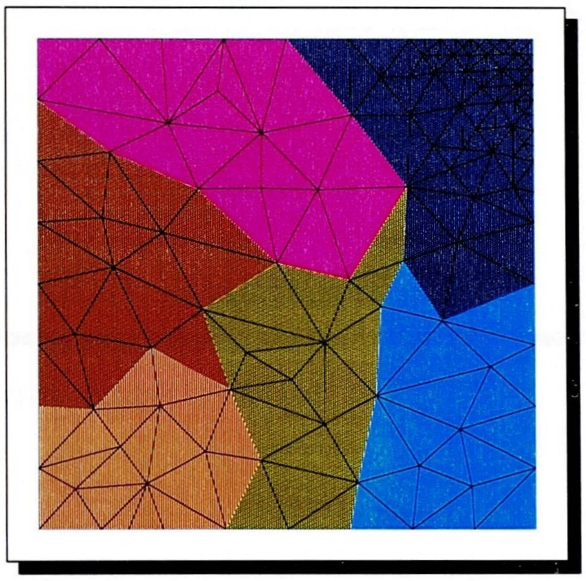


Figure B.6: Square (start partition)

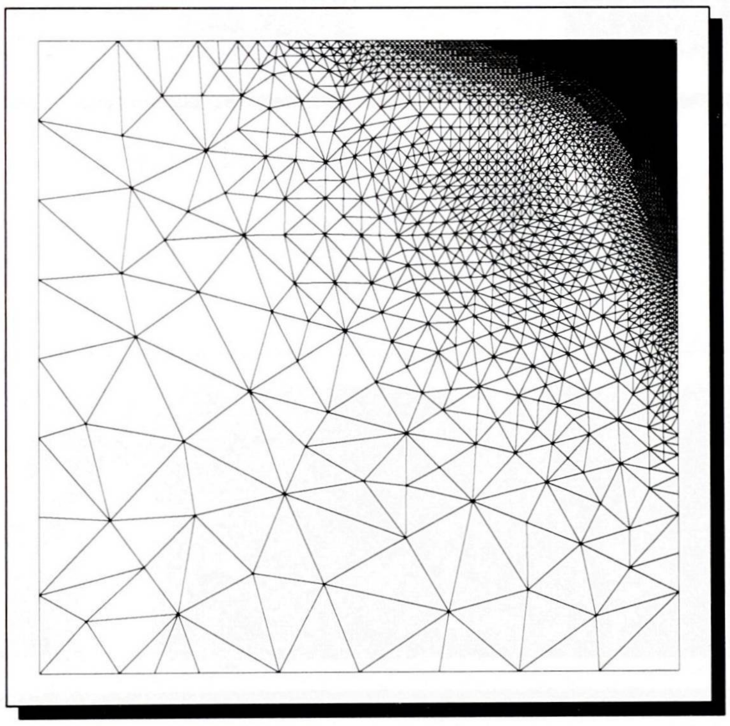


Figure B.7: Square (final)

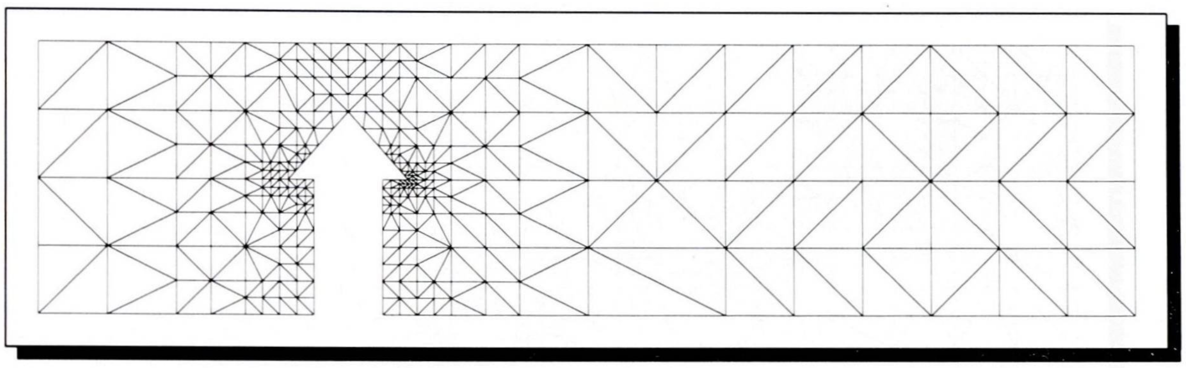


Figure B.8: Arrow (graph)

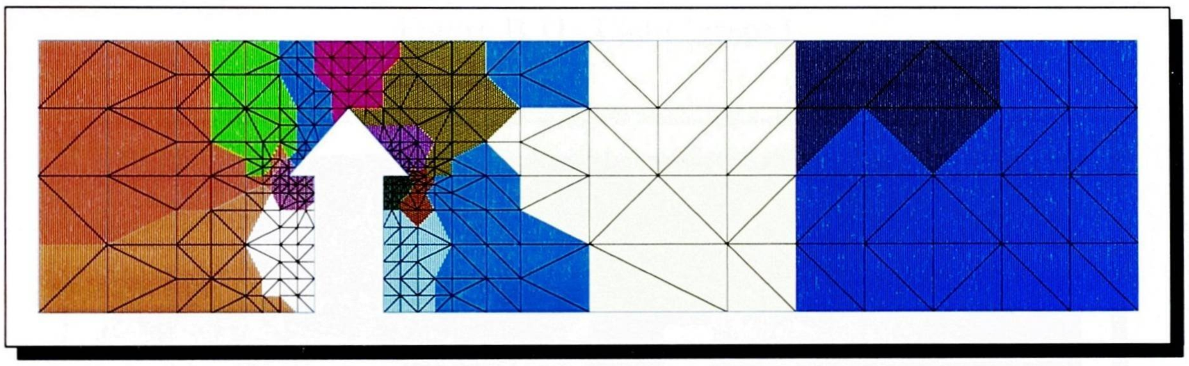


Figure B.9: Arrow (start partition)

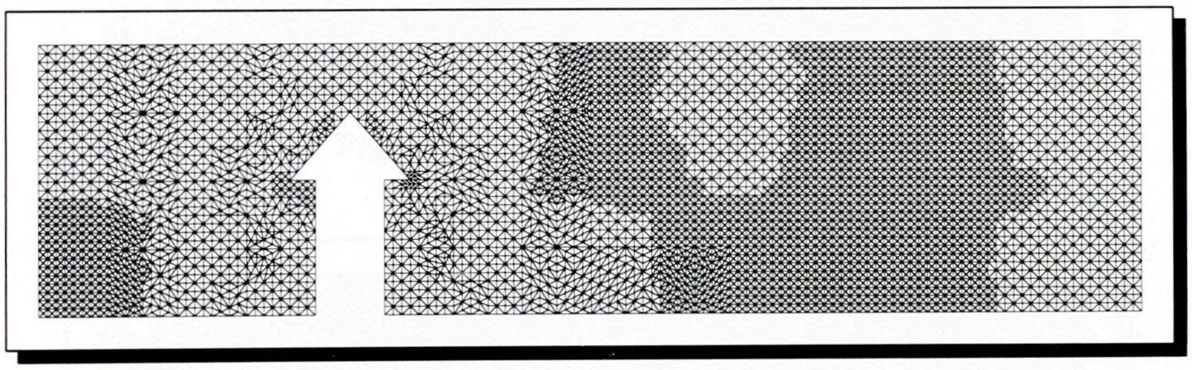


Figure B.10: Arrow (final)

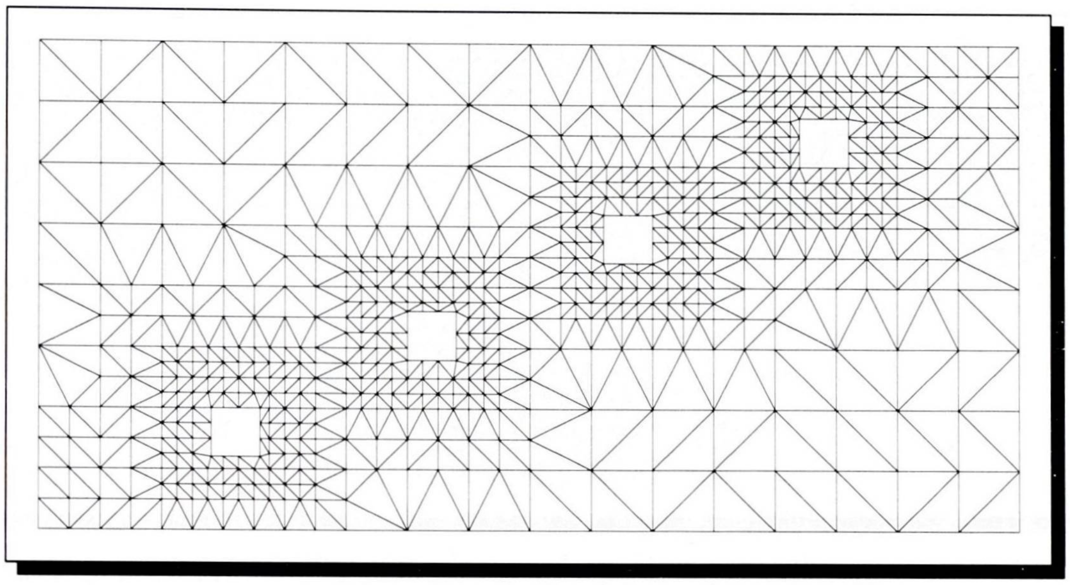


Figure B.11: Plate (graph)

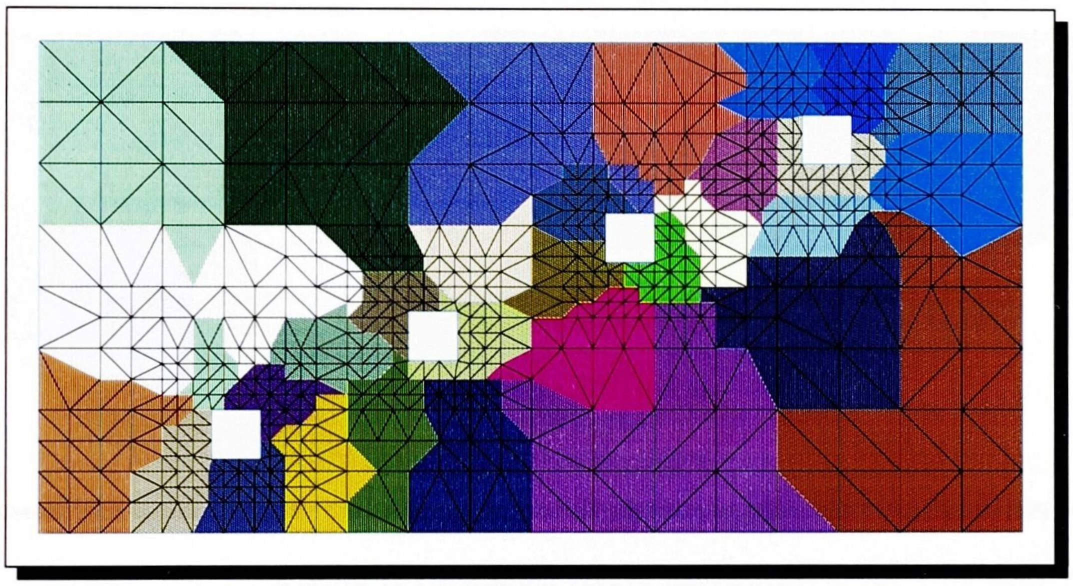


Figure B.12: Plate (start partition)

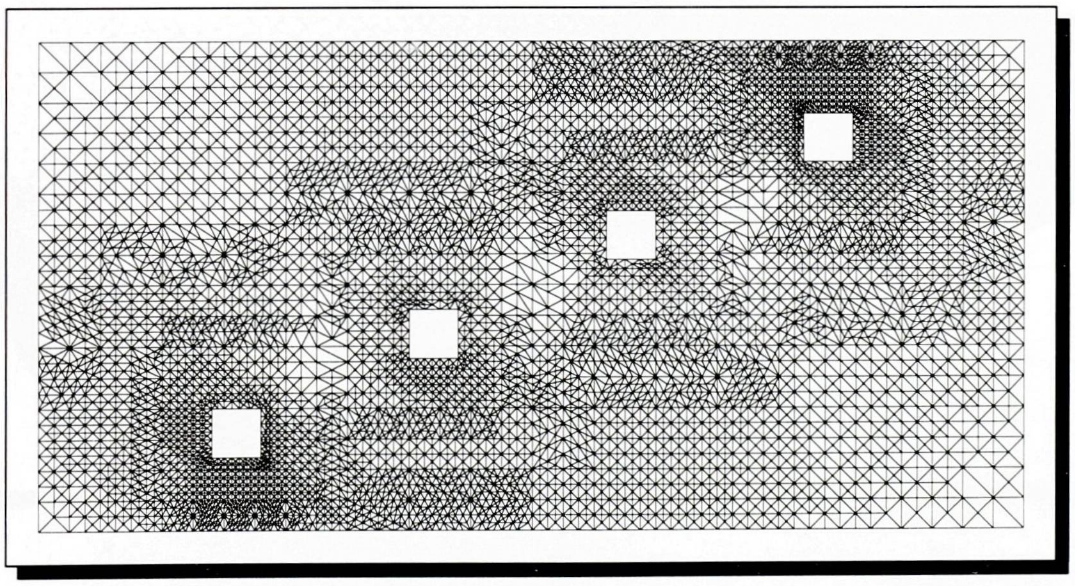


Figure B.13: Plate (final)

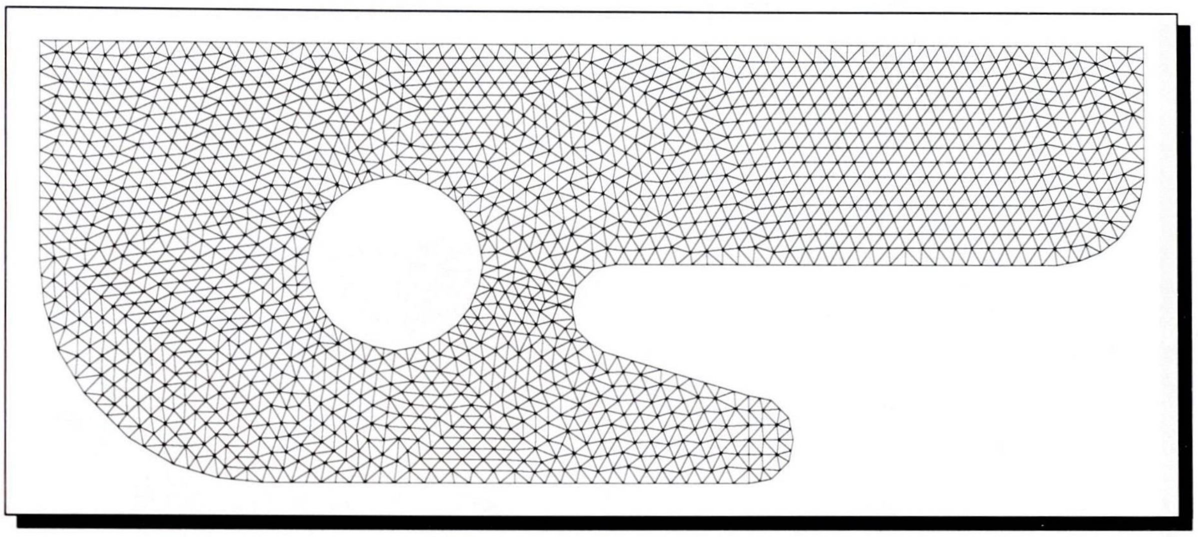


Figure B.14: Shock (graph)

Figure B.15: Shock (start partition)

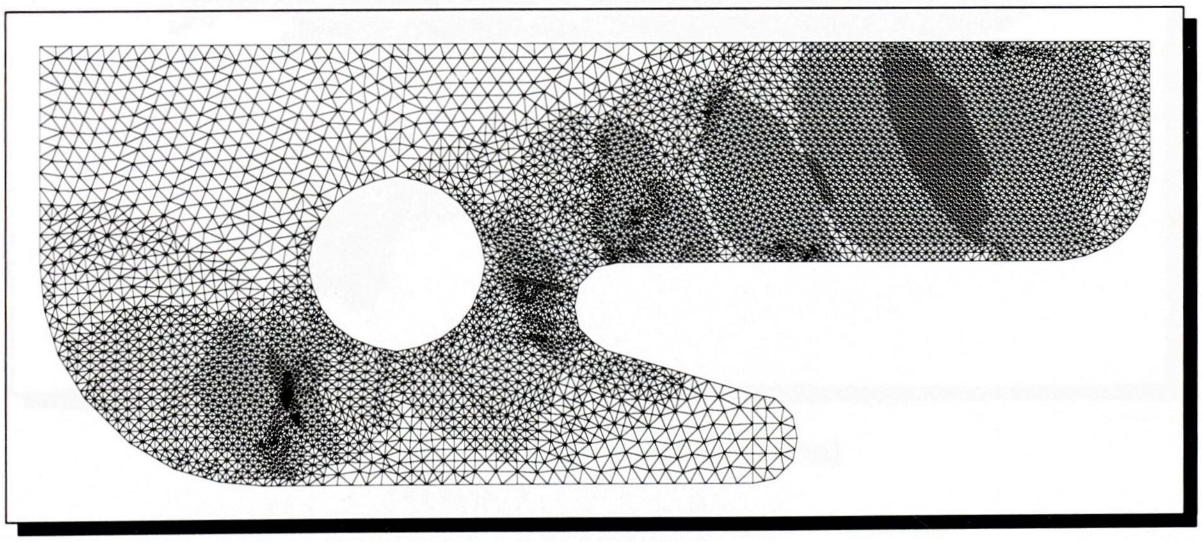


Figure B.16: Shock (final)

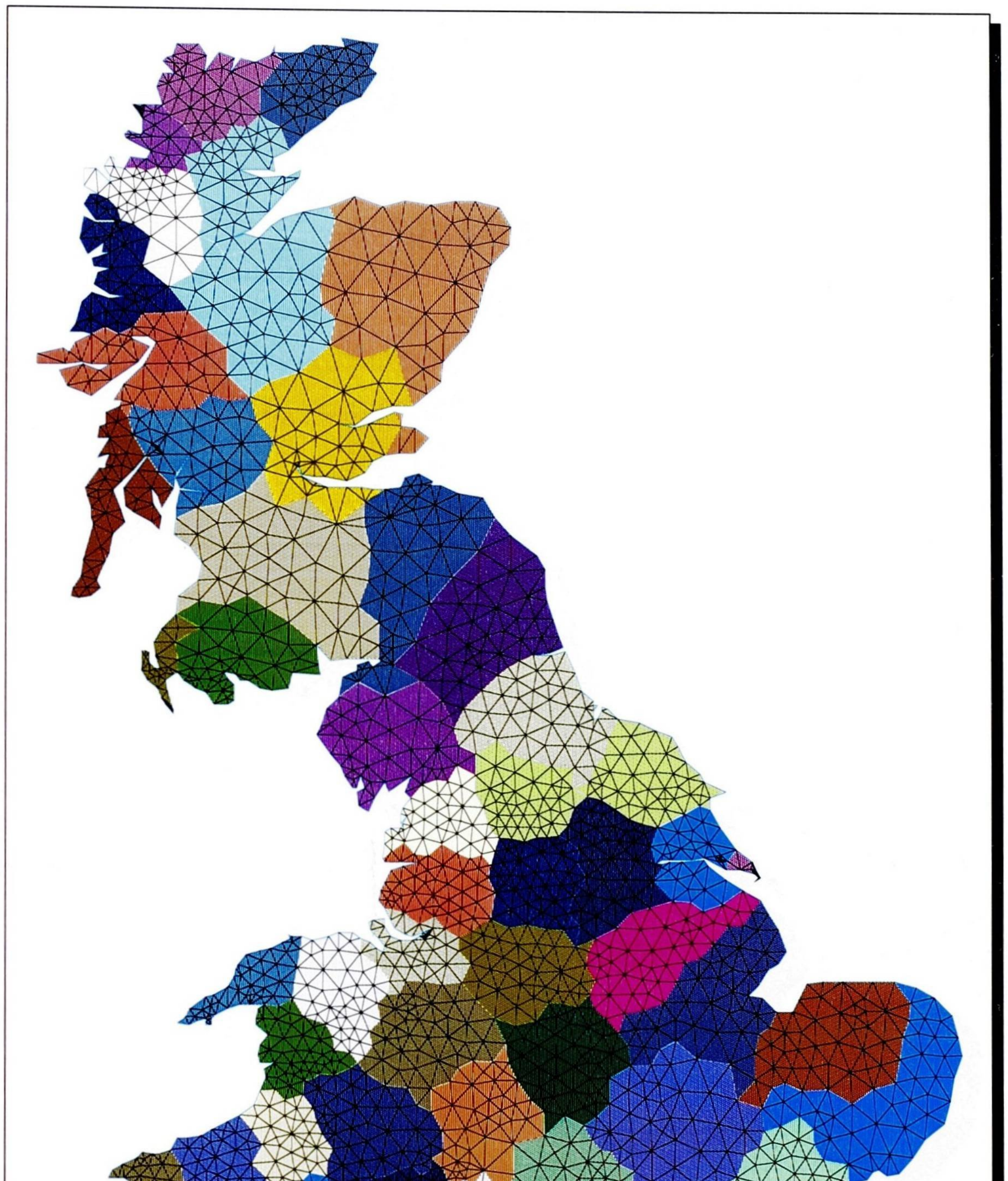


Figure B.17: Uk (start partition)

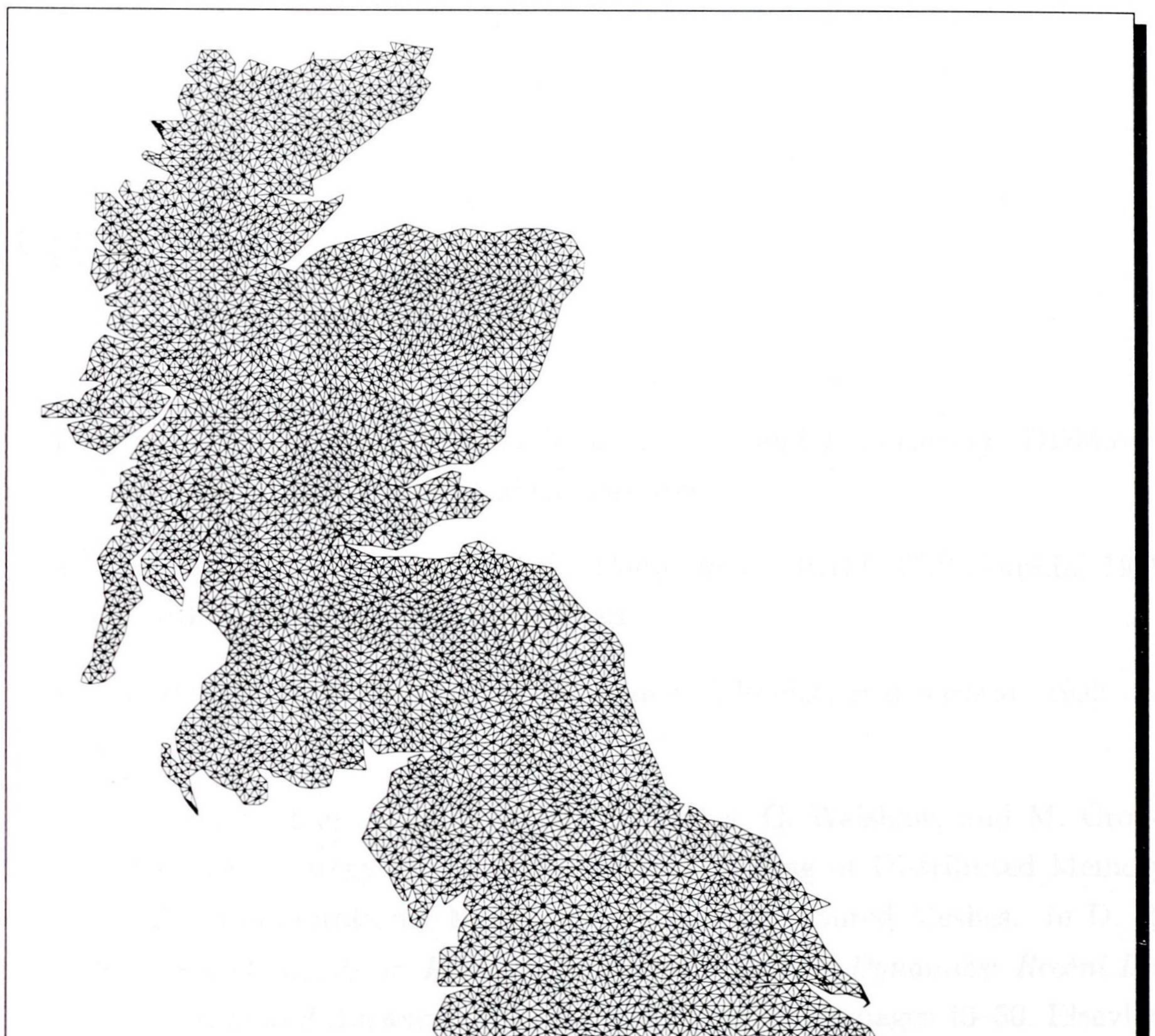


Figure B.18: Uk (final)

Bibliography

[1] *Domain Decomposition Methods in Sciences and Engineering.* DDM.org, 1987 onwards. Proceedings of Conferences.

[2] *Parallel Processing for Scientific Computing.* SIAM, Philadelphia, 1989 onwards. Proceedings of Conferences.

[3] *Parallel Computational Fluid Dynamics.* Elsevier, Amsterdam, 1992 onwards. Proceedings of Conferences.

[4] A. Arulananthan, S. Johnson, K. McManus, C. Walshaw, and M. Cross. A Generic Strategy for Dynamic Load Balancing of Distributed Memory Parallel Computational Mechanics Using Unstructured Meshes. In D. R. Emerson *et al.*, editor, *Parallel Computational Fluid Dynamics: Recent Developments and Advances Using Parallel Computers*, pages 43–50. Elsevier, Amsterdam, 1998. (Proc. Parallel CFD'97, Manchester, 1997).

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.22, Argonne National Laboratory, 1998.

[6] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. Tech. Rep. RNR-92-033, NASA Ames, Moffat Field, CA, 1992.

[7] J. K. Beddow and T. P. Meloy. *Testing and characterization of powders and fine particles.* Heyden, London, 1980.

[8] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transaction on Computers*, 36(5):579–580, 1987.

[9] M. Bern and D. Eppstein. Mesh Generation and Optimal Triangulation. In Hwang Du, editor, *Computing in Euclidean Geometry*, pages 23–90. Springer, 1992.

[10] R. Biswas and L. Oliker. Experiments with repartitioning and load balancing adaptive meshes. Nas-97-021, NASA Ames, Moffet Field, CA, 1997.

[11] S. Blazy, W. Borchers, and U. Dralle. Parallelization methods for chrateristic's pressure correction scheme. In E. Hirschel, editor, *Flow Simulation with High-Performance Computers II*, 1995. Notes on Numerical Fluid Mechanics.

[12] Bottasso, C. L. and Flaherty, J. E. and Özturan, C. Shephard, M. S. and Szymanski, B. K. and Teresco, J. D. and Ziantz, L. H. The Quality of Partitions Produced by an Iterative Load Balancer. In B.K. Szymanski and B. Sinharoy, editors, *Languages, Compilers, & Run-Time Systems for Scalable Computers*, pages 265–277, Reading, MA, 1996. Kluwer.

[13] I. Bronstein and K. Semendjajew. *Taschenbuch der Mathemetik.* B.G. Teubner Verlagsgesellschaft (Stuttgart, Leibzig) und Verlag Nauka (Moskau), 1991.

[14] P. Buch, J. Sanghavi, and A. Sangiovanni-Vincentelli. A Parallel Graph Partitioner on a Distributed Memory Multiprocessor. In *Proc. 5th IEEE Symp. on Frontiers of Massively Parallel Computation*, pages 360–366. IEEE, 1995.

[15] T.N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sisper. Graph Bisection Algorithms with Good Average Case Behaviour. *Combinatorica*, 7(2):171–191, 1987.

[16] Tony Chan and Victor Eijkhout. Design of a library of parallel preconditioners. Technical Report UCLA CAM report 97-58, University of California, Los Angeles, 1997.

[17] Tony Chan and Victor Eijkhout. Design of a library of parallel preconditioners. *Int. J. Supercomputer Appl.*, 2000.

[18] N. Chrisochoides, E. Houstis, and J. Rice. Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers. *J. Par. Dist. Comput.*, 21(1):75–95, 1994.

[19] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Par. Dist. Comput.*, 7(2):279–301, 1989.

[20] R Diekmann. *Load Balancing Strategies for Data Prallel Applications*. PhD thesis, Fachbereich Mathematik und Informatik, Universität–GH Paderborn, Germany, 1998.

[21] R. Diekmann, U. Dralle, F. Neugebauer, and T. Roemke. PadFEM: A portable parallel FEM-Tool. *HPCN-Europe*, pages 580–585, 1996.

[22] R. Diekmann, A. Frommer, and B. Monien. Efficient Schemes for Nearest Neighbor Load Balancing. Technical report, Fachbereich Mathematik und Informatik, Univ. Paderborn, Furstenallee 11, D-33102 Paderborn, Germany, May 1998.

[23] R. Diekmann, D. Meyer, and B. Monien. Parallel Decomposition of Unstructured FEM-Meshes. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, pages 199–215. Springer, 1995.

[24] R. Diekmann, B. Monien, and R. Preis. Using helpfull sets to improve graph bisections. Technical Report tr-rf-94-008, Fachbereich Mathematik und Informatik, Universität–GH Paderborn, Germany, 1994.

[25] R. Diekmann, B. Monien, and R. Preis. Using helpfull sets to improve graph bisections. *DIMACS Disc. Math. Th. Com. Sci*, 21:57–73, 1995.

[26] R. Diekmann, B. Monien, and R. Preis. Load Balancing Strategies for Distributed Memory Machines. In Karsch/Monien/Satz, editor, *Multi-Scale Phenomena and their Simulation*, pages 255–266. World Scientific, 1997.

[27] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Aspect Ratio for Mesh Partitioning. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing*, volume 1470 of *LNCS*, pages 347–351. Springer, 1998.

[28] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-Optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM. (submitted to Par. Comp.), 1999.

[29] R. Diekmann, Muthukrishnan S., and M. Nayakkankuppan. Engineering diffusive load balancing algorithms using experiments. In *Proc. IRREGULAR '97*, volume 1253 of *LNCS*, pages 111–122. Springer, 1997.

[30] R. Diekmann, F. Schlimbach, and C. Walshaw. Load Balancing for Parallel Adaptive FEM. In Bode et al. , editor, *Proceedings of the ALV'89*, SFB-Bericht 342/01/98 A, pages 41–52, Germany, 1998.

[31] R. Diekmann, F. Schlimbach, and C. Walshaw. Quality Balancing for Parallel Adaptive FEM. In A. Ferreira *et al.*, editor, *Proc. IRREGULAR '98: Solving Irregularly Structured Problems in Parallel*, volume 1457 of *LNCS*, pages 170–181. Springer, 1998.

[32] R. Diekmann and J. Simon. Verteilte Implementierung von Simulated Annealing. Diplomarbeit, Fachbereich Mathematik und Informatik, Universität–GH Paderborn, Germany, 1991.

[33] Dijkstra, Feijen, and van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 19:217–219, 1983.

[34] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel Algorithms for Dynamically Partitioning Unstructured Grids. In D. Bailey *et al*, editor, *Parallel Processing for Scientific Computing*, pages 615–620. SIAM, 1995.

[35] Victor Eijkhout and Tony Chan. ParPre: a parallel preconditioners package;reference manual for version 2.0.17. Technical Report CAM report 97-24, University of California, Los Angeles, 1997.

[36] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Comp. & Struct.*, 28(5):579–602, 1988.

[37] C. Farhat, S. Lanteri, and H. Simon. TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. *Computing Systems in Engineering*, 6(1):13–26, 1995.

[38] C. Farhat and M. Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *Int. J. Num. Meth. Engng.*, 36:745–764, 1993.

[39] C. Farhat, N. Maman, and G. Brown. Mesh partitioning for implicit computations via iterative domain decomposition: Impact and optimization of the subdomain aspect ratio. *Int. J. Num. Meth. Engng.*, 38:989–1000, 1995.

[40] C. Farhat, H. D. Simon, and Lanteri. TOP/DOMDEC – a Software Tool for Mesh Partitioning and Parallel Processing. *Computing Systems Engrg.*, 6(2):13–26, 1995.

[41] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, Piscataway, NJ, 1982.

[42] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[43] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[44] B. Ghosh, S. Muthukrishnan, and M. H. Schultz. Faster Schedules for Diffusive Load Balancing via Over-Relaxation. TR 1065, Department of Computer Science, Yale University, New Haven, CT 06520, USA, 1995.

[45] C. Goerdes. Inertial Partitionierung. Projektgruppe ParFem, Tool Documentation, 1995.

[46] B. Hendrickson and K. Devine. Dynamic Load Balancing in Computational Mechanics. (to appear in *Comput. Meth. Appl. Mech. Engrg.*), 1998.

[47] B. Hendrickson and R. Leland. The chaco user's guide, version 2.0. Tech. Rep. SAND94-2692, Sandia National Laboratories, October 1994.

[48] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. Supercomputing '95*, 1995.

[49] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM J. Sci. Stat. Comput.*, 16, 1995.

[50] Martin Hershoff. Numerische Finite–Elemente–Simulation einer 2D–Kanalströmung mit Hilfe von Mehrgittermethoden. Diplomarbeit Universität–GH Paderborn, 1998.

[51] D. Hodgson and P. Jimack. Efficient Mesh Patitioning for Parallel P.D.E. Solvers on Distributed Memory Machines. Report 93.1, School of Computer Studies, University of Leeds, January 1993.

[52] D. Hodgson and P. Jimack. A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids. Report 95.1, School of Computer Studies, University of Leeds, January 1995.

[53] D. Hodgson and P. Jimack. Efficient Mesh Partitioning for Parallel Elliptic Differential Equation Solvers. *Computing Systems in Engineering*, 6:1–12, 1995.

[54] D. Hodgson and P. Jimack. A Domain Decomposition Preconditioner for a Parallel Finite Element Solver on Distributed Unstructured Grids. *Parallel Computing*, 23:1157–1181, 1997.

[55] G. Horton. A multi-level diffusion method for dynamic load balancing. *PARCO*, 19(754):209–218, 1993.

[56] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. 1995. Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK.

[57] Y. F. Hu and R. J. Blake. The optimal property of polynomial based diffusion-like algorithms in dynamic load balancing. In K. D. Papailiou *et al.*, editor, *Computational Dynamics '98*, pages 177–183. Wiley, 1998.

[58] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice & Experience*, 10(6):467–483, 1998.

[59] Jan Hungershöfer. Parallele Algorithmen zur Verfeinerung zweidimensionaler Finite-Elemente-Netze. Diplomarbeit Universität–GH Paderborn, 1997.

[60] M. T. Jones and P. E. Plassmann. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In *Proc. Scalable High Performance Comput. Conf. '94*, pages 478–485. IEEE, 1994.

[61] G. Karypis, Schloegel K., and Kumar V. *PARMETIS, Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 2.0.* Computer Science Department, University of Minnesota and Army HPC Research Center, September 1998.

[62] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. TR 96-036, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1996.

[63] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In M. Heath *et al*, editor, *Parallel Processing for Scientific Computing.* SIAM, Philadelphia, 1997.

[64] G. Karypis and Kumar V. *METIS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Fill-Reducing Orderings of Sparse Matrices, Version 4.0.* Computer Science Department, University of Minnesota and Army HPC Research Center, September 1998.

[65] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, February 1970.

[66] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[67] P. Le Tallec, E. Saltec, and M. Vidrascu. Solving Large Scale Structural Problems on Parallel Computers using Domain Decomposition Techniques. In B. Topping and M. Papadrakakis, editors, *Advances in Parallel and Vector Processing for Structural Mechanics*, pages 127–132. Civil-Comp Press, 1994.

[68] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, 1992.

[69] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* B.G. Teubner, 1990.

[70] F. Lingen. A versatile load balancing algorithm for parallel applications based on domain decomposition. In B. Topping, editor, *Euroconference: Parallel and Distributed Computing for Computational Mechanics 1999.* Civil-Comp Press, 1999.

[71] R. Löhner, R. Ramamurti, and D. Martin. A Parallelizable Load Balancing Algorithm. AIAA-93-0061, American Institute of Aeronautics and Astronautics, Washington, DC, 1993.

[72] Message Passing Interface Forum. MPI: A Message-Pasing Interface standard. *Int. J. of Supercomputer Applications and High Performance Computing*, 8, 1994.

[73] D. Meyer. Datenparallele k-Partitionierung unstrukurierter Finite Elemente Netze. Diplomarbeit, Fachbereich Mathematik und Informatik, Universität-GH Paderborn, Germany, 1995.

[74] G. L. Miller, S. H. Teng, and S. A. Vavasis. A unified geometric approach to graph seperators. In *Proceedings of 32. Symp. on Foundat. of Comp. Science*, pages 538–547, 1991. FOCS'91.

[75] S. A. Mitchell and S. A. Vavasis. Quality Mesh Generation in Three Dimensions. In *Proc. of 8th ACM Conf. on Comp. Geometry*, pages 212–221, 1992.

[76] Leonid Oliker and Rupak Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *J. Par. Dist. Comput.*, 51(2):150–177, 1 August 1998.

[77] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *HPCN*, pages 493–498, April 1996.

[78] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.*, 11:430–452, 1990.

[79] R. Preis. Efficiant Partitioning of Very Large Graphs with the New and Powerfull Helpfull-Set Heuristic. Diplomarbeit, Fachbereich Mathematik und Informatik, Universität-GH Paderborn, Germany, 1994.

[80] R. Preis and R. Diekmann. The PARTY partitioning-library, user guide, version 1.1. Technical Report TR-RSFB-96-024, Universität Paderborn, 1996.

[81] R. Preis and R. Diekmann. PARTY - A software library for graph partitioning. In B.H.V. Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 63–71, 1997.

[82] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1988.

[83] E. Rehling. Datenstrukturen für adaptiv parallele Finite–Elemente–Methoden. Diplomarbeit Universität–GH Paderborn, 1998.

[84] Scali AS. *The ScaMPI User's Guide*, 1997.

[85] F. Schlimbach. Lokale Verbesserung. Projektgruppe ParFem, Tool Documentation, 1995.

[86] F. Schlimbach. Load Balancing Heuristics Optimising Subdomain Aspect Ratios for Adaptive Finite Element Simulations. Diplomarbeit Universität–GH Paderborn, 1998.

[87] F. Schlimbach, R. Diekmann, M. Cross, and Walshaw. C. Load Balancing for Aspect Ratio in Adaptive Finite Element Simulations. In B.H.V. Topping, editor, *Developments in Computational Mechanics With High Performance Computing*, pages 21–30. Civil-Comp Press, 1999. (Proc. Parallel & Distributed Computing for Computational Mechanics).

[88] K. Schloegel, G. Karypis, and V. Kumar. Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes. TR 97-014, Computer Science Department, University of Minnesota and Army HPC Research Center, 1997.

[89] K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. TR 98-034, Computer Science Department, University of Minnesota and Army HPC Research Center, 1998.

[90] J. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1994.

[91] H. D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[92] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. Technical report, Dep. of Math and Computer Science, Emory University, Atlanta, GA 30322, 1990.

[93] N. Touheed and P. Jimack. Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers. Report 96.34, School of Computer Studies, University of Leeds, December 1996.

[94] D. Vanderstraeten, C. Farhat, P. S. Chen, R. Keunings, and O. Zone. A Retrofit Based Methodology for the Fast Generation and Optimization of Large-Scale Mesh Partitions: Beyond the Minimum Interface Size Criterion. TR 94.62, Center for Systems Engineering and Applied Mechanics, Université Catholique de Louvain, B-1348, Louvain-la-Neuve, Belgium, 1994.

[95] D. Vanderstraeten, R. Keunings, and C. Farhat. Beyond Conventional Mesh Partitioning Algorithms and the Minimum Edge Cut Criterion: Impact on Realistic Applications. In D. Bailey *et al*, editor, *Parallel Processing for Scientific Computing*, pages 611–614. SIAM, 1995.

[96] D. Vanderstraeten, R. Keunings, and C. Farhat. Optimization of mesh partitions and impact on parallel CFD. In A. Ecer *et al*, editor, *Parallel Computational Fluid Dynamics: New Trends and Advances*, pages 233–239. Elsevier, Amsterdam, 1995. (Proceedings of Parallel CFD'93, Paris, 1993).

[97] C. Walshaw. Private Communication. 1997-99.

[98] C. Walshaw. *The Jostle user manual: Version 2.1.* University of Greenwich, London SE18 6PF, UK, March 1999.

[99] C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. Tech. Rep. 98/IM/35, University of Greenwich, London SE18 6PF, UK, 1998.

[100] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. Tech. Rep. 99/IM/44, Univ. Greenwich, London SE18 6PF, UK, February 1999.

[101] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. Tech. Rep. 98/IM/38, Univ. Greenwich, London SE18 6PF, UK, July 1998.

[102] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. *Int. J. Supercomputer Appl.*, 13(4):334–353, 1999.

[103] C. Walshaw, M. Cross, and M. Everett. A Localised Algorithm for Optimising Unstructured Mesh Partitions. *Int. J. Supercomputer Appl.*, 9(4):280–295, 1995.

[104] C. Walshaw, M. Cross, and M. Everett. A Localised Algorithm for Optimising Unstructured Mesh Partitions. *Int. J. Supercomputer Appl.*, 9(4):280–295, 1995. (originally published as Univ. Greenwich Tech. Rep. 95/IM/03).

[105] C. Walshaw, M. Cross, and M. Everett. A Parallelisable Algorithm for Optimising Unstructured Mesh Partitions. Tech. Rep. 95/IM/03, University of Greenwich, London SE18 6PF, UK, 1995.

[106] C. Walshaw, M. Cross, and M. Everett. Dynamic mesh partitioning: a unified optimisation and load-balancing algorithm. Tech. Rep. 95/IM/06, University of Greenwich, London SE18 6PF, UK, 1995.

[107] C. Walshaw, M. Cross, and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Par. Dist. Comput.*, 47(2):102–108, 1997. (originally published as Univ. Greenwich Tech. Rep. 97/IM/20).

[108] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph-partitioning for unstructured meshes. Tech. Rep. 97/IM/20, University of Greenwich, London SE18 6PF, UK, March 1997.

[109] C. Walshaw, M. Cross, M. Everett, S. Johnson, and K. McManus. Partitioning & Mapping of Unstructured Meshes to Parallel Machine Topologies. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.

[110] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice & Experience*, 3:457–481, 1991.

[111] R. D. Williams. Unification of spectral and inertial bisection. Technical report, California Institute of Technology, 1994.

[112] C. Z. Xu and F. C. M. Lau. Decentralized Remapping of Data Parallel Computations with the General Dimension Exchange Method. In *Proc. of 1994 Scalable High Performance Computing Conference, Knoxville*, pages 414–421. IEEE Computer Society Press, 1994.