Abstract

A second order space and time adaptive method for parallel solution of hyperbolic PDEs on structured grids is presented. The grid is adapted to the underlying solution by successive refinement in blocks. Therefore, there may be jumps in the cell size at the block faces. Special attention is needed at the block boundaries to maintain second order accuracy and stability. The stability of the method is proven for a model problem.

The step sizes in space and time are selected based on estimates of the local truncation errors and an error tolerance provided by the user. The global error in the solution is also computed by solving an error equation similar to the original problem on a coarser grid.

The performance of the method depends on the number of blocks used in the domain. A method of predicting the optimal number of blocks is presented. The cells are distributed in blocks over the processor set using a number of different partitioning schemes.

The method is used to successfully solve a number of test problems where the method selects the appropriate space and time steps according to the error tolerance.
Keywords: SAMR, Finite Volume Method, error control, GMRES, parallel computation, load balancing

Contents

A Implicit Solution of Hyperbolic Equations with Space-Time Adaptivity, *BIT*, 42:1, P. 128–153, 2002.

B A Comparison of Partitioning Schemes for a Blockwise Parallel SAMR Algorithm, IN T.SOREVIK, F.MANNE, R.MOE AND A. H. GEBREMEDHIN, *Applied Parallel Computing – New Paradigms for HPC in Industry and Academia*, SPRINGER-VERLAG. LECTURE NOTES IN COMPUTER SCIENCE, VOL. 1947, P. 160-169, HEIDELBERG, 2001.

C Optimal Parameter Values for a Parallel Structured Adaptive Mesh Refinement Algorithm, IN T.SOREVIK, F.MANNE, R.MOE AND A. H. GEBREMEDHIN, *Applied Parallel Computing – New Paradigms for HPC in Industry and Academia*, SPRINGER-VERLAG. LECTURE NOTES IN COMPUTER SCIENCE, VOL. 1947, P. 177-186, HEIDELBERG, 2001.

Chapter 1

Introduction

Numerical simulation of complex physical phenomena described by partial differential equations (PDEs) has become an interesting alternative to performing expensive and time consuming real-life experiments. Rebuilding and performing tests on real-life models requires much more effort compared to redesigning a computer model and running a numerical simulation.

Rapid development in both computer hardware and numerical methods has made it possible to solve realistic models of physically complex phenomenon. However, many problems still require massive amounts of computational resources in order to be solved accurately. Also, the quality of the obtained solution is not always known. Therefore, there is a great need to improve the efficiency of the numerical methods as well as to provide quantitative control of the numerical discretisation and solution errors.

Much of the recent efforts in the development of numerical methods has been focused on deriving different types of *adaptive methods* - a class of methods that utilizes the fact that, in many applications, the need for high resolution is confined to a small part of the computational domain. A typical class of problems suitable for adaptive solvers is of course flow problems. Take flow around an airfoil for instance, high resolution is mainly needed close to the airfoil, whereas the flow at some distance away can be solved with much less resolution.

Adaptive methods have become a commonly used approach to reducing computational complexity while maintaining or improving accuracy. However, adaptivity is not only used for reducing computational complexity, another very important aspect is automatic and quantitative error control - something that has been used for many years in the field of Ordinary Differential Equations (ODEs) but for PDEs, the techniques are much less matured.

The general idea is not to spend more resources on the problem than absolutely necessary, i.e. attain a user defined error tolerance while using as large time steps and space steps as possible. Typically, the solver starts with a coarse mesh, estimates the pointwise errors in the cells and then flags the points where the accuracy requirement is not attained. The flagged points are then refined in some way and the solution is recomputed with the new mesh. This process is repeated iteratively, thereby adapting the mesh to the underlying solution.

Of course, in some cases, an expert user could distribute the cells over the domain and choose a suitable time step but it is much more attractive to leave these choices up to the solver, thereby alleviating the user from having to make such decisions. Also, when solving time-dependent problems, features may move or develop in the solution during the simulation which introduces a need of dynamically altering the mesh and time step at run-time.

A number of different adaptive schemes have been presented over the years, many of them differ vastly in the way the grid is represented. This thesis will focus on methods for structured grids although the techniques developed here extend to unstructured grids, see [6]. For structured grids, a commonly used approach has been developed in [1], [3], [4], [5], [7], [20], [32], [33] where the grid is built up of a hierarchy of successively refined grid patches. Here, the adaptive process consists of three steps: flagging the cells in need of refinement, clustering the flagged cells and fitting grids over the clusters.

Another approach is to introduce cells into the original grid as shown in [8], [10], [19], [23], [25], [29], [30]. The grid could also be distorted to better fit the features in the solution as shown in [15], [22]. In most of the mentioned papers, either the steady-state solution was computed or a constant time step was used. Adaptivity for time-dependent problems is a more complicated matter.

In this thesis, an alternative approach is considered. The time accurate solution is computed using adaptivity both in space and in time. The grid is partitioned into a number of blocks, each containing a structured grid patch, see [12], [13], [14], [24], [28]. The grid is refined with respect to entire blocks, i.e. if one or more cells in a block is in need of refinement then the whole block is refined. Blocks are refined or coarsened based on information from the underlying solution, in particular a computed estimate of the local truncation error in the solution. The cell size may have jumps at the block boundaries but the jump is restricted to at most a factor of two across a block boundary. Special attention is needed to maintain stability and second order accuracy at the block boundaries.

The local errors in space are estimated by comparing the solution on two grids of different resolution and the local errors in time are estimated by comparing two different time stepping schemes. The time step control resembles the techniques used in solving ODEs.

This blockwise approach avoids the clustering and grid fitting steps needed in [3], [4], [5]. However, the granularity, or the number of blocks used in the grid, has an impact on the performance of the algorithm. Some suggestions as to how the granularity can be chosen automatically will be presented. Other parallel adaptive algorithms are found in [9], [38].

The code used in the numerical experiments is parallelized under the SPMD paradigm and uses the Message Passing Interface (MPI) for the communication. In the next chapter, the principles and parallel implementation of the adaptivity are discussed. Then the numerical scheme is described. In Chapter 4, the algorithms are presented in detail followed by a chapter on the dynamic load balancing problem. Finally, some numerical experiments are shown.

Chapter 2

Structured Adaptive Mesh Refinement

2.1 Overview and related work

This section discusses some of the common issues addressed in Structured Adaptive Mesh Refinement (SAMR) and gives a brief overview of some of the most commonly used approaches.

SAMR deals with the following simple problem:

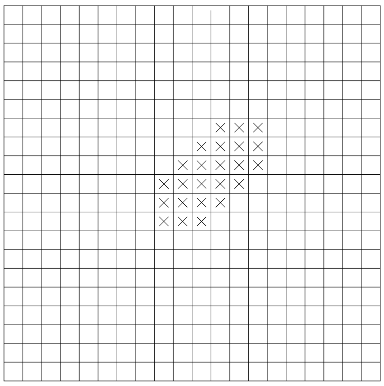

Figure 2.1: A grid with a set of flagged points where the accuracy is too low.

Consider Figure 2.1 - a grid with a set of grid points flagged for refinement. Now, what is the best way of refining the grid in the flagged region? There are a few issues that should be considered when developing an adaptive mesh refinement algorithm:

- *High cell efficiency* - refine as few non-flagged cells as possible.
- *Low storage overhead* - the data structures used to manage the grid should yield little storage overhead. Everything that is not solution data such as pointers, refinement level information etc, is considered to be storage overhead.
- *High computational performance* - when applying an integrator to the grid, the performance penalty compared to using a simple rectangular grid must be small. Also, it is beneficial if the data structure does not require severe modifications to the integrator itself.
- *Good parallel performance* - since many applications are required to run on parallel computers, the mesh structure should be suitable for parallelization.
- *Low overhead cost for administration* - the cost for administration of the mesh structure such as refining, coarsening, repartitioning should be low.
- *Cache performance* - to achieve a well performing code, the cache hierarchy of the machine must be utilized efficiently. Without at least some degree of locality in the storage of the solution data, this will become difficult.

Over the years a number of ways of addressing these issues have been proposed. The perhaps most intuitive way of adapting the grid would be to refine the grid cell by cell like in Figure 2.2, see [8], [10], [19], [29], [30]. In the sequel, this is referred to as *irregular mesh refinement*. This yields a highly flexible grid and it is very efficient in terms of cell consumption since in principle, only the flagged cells could be refined, resulting in 100% cell efficiency. In practise however, these methods typically introduce a buffer region of refined cells adjacent to the refined region. Otherwise, accuracy problems may occur at the boundary of the flagged region. This is the way unstructured grids are adapted to the solution.

The data structures used in this type of methods are usually associated with a storage overhead proportional to the number of cells. Since the solution data can not be stored in a simple array structure, the performance of the integrator is likely to be lower than for an integrator designed for a rectangular grid. Also, the data structure of the grid must be incorporated into the integrator making the migration from a non-adaptive code cumbersome. Since irregular mesh refinement destroys data locality, cache utilization is typically poor. This can be avoided to some extent by reallocating a whole patch of cells rather than just the cells being refined or coarsened.

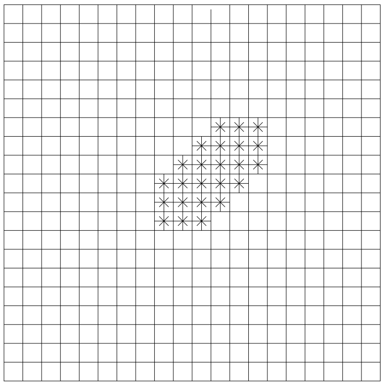

Figure 2.2: An irregularly refined grid. Only the flagged cells have been refined.

[3], [4] introduce a hierarchical grid by successively inserting a set of possibly rotated subgrids overlapping the flagged region, see Figure 2.3. Compared to irregular mesh refinement, this yields a very low storage overhead since the adjacency information is stored on subgrid level rather than on cell level. Hence, the storage overhead is proportional to the number of subgrids. On the other hand, cell efficiency can not be expected to be as high as for irregular mesh refinement. Since the grid is built from a set of rectangular grid patches, this approach benefits from performance and cache utilization properties similar to those of a uniform grid. It is also very straightforward to use an integrator designed for a rectangular grid since the grid patches themselves are rectangular.

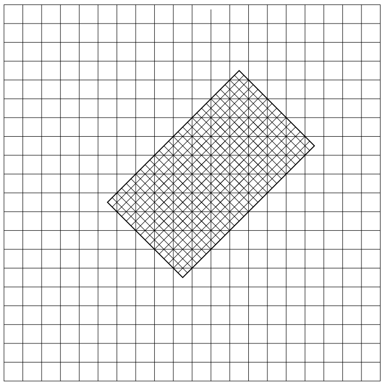

Figure 2.3: A hierarchical grid where the flagged cells have been superimposed by a rotated subgrid.

The choice whether to use rotated or non-rotated subgrids depends on the application. Using rotated subgrids can in some cases yield a higher cell efficiency but it also introduces an additional interpolation overhead when transferring data between the grids. Another concern is that subgrids on the same refinement level may overlap each other which can reduce cell efficiency. When global conservation is a consideration, rotated grids are rarely used since maintaining conservation typically becomes a complicated matter. In 1D the data structures used to store the grid hierarchy are fairly simple but there are several complications in 2D such as subgrids having more than one parent grid, subgrids on the same refinement level overlapping each other etc. Also, the process of determining which cells that go together in the same subgrid (clustering) and the size, position and orientation of the subgrids (grid fitting) are in general non-trivial.

Another class of methods is the moving grid point methods which basically contracts the flagged cells, thereby increasing the resolution in the flagged region at the expense of the rest of the domain, see [15], [22]. This type of method often has problems with controlling skewness of the cells, particularly in 3D. There is a fundamental difference compared to the previously discussed methods in that this method tries to achieve the maximum accuracy for a fixed cost whereas the other methods try achieve a fixed accuracy for a minimum cost.

2.2 A blockwise adaptive mesh

The approach considered in this thesis resembles the method proposed by Berger et al. in the sense that patches of rectangular grids are used. However, there are some significant differences, the main one being that there is no grid hierarchy where refined grids are inserted on top of the coarser grids. Instead, the domain is partitioned into a set of equally coarse patches or *blocks* on startup. The blocks can be refined or coarsened in order to adapt the grid to the underlying solution as it develops. Figure 2.4 shows an example of such a grid with three refinement levels present.

The number of blocks is constant during the whole simulation. The blocks are refined if at least one cell in the the block is flagged for refinement and coarsened if the error is sufficiently small in all the cells the entire block. This approach avoids the clustering and grid fitting steps used in the hierarchical approach. Also, when a coarse block is refined it is replaced rather than superimposed by a refined block which avoids the problem with unused cells in the refined regions.

However, the efficiency of the method depends on the granularity (the number of blocks). If too few blocks are used, the cell efficiency is likely to be low. Using a very large number of blocks strongly improves the cell efficiency but also introduces a significant overhead due to data migration at the block boundaries.

Choosing the optimal granularity can be a nontrivial task for an unexperienced user. Hence, this parameter is preferably set automatically by the code. Paper C gives a suggestion as to how the optimal granularity can be computed.

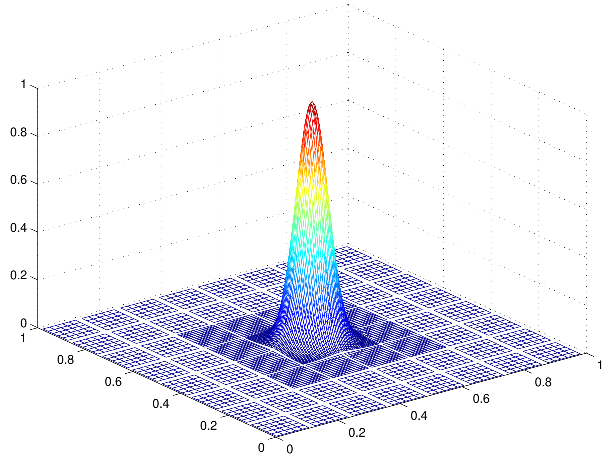

Figure 2.4: A blockwise SAMR grid generated for a model problem. Three refinemend levels are present.

Also, given an algorithm for computing the optimal granularity, a scheme for dynamically adapting the granularity at run-time can be devised.

2.3 Data structures

The code used in the numerical experiments is written in Fortran 90. This section discusses the data structures used to manage the blockwise adaptive grid on a parallel computer.

Recall that the grid consists of a number of blocks that can be refined or coarsened in order to adapt the grid to the underlying solution. The decision whether to refine or coarsen a block is based on information from the computed solution, in particular an estimate of the local truncation error in space, see Chapter 3. Blocks are refined or coarsened by halving or doubling the cell size. Hence, there may be jumps in the cell size at the block boundaries. In this implementation the jump is restricted to at most a factor of two. The spatial error is estimated by comparing the solution on two different grids or grid levels, one with twice the cell size in all directions. Do not confuse the different grid levels with the refinement levels of the individual blocks. Additional grid levels can be added and utilized for multigridding purposes as in [13] although this is not explored further in this thesis.

To simplify the approximation of spatial derivatives within a block, the arrays storing the solution data in the block are augmented by a layer of ghost cells. These cells overlap into neighbouring blocks and are updated by copying or interpolating data from the neighbour. A block with two layers of ghost cells is shown in Figure 3.1 in the next chapter.

A brief description of the derived types used in the code follows:

2.3.1 The coordinate type

The coordinate type handles all the coordinate information of the cells within a block. It stores either cell centers or cell vertices and cell face normals depending on whether a finite difference or a finite volume discretisation is used. The coordinate type has the following attributes:

1. Arrays storing coordinate information of the cells
2. The number of cells in each direction.

2.3.2 The solution type

The solution type is used to represent a patch of solution data at some grid or refinement level. The solution type has the following attributes:

1. The solution data
2. The number of cells in each direction
3. One instance of the coordinate type

2.3.3 The block type

The block type represents the blocks used to build the adaptive grid and they are also the nodes in the block graph described below. The block type has the following attributes:

1. ID of host process
2. Refinement level
3. Pointers to adjacent blocks
4. An array of solution type, storing the grid levels currently used.

2.3.4 The grid type

The grid type stores all the blocks in a graph structure, including the off processor blocks. One instance of the grid type is stored on each processor. However, blocks that are not local to the current processor do not store any solution or coordinate data - only the adjacency information and refinement level etc. Keeping a copy of the global graph on each processor greatly simplifies the administration of the grid and the storage overhead is small since the number of blocks is small compared to the number of cells. The graph remains constant during the simulation.
The grid type has the following attributes:

1. An array of the block type containing all the blocks in the grid
2. The global time step
3. The number of blocks in the grid

Figure 2.5 shows a block graph with 4×4 blocks stored on a processor, the highlighted blocks are local. The numbers in the blocks are the indices in the global graph.

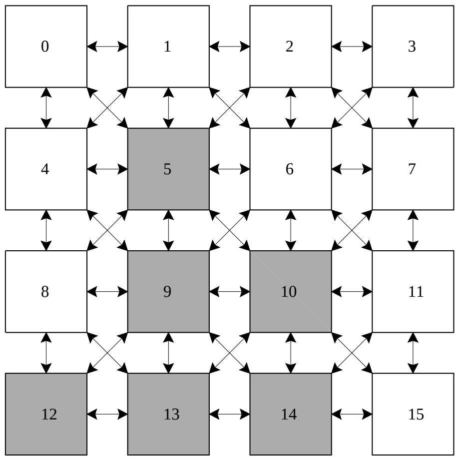

Figure 2.5: A 2D block graph with 4×4 blocks. The highlighted blocks are local to the current processor. The numbers in the blocks are the indices in the global graph.

2.4 Blocking overhead

Typically, the number of blocks used by the adaptive solver is greater than the number of processes it is run on. If too few blocks are used, there is not enough resolution in the block structure to allow the solver to adapt the grid in an efficient way.

On the other hand, the administration of a large number of blocks introduces a certain amount of overhead in the solver. This is mainly due to the communication needed between blocks on the same processor when updating the ghost cells. In an attempt to investigate the behavior of this overhead, two different types of experiments were carried out. In these experiments, no adaptivity is used and the solver is run on a single processor varying the number of blocks in two different ways.

2.4.1 Fixed mesh size

In the first case, the total number of cells in the mesh is kept constant as the number of blocks, nb, is increased, i.e. the number of cells in each block decreases as nb grows. The execution time is measured for a few different nb and then divided by the execution time for running the solver with the same mesh, partitioned into a single block. If $t(nb, N)$ is the execution time when running the solver on a single processor using a mesh with N cells partitioned into nb blocks, a measure can be defined as follows:

$$f_{mesh}(nb) = \frac{t(nb, N_{mesh})}{t(1, N_{mesh})} \tag{2.1}$$

In the experiment, N_{mesh} is kept constant as nb is varied.

2.4.2 Fixed block size

In the other case, the size of the blocks is kept constant, i.e. the total number of cells in the mesh increases as nb increases. Just as for the case with fixed mesh size, the execution time is measured for a few different nb and then divided by the execution time for running the solver with same mesh using only a single block. The measure for this case can be defined as follows:

$$f_{block}(nb) = \frac{t(nb, N_{block} \times nb)}{t(1, N_{block} \times nb)} \tag{2.2}$$

N_{block} is the number of cells in each block. In the experiment, N_{block} is kept constant as nb is varied.

The computational complexity in the solver will of course affect the influence of the overhead on the total execution time. Hence, the tests are performed solving two different equations - a simple scalar PDE and a more complex nonlinear system of PDEs. These examples are discussed more thoroughly in Chapter 6.

First, we consider a simple, scalar advection problem solved using a fourth order Runge-Kutta method (RK4). Here, the cost for computing the residual, i.e. the discretisation of the space derivatives, is fairly low. Second, we consider solving a nonlinear system of equations, in particular the Euler equations for inviscid, compressible flow, also using RK4. In this case, computing the residual in RK4 is a much more intensive operation in terms of arithmetic operations.

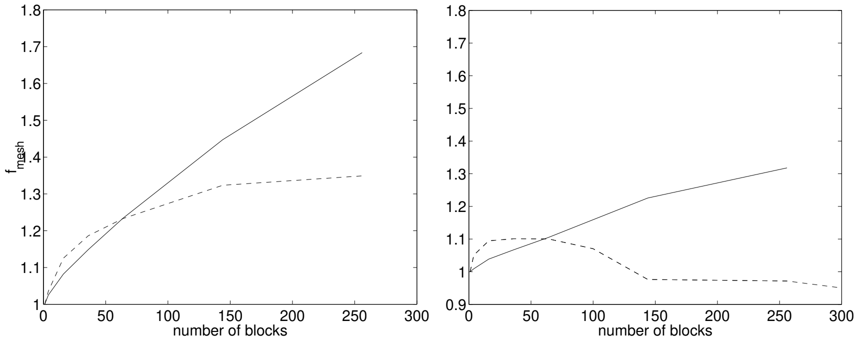

Figure 2.6: f_{mesh} (solid) and f_{block} (dashed) for the scalar advection problem (left) and the nonlinear problem (right).

Figure 2.6 shows f_{mesh} and f_{block} for the scalar problem and the nonlinear problem respectively as well as the execution time for the nonlinear problem as a function of granularity. As could be expected, the overhead plays a more significant role in the scalar problem.

In the nonlinear problem, which is closer to a realistic simulation in terms of computational complexity, the overhead is well behaved. In fact, the blocked code is actually faster when the problem is large, even though we are using a large number of blocks. This is occurs when the problem is too large to fit in cache. In such a situation, the code will run faster when the grid is partitioned into blocks since the blocking helps improve data locality. Entire blocks can be moved into cache and once the ghost cells are updated, the residual in that block can be computed without referring to cells in neighbouring blocks. This is illustrated in Figure 2.7 where a mesh of fixed size is used with a different number of blocks.

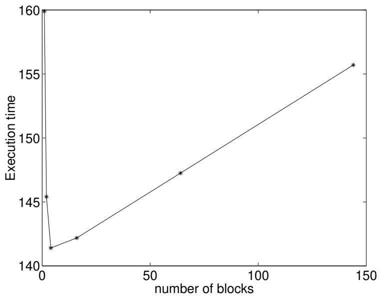

Figure 2.7: The execution time as a function of the number of blocks for the nonlinear problem.

Recall that these experiments where run without adaptivity to study the blocking overhead separately. When adaptivity is used, an increased number of blocks greatly improves the cell efficiency and thereby also the total computational load. Figure 2.8 shows the execution time as a function of granularity for a 2D fluid flow simulation. The granularity has a significant impact on the performance.

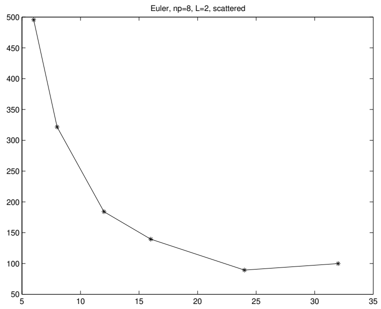

Figure 2.8: The execution time in seconds as a function of granularity $\sqrt{n_b}$ for a 2D fluid flow simulation. The granularity has a significant impact on the performance.

Paper C aims to derive an algorithm for predicting the optimal granularity based on a model of the execution time. The input is timings of communication and arithmetic work on a particular computer system and a typical refinement pattern. The experimental validation shows that the approach is promising but needs further studies. To fully automatize the approach, an inexpensive simulation using a very coarse grid can be run to gather information on the refinement pattern and to make the necessary measurements of communication and arithmetic costs.

Chapter 3

Discretisation

We wish to solve a hyperbolic PDE in conservative form

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} + \frac{\partial g(u)}{\partial y} = 0,\ (x, y, t) \in \Omega \times [0, t_E], \tag{3.1}$$

with $f(u) = P(x,y)u$ and $g(u) = Q(x,y)u$. Most of the numerical techniques here are applicable to (3.1) in its general form, but our theoretical results are restricted to the case when $f(u) = Pu$ and $g(u) = Qu$, where P and Q are constant scalars or matrices. In Paper A and in Chapter 6 we also solve a genuinely nonlinear problem.

The space derivatives are approximated by a finite volume method and the equations are integrated in time by a backward differentiation formula of second order (BDF-2), see [16]. The interpolation at block boundaries with different cell sizes is such that second order accuracy is achieved on Cartesian grids. We prove that the scheme is stable for a model equation.

3.1 Space and time discretisation

After integrating (3.1) over grid cell ω_{jk} of area $|\omega_{jk}|$ and using Gauss' theorem we obtain the equation for the cell average u_{jk}

$$\frac{\partial u_{jk}}{\partial t} + \frac{1}{|\omega_{jk}|} \int_{\partial \omega_{jk}} (f n_x + g n_y) dS = 0, \tag{3.2}$$

where $\partial \omega_{jk}$ is the cell boundary and (n_x, n_y) is the normal. The values of f and g are approximated on $\partial \omega_{jk}$ using the averages u_{jk} from adjacent cells.

We use two different approximations that are of second order on Cartesian grids with constant step sizes h_x and h_y in the x- and y-directions. On the face $(j+1/2, k)$ between the cells (j,k) and $(j+1,k)$, either we take

$$u_{j+1/2,k} = (u_{jk} + u_{j+1,k})/2, \tag{3.3}$$

yielding a centered difference approximation of $\partial u/\partial x$ when $P = 1$, or

$$u_{j+1/2,k} = (3u_{jk} - u_{j-1,k})/2 \text{ or } u_{j+1/2,k} = (3u_{j+1,k} - u_{j+2,k})/2, \qquad (3.4)$$

(depending on the direction of the flow) giving an upwind approximation. The y-direction is treated similarly.

The time derivative in (3.2) is discretised by the second order backward differentiation formula (BDF-2). The coefficients have to be modified when the time step changes during the simulation. This method is implicit and A-stable. One drawback of using an implicit method as opposed to an explicit method is that a system of equations has to be solved at every time step but for problems with different time scales, such as low Mach number flows, the time steps do not have to be adjusted to the fastest scale for stability.

The time step is also the same for all cells in the grid regardless of their size. An initial guess for the solution at t_n, given the solutions at t_{n-1} and t_{n-2}, is computed by an explicit predictor. Then u^n_{jk} is the solution to the corrector equation defined by BDF-2 with variable coefficients depending on the time steps $\Delta t_{n-j} = t_{n-j} - t_{n-j-1}$, $j = 0, 1$. This is described in more detail and in section 3.1.2 and in Paper A.

3.1.1 Space discretisation error

The local error is computed by comparing the restriction of the residual on the fine grid with the coarse grid residual evaluated at the restriction of the fine grid solution as in [14]. Let $\mathcal{F}_{jk}$ be the averaged integral in (3.2) and let $\Phi_{jk}(u_{jk})$ be its finite volume approximation in cell (j,k). The truncation error in space is then

$$\tau_{sjk} = \mathcal{F}_{jk} - \Phi_{jk}(u_{jk}). \qquad (3.5)$$

Let r be the restriction operator from the fine grid to the coarse grid, Φ_c and Φ_f be the coarse and fine grid residuals. Then the truncation error on the fine grid τ_s is estimated by

$$\tau_{sjk} = \frac{1}{2^p - 1}\left[r_{jk}(\Phi_{fmn}(u_{mn})) - \Phi_{cjk}(r_{jk}(u_{mn}))\right] \qquad (3.6)$$

where p is the order of the discretisation. Note that the resolution of τ_s is the same as the coarse grid resolution.

An alternative would be to compare with a third or fourth order method. However, to implement such a scheme requires some effort and since the local error is controlled by a change in step size, it is a natural test to see the effect of such a change by computing the space discretisation for two different resolutions. The local error in time is estimated in the next section.

3.1.2 Time discretisation error

The time derivatives are approximated by the second order backward differentiation formula (BDF-2). The time step is selected in a manner similar to the techniques for ordinary differential equations [16]. The coefficients of the method have to be modified as the time step is adapted during the simulation. We want to approximate u^n by $\Phi(u^n)$ and the known solutions u^{n-1} and u^{n-2} at t_{n-1} and t_{n-2}. The time step is defined by $\Delta t_i = t_i - t_{i-1}$. Using a Taylor expansion of u at t_n and cancelling terms of order less than 3 in the time step, we obtain a second order approximation with variable coefficients given by

$$\begin{aligned}
&\alpha_0^n u^n = \Delta t_n \Phi(u^n) - \alpha_1^n u^{n-1} - \alpha_2^n u^{n-2},\\
&\theta^n = \Delta t_n/\Delta t_{n-1},\\
&\alpha_0^n = (1+2\theta^n)/(1+\theta^n),\ \alpha_1^n = -(1+\theta^n),\ \alpha_2^n = (\theta^n)^2/(1+\theta^n).
\end{aligned} \tag{3.7}$$

With $\Delta t = \max(\Delta t_{n-1}, \Delta t_n)$ the local error is

$$\begin{aligned}
&u(t_n) - u^n = c_I(\theta^n)\Delta t_n^3 u''' + O(\Delta t^4),\\
&c_I(\theta^n) = -(1+\theta^n)^2/(6\theta^n(1+2\theta^n)).
\end{aligned} \tag{3.8}$$

The leading term in the local error in time in (3.8) is $c_I(\theta^n)\Delta t_n^3 u'''$. It is estimated by comparing u^n computed by BDF-2 in (3.7) with $\tilde{u}^n$ determined by an explicit two step method with a different c_I but still using u^{n-1} and u^{n-2} ('Milne's device'). The difference $u^n - \tilde{u}^n$ gives an estimate of the leading term of the local error. The solution $\tilde{u}^n$ also serves as the initial guess in the iterative solution of (3.7). Another possibility would be to compute u^n with two different time steps and then compare the results.

Here we choose the following explicit method as predictor:

$$\begin{aligned}
&\tilde{\alpha}_0^n \tilde{u}^n = \Delta t_n \Phi(u^{n-1}) - \tilde{\alpha}_1^n u^{n-1} - \tilde{\alpha}_2^n u^{n-2},\\
&\tilde{\alpha}_0^n = 1/(1+\theta^n),\ \tilde{\alpha}_1^n = \theta^n - 1,\ \tilde{\alpha}_2^n = -(\theta^n)^2/(1+\theta^n),
\end{aligned} \tag{3.9}$$

with the local error

$$\begin{aligned}
&u(t_n) - \tilde{u}^n = c_P(\theta^n)\Delta t_n^3 u''' + O(\Delta t^4),\\
&c_P(\theta^n) = (1+1/\theta^n)/6.
\end{aligned} \tag{3.10}$$

If $\partial\Phi/\partial u$ is of $O(1)$, then we infer from (3.7), (3.8), and (3.10) that the leading term in the discretisation error in the BDF-2 approximation of u' in (3.12) is

$$\tau_t^n = -\alpha_0^n c_I \Delta t_n^2 u''' = \alpha_0^n c_I (u^n - \tilde{u}^n)/(\Delta t_n(c_I - c_P)). \tag{3.11}$$

The first step from $t_0 = 0$ to $t_1 = \Delta t_1$ is computed by BDF-1 or the backward Euler method with $\alpha_0^1 = 1$, $\alpha_1^1 = -1$ and $\alpha_2^1 = 0$.

3.1.3 Error equation

Local error estimates provide a good basis for developing grid and time step adaption schemes. However, it is really the global error in the solution that is of interest. Using the estimates of the local errors, an estimate of the global error can be computed by solving an error equation similar to (3.1). The error e in the solution approximately satisfies the following differential equation, see [14],

$$\frac{\partial(u+e)}{\partial t} + \frac{\partial f(u+e)}{\partial x} + \frac{\partial g(u+e)}{\partial y} = \tau_t + \tau_s.$$

Taylor expansion around u and dropping the nonlinear terms in e gives the error equation for (3.1) as

$$\frac{\partial e}{\partial t} + F(u)\frac{\partial e}{\partial x} + G(u)\frac{\partial e}{\partial y} = \tau_t + \tau_s \tag{3.12}$$

where

$$F(u) = \frac{\partial f}{\partial u} \text{ and } G(u) = \frac{\partial g}{\partial u} \tag{3.13}$$

The error equation is discretised and solved in the same manner as (3.1) on the next coarser grid and only at every second time step. In two space dimensions, this introduces a storage overhead of about 25% and a computational overhead of about 12% compared to solving (3.1). The overhead is even smaller in three dimensions.

3.2 Interpolation at block boundaries

When the computational grid is generated, it is partitioned into blocks. In order to simplify the space discretisation in cells at a block boundary, each block has two rows of ghost cells overlapping the adjacent block, see Figure 3.1.

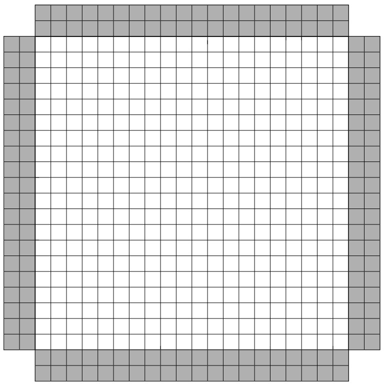

Figure 3.1: A block with 20×20 cells and two layers of ghost cells (highlighted).

We allow a discontinuous change of the step size at a block boundary. In our implementation, the jump in step size, if any, is always a factor of two, so that two cells in a fine block share a face with one cell in a coarse neighbouring block. This is illustrated in Figure 3.2.

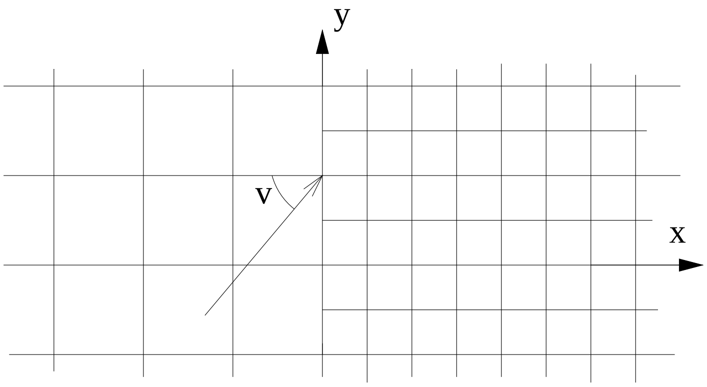

Figure 3.2: A block boundary with a jump in the step size.

In a cell (j, k) with at least one face on a block boundary with a jump in the step size, the space derivatives are approximated by an upwind calculation of $f_{j+1/2,k}$ as in (3.4). In all other cells, the centered approximation (3.3) is used. The change of stencil at the boundary is necessary for stability there. The reason why the centered scheme is preferred in the interior is that it has a smaller local error.

If the cells are of the same size at a block boundary, then the variables in the ghost cells are copied from the adjacent block. If there is a jump in the step size across a boundary, then interpolation is necessary. The value in a ghost cell in the coarse block is computed by area weighted averaging. The variables in the ghost cells in the fine grid have to be interpolated from available coarse grid values. In Paper A, an interpolation is devised which provides second order accuracy in the derivative approximation at a block boundary. Only variables from the coarse grid are used to update the ghost cells in the fine grid.

It seems to be important for the stability of the solution at the block boundaries that the upwind discretisation is combined with the one-sided interpolation. In the next section, we reproduce a proof from [18] where our discretisation (including interpolation at a block boundary) is proved to be stable for a convection equation. The same interpolation is utilized for calculation of the steady state and time-dependent solutions of the Euler equations with an upwind scheme in Paper A and [13]. The stability at interfaces is analyzed in [2]. One-sided approximations at the outer boundaries are proved to be stable in [36].

3.3 Stability at block boundaries

In Paper A, the stability of the scheme is shown experimentally. Here, the stability of the discretisation is investigated for the following scalar model problem:

$$\begin{array}{l} u_t + \cos(v)u_x + \sin(v)u_y = 0, \\ x_{min} \le x \le x_{max},\ y_{min} \le y \le y_{max},\ v \in [-\pi/2, 3\pi/2), \\ u(x, y_{min}) = u(x, y_{max}). \end{array} \tag{3.14}$$

A partial derivative with respect to x, y, or t is denoted by a subscript. If $-\pi/2 \le v \le \pi/2$ then $u(x_{min}, y) = u_0(y)$. Otherwise, $u(x_{max}, y) = u_0(y)$. Equation (3.14) is discretised with the method in the previous section on a grid with one block boundary, where the step size is h/ν, $\nu \in \mathbb{Z}^+$, in the fine grid and h in the coarse grid. That is, in Figure 3.2 we have $\nu = 2$.

After space discretisation, equation (3.14) can be written as

$$U_t + R(v)U = f,\ U, f \in \mathbb{R}^N,\ R \in \mathbb{R}^{N\times N}, \tag{3.15}$$

where N is the total number of cells and f is determined by the boundary conditions. The unknown variables in U are ordered with increasing index from the left to the right in each row and from the bottom to the top row (see Figure 3.2). The ordering is the same in the right block. In the boundary cells at $x = x_{min}$ and $x = x_{max}$, we use an upwind approximation to u_x.

The time integration in (3.15) is stable if the eigenvalues λ_j of $R(v)$ satisfy $\Re\lambda_j \ge 0$: we will now show that this is the case. We partition R into blocks R_{ll} and R_{rr} corresponding to the n_l and n_r cells and unknowns in the left and right blocks, respectively, so that

$$R = \begin{pmatrix} R_{ll} & 0 \\ R_{rl} & R_{rr} \end{pmatrix},\ R_{ll} \in \mathbb{R}^{n_l\times n_l},\ R_{rr} \in \mathbb{R}^{n_r\times n_r}. \tag{3.16}$$

When $-\pi/2 \le v \le \pi/2$, the upper right block in R is zero, because the discretisation in the left block is independent of the right block variables. The eigenvalues of R_{ll} and R_{rr} are also eigenvalues of R.

Since the boundary conditions are periodic in the y direction, the solution can be represented by Fourier modes. We obtain for each row k

$$U_{jk} = u_j(\omega)\exp(i\omega y_k),\ y_k = y_{min} + (k-1/2)h,\ j = 1,\dots,n_x,\ k = 1,\dots,n_y, \tag{3.17}$$

for n_y discrete values ω. Suppose that we have n_x cells in each row and n_y rows in the left block. Then, for $-\pi/2 \le v \le \pi/2$ and $1 \le k \le n_y$, the x-derivative is approximated by

$$\begin{array}{lll} j = 1 & : & (2h)^{-1}(3U_{1k} - 4U_{0k} + U_{-1,k}) \\ & & = (2h)^{-1}(3u_1 - 4u_0 + u_{-1})\exp(i\omega y_k), \\ 1 < j < n_x & : & (2h)^{-1}(U_{j+1,k} - U_{j-1,k}) = (2h)^{-1}(u_{j+1} - u_{j-1})\exp(i\omega y_k), \\ j = n_x & : & (2h)^{-1}(3U_{n_x,k} - 4U_{n_x-1,k} + U_{n_x-2,k}) \\ & & = (2h)^{-1}(3u_{n_x} - 4u_{n_x-1} + u_{n_x-2})\exp(i\omega y_k). \end{array} \tag{3.18}$$

The boundary condition defines u_0 and u_{-1}. The space discretisation for the cells in one row of the left grid can be written as

$$\cos(v)(2h)^{-1}Su\exp(i\omega y_k) + \sin(v)ih^{-1}\sin(\omega h)u\exp(i\omega y_k) - f_k, \tag{3.19}$$

where

$$S = \begin{pmatrix} 3 & & & & & \\ -1 & 0 & 1 & & & \\ & -1 & 0 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 0 & 1 \\ & & & 1 & -4 & 3 \end{pmatrix} \in \mathbb{R}^{n_x \times n_x}, \tag{3.20}$$

and f_k contains the boundary terms.

Let $\lambda_j(S), j = 1 : n_x$, be the eigenvalues of S in (3.20). Then it follows from (3.19) that the eigenvalues of R_{ll} are

$$\begin{array}{l} \lambda_{jk}(R_{ll}) = \cos(v)(2h)^{-1}\lambda_j(S) + i\sin(v)h^{-1}\sin(\omega_k h), \\ j = 1 : n_x,\ k = 1 : n_y. \end{array} \tag{3.21}$$

The discretisation in the right block is as in (3.18) and for one row of cells, the eigenvalues are as in (3.21) with h replaced by $h/2$. The eigenvalues of R_{rr} have the same form as in R_{ll} in (3.21).

If the wind blows in the opposite direction, $v > \pi/2$, then the structure of R is

$$R = \begin{pmatrix} R_{ll} & R_{lr} \\ 0 & R_{rr} \end{pmatrix}, \tag{3.22}$$

where the eigenvalues of R are still the eigenvalues of R_{ll} and R_{rr}. Furthermore, the approximation of the x derivative with the upwind scheme in the left block is (cf. (3.18))

$$\begin{array}{lcl} j=1 & : & -(2h)^{-1}(3U_{1k}-4U_{2k}+U_{3k}), \\ 1<j<n_x & : & (2h)^{-1}(U_{j+1,k}-U_{j-1,k}), \\ j=n_x & : & -(2h)^{-1}(3U_{n_x,k}-4U_{*k}+U_{\star k}), \end{array}$$

where U_{*k} and $U_{\star k}$ are data interpolated from the right block. By applying the difference stencil to U_{jk} in (3.17) we arrive at

$$\cos(v)(2h)^{-1}Tu\exp(i\omega y_k)+\sin(v)ih^{-1}\sin(\omega h)\exp(i\omega y_k)-f_k. \tag{3.23}$$

Here, T is a permutation of S with the opposite sign

$$T=-ZSZ,\ Z_{jk}=\delta_{j,n_x+1-k},\ j,k=1:n_x, \tag{3.24}$$

where δ_{jk} is the Kronecker tensor. The eigenvalues of T are

$$\lambda_j(T)=-\lambda_j(S),\ j=1:n_x, \tag{3.25}$$

and as in (3.21)

$$\lambda_{jk}(R_{ll})=-\cos(v)(2h)^{-1}\lambda_j(S)+i\sin(v)h^{-1}\sin(\omega_k h). \tag{3.26}$$

The eigenvalues of S have the following property:

Lemma 2.1

The eigenvalues $\lambda_j(S)$ of S in (3.20) satisfy $\Re\lambda_j\geq 0$ for $j=1:n_x$.

Proof

We follow the ideas in the proof of lemma 6.1 in [27]. The matrix S has one 1×1 block with eigenvalue 3. If $n_x=1$ then $\lambda_1=3$ and if $n_x=2$ then $\lambda_1=\lambda_2=3$. If $n_x>2$, then for the big, lower block in (3.20) with $K=n_x-1$ we make the ansatz for the eigenvector $z\in\mathbb{C}^K$

$$z_j=a\kappa_1^j+b\kappa_2^j,\ j=1:K.$$

The difference equation for $z_j,\ j=1:K-1$ is

$$z_{j+1}-\lambda z_j-z_{j-1}=0.$$

It therefore follows that

$$\kappa_k^2-\lambda\kappa_k-1=0,\ k=1,2, \tag{3.27}$$

$\kappa_1\kappa_2 = -1$, and that the corresponding eigenvalue is

$$\lambda = \kappa_1 - 1/\kappa_1 = \kappa_2 - 1/\kappa_2.$$

At $j = 0$ and $j = K$ we have

$$z_0 = a + b = 0, \tag{3.28a}$$
$$(3-\lambda)z_K - 4z_{K-1} + z_{K-2} = a((3-\lambda)\kappa_1^K - 4\kappa_1^{K-1} + \kappa_1^{K-2}) \tag{3.28b}$$
$$+ b((3-\lambda)\kappa_2^K - 4\kappa_2^{K-1} + \kappa_2^{K-2}) = 0. \tag{3.28c}$$

If (3.27) has a double root it has to be $\kappa_1 = \kappa_2 = \pm i$. Then $z_j = (a + jb)\kappa^j$ and since $z_0 = 0$ we have $a = 0$. Inserting z_j into (3.28c) we find that $b = 0$. The conclusion is that $z = 0$. This is not an eigenvector, so (3.27) cannot have a double root.

For a nontrivial solution (a, b) to exist, the determinant of the system matrix S in (3.28) must vanish. Let $\kappa = \kappa_1 = -1/\kappa_2$ to obtain a polynomial equation for κ:

$$\det S = \kappa^{-(K+1)}(\kappa^{2K-1}(\kappa-1)^3 + (-1)^K(\kappa+1)^3) = 0.$$

The equation for κ is

$$\kappa^{2K-1} = (-1)^{K+1}\left(\frac{\kappa+1}{\kappa-1}\right)^3. \tag{3.29}$$

Let $\kappa = r\exp(i\phi)$ and consider the modulus of both sides of (3.29)

$$r^{2K-1} = \left(\frac{(r\cos\phi + 1)^2 + r^2\sin^2\phi}{(r\cos\phi - 1)^2 + r^2\sin^2\phi}\right)^{3/2}. \tag{3.30}$$

Let p_l be the left hand side and p_r the right hand side of (3.30). For $-\pi/2 \leq \phi \leq \pi/2$ and $r < 1$, we have $p_l < 1$ and $p_r \geq 1$ and (3.29) has no solution. Also, for $\pi/2 < \phi < 3\pi/2$ and $r > 1$, $p_l > 1$ and $p_r < 1$ and there is no solution to (3.29) (see the shaded area in Figure 3.3). The κ-domain without a solution to (3.29) is mapped by $\kappa - 1/\kappa$ to the left half plane of the λ-plane. Hence, $\Re\lambda \geq 0$.

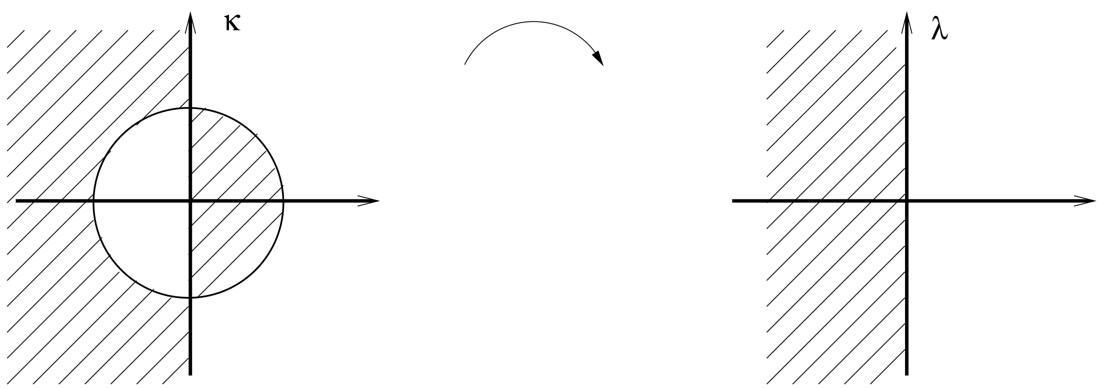

Figure 3.3: The map $\lambda = \kappa - 1/\kappa$ from the κ-plane to the λ-plane.

We are now ready to prove the following theorem:

Theorem 2.2 *The space-time discretisation of (3.14) described in sections 3.1 and 3.2 with one coarse block (step size h) and one fine block (step size h/ν) for any integer $\nu \geq 2$ is stable.*
Proof We conclude from (3.21) and (3.26) that

$$\begin{array}{lcl} -\pi/2 \leq v \leq \pi/2 & : & \left\{ \begin{array}{l} \Re\lambda(R_{ll}) = \cos(v)(2h)^{-1}\Re\lambda(S_l) \\ \Re\lambda(R_{rr}) = \cos(v)h^{-1}\Re\lambda(S_r) \end{array} \right. \\ \pi/2 < v < 3\pi/2 & : & \left\{ \begin{array}{l} \Re\lambda(R_{ll}) = -\cos(v)(2h)^{-1}\Re\lambda(S_l) \\ \Re\lambda(R_{rr}) = -\cos(v)h^{-1}\Re\lambda(S_r) \end{array} \right. \end{array}$$

The dimensions of S_l and S_r in (3.20) are determined by the number of cells in one row of the left and right blocks, respectively. It follows from Lemma 2.1 that

$$\Re\lambda(R) \geq 0,\ v \in [-\pi/2, 3\pi/2).$$

Since the time discretisation is A-stable and the imaginary axis is inside the stability region except at 0, the time stepping procedure is stable.■

Remark We have nowhere made any assumption about how many cells in the fine block meet one cell in the coarse block. Thus, the results are independent of ν. The conclusions are also independent of whether the fine block is located to the right or to the left: both cases are covered by changing v. In addition, the interpolation of data in the ghost cells does not affect the stability as long as it is one-sided, i.e. the stencil involves variables in only one block. If $\nu = 1$, then the grid can be treated as one single block with one S-matrix.

Chapter 4

Adaptive algorithm

The grid and time step adaption algorithms are designed for parallel execution. Recall that every processor stores a copy of the global graph containing all the blocks in the grid. However, only blocks local to the processor stores solution and coordinate data. The *root process* or the process running on processor 0, performs some additional administrative work compared to the other processes.

When a feature in the solution is moving across a block boundary into a coarser adjacent block, small reflections may occur. This can be avoided by refining the adjacent block to at least the same refinement level as the current block prior to the transition.

This can be regarded as a look-ahead strategy refining the adjacent block slightly in advance instead of waiting for the feature to cross the boundary. It is achieved by introducing boundary regions at every edge of the block and monitoring the error in those regions individually. A boundary region includes the cells along a face of the block and also the cells within a certain distance away from the face. The width of the region is determined as a certain percentage of the total width of the block. Such a boundary region is shown in Figure 4.1. In the experiments in Chapter 6, the width of the boundary region is 20% of the width of the block.

If the error is considered substantial in a boundary region and if the neighbouring block in that direction is at a coarser refinement level, then that neighbour is flagged for refinement. The threshold for the boundary regions is set to a certain fraction of the error tolerance.

The adaptive algorithm described in Algorithm 4.4 can be built from a few basic operations described in Algorithms 4.1-4.3. At startup, the user specifies an error tolerance ϵ on the maximum norm of the discretisation errors in space and in time, τ_s and τ_t as described in 3.1.1 and 3.1.2. The grid is adapted according to Algorithm 4.1

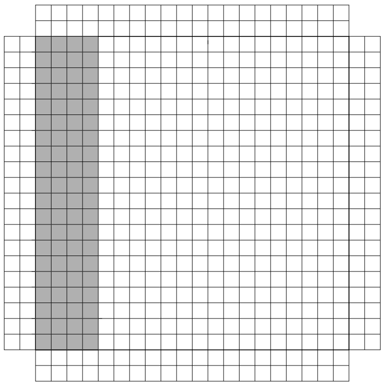

Figure 4.1: An example of a boundary region used in the look-ahead strategy.

Algorithm 4.1 - grid adaption

1. Compute the estimate of the spatial error τ_s according to (3.6) in all cells in the local blocks.
2. Mark the blocks where $max_{jk} |\tau_{sjk}| > \epsilon$ for refinement and the blocks where $max_{jk} |\tau_{sjk}| < 0.1\epsilon$ for coarsening.
3. Check the boundary regions Γ_i (see Figure 4.1) in every direction of the local blocks. If $max_{jk \in \Gamma_i} |\tau_{sjk}| > 0.6\epsilon$ in the region and the neighbour block in that direction is at a coarser level, mark the neighbour for refinement.
4. Gather all the proposed grid changes in the root process.
5. The root process determines the new refinement levels for each block in the grid, not just the local blocks. All blocks marked for refinement must be refined but blocks marked for coarsening are only allowed to be coarsened if the resulting jump in cell size over each block boundary is at most a factor of two. (4.1)
6. The root process broadcasts the accepted grid changes, if any, to all the processors which then execute their local refinements and updates the information in the off processor blocks in their copy of the block graph.
7. If necessary, compute a new partitioning and repartition the grid.

The ghost cells in the blocks are updated using Algorithm 4.2. MPI is used for the interprocessor communication.

Algorithm 4.2 - interblock communication

1. For all local blocks: check the processor id and refinement level of each neighbouring block. If the neighbour is local, just copy the interpolated data to the neighbour, otherwise pack the interpolated data, destination block id and block face id to a buffer corresponding to the processor id of the neighbour.
2. If not empty, send the buffers to their destination processors.
3. Receive buffers from other processors and unpack the data to the destination block faces.

(4.2)

The number of blocks used to build the grid is either specified by the user or computed according a method such as the one described in Paper C. A grid partitioning algorithm is then applied to the block graph to determine the initial distribution of the blocks. The initial refinement levels of the blocks are determined by Algorithm 4.3.

Algorithm 4.3 - finding the initial grid

1. Create the block graph and allocate all the local blocks.
2. Compute the initial solution.
3. Adapt the grid using Algorithm 4.1. If any blocks were changed, Goto 1 else Goto 4.
4. Start the simulation using Algorithm 4.4

(4.3)

The time steps are adapted as follows. If τ_t is the estimate of the temporal error and ϵ is the error tolerance, then let $\zeta = \sqrt{\epsilon / max_{jk} |\tau_{tjk}|}$. If $1 < \zeta < 1.15$, the step is accepted and the solution is advanced to the next time level without changing the time step. If $\zeta < 1$ then the step is rejected, the time step is reduced according to $\Delta t_n = 0.9\zeta\Delta t_n$ and the step is recomputed with the new time step. If $\zeta > 1.15$ then the time step is increased according to $\Delta t_n = 1.1\Delta t_n$ and used in the next time step. The solution is advanced from t_{n-1} to t_n using Algorithm 4.4. In the implicit time step, a linear system of equations has to be solved for u^n. A parallel version of GMRES [31] has been implemented for this purpose.

Algorithm 4.4 - advancing the solution from t_{n-1} to t_n

1. Adapt the grid using Algorithm 4.1
2. Compute a new partitioning and redistribute the blocks if necessary.
3. Take a predictor step using the explicit method (3.9)
4. Take a corrector step using the implicit method (3.7). The system is solved with GMRES or nonlinear GMRES with the predictor solution as the initial guess.
5. Compute the estimate of the temporal error according to (3.11)
6. If $\zeta < 1$, $\Delta t_n = 0.9\zeta\Delta t_n$, Goto 3.
   Else if $\zeta > 1.15$, $\Delta t_{n+1} = 1.1\Delta t_n$, Goto 7.
   Else Goto 7.
7. Advance the solution to the next time level, $t_n = t_{n-1} + \Delta t_n$

(4.4)

Chapter 5

Load balancing

Since the grid is repeatedly adapting to the underlying solution, even a good initial load balance is likely to deteriorate as the simulation progresses. Therefore, the partitioning of the grid must be updated dynamically as the grid is adapted. Dynamic partitioning differs from static partitioning in a number of ways. Speed becomes a key issue since the algorithm is applied frequently during the simulation. Data migration costs for redistributing the grid from the current partitioning to the new partitioning must also be taken into account. Thus, a small change in load should yield a small change in the data distribution.

The increase in administration cost and implementational complexity associated with partitioning the adaptive grid on a cell by cell basis is considered to be too high. Therefore, the grid is partitioned by distributing entire blocks over the processor set. This way, the partitioning scheme can be applied to the block graph instead of the entire grid, significantly reducing the cost for computing a new partitioning. However, one can expect a slight decrease in partitioning quality depending on the granularity of the blockwise grid. Figure 5.1 shows an example distribution of a grid with 16×16 blocks and three grid levels computed with the Recursive Spectral Bisection algorithm. The different colours represent the processor assignment of the blocks.

The performance of a particular partitioning scheme depends on the behaviour of both the application and the computer system. Certain applications may even experience a change in behaviour at different stages of the simulation. This is a motivation for introducing a dynamically adaptive meta-partitioner, i.e. a partitioner that uses different partitioning schemes depending on the current behaviour of the application. Paper B aims to derive such a meta-partitioner by making a classification of the partitioning behaviour based on the tuple Partitioning, Application and Computer system (PAC) and choosing the appropriate partitioning scheme accordingly. The application and computer system are classified according to a quadrant approach, see Figure 5.2. The classification is made with respect to (a) the amount of communication (communication dominated or computation dominated) and to (b) the distribution of the refinements (localized or scattered). Due to the dynamic nature of adaptive applications and multiuser computer systems, an application may start in one quadrant and move to other quadrants as the simulation progresses. Paper B evaluates the performance of a selection of partitioning schemes from ParMETIS [21] and Vampire [35] and outlines policies for selecting partitioner depending on the classification. Experiments show that the quadrant approach to classification is not powerful enough. Additional parameters must be included in the model in order to capture the nature of parallel, blockwise SAMR applications.

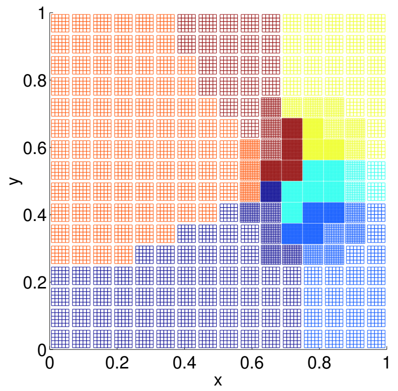

Figure 5.1: A grid with 16×16 blocks, three refinement levels, partitioned onto six processors with the Recursive Spectral Bisection algorithm. The colours represent processor assignment.

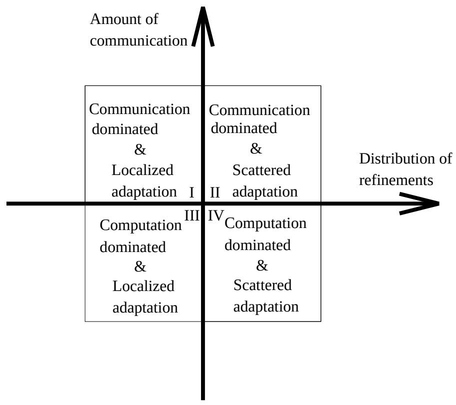

Figure 5.2: Applications and computer systems are classified with respect to the distribution of refinements and the amount of communication. Applications may start in one quadrant and move to other quadrants as the simulation progresses.

Chapter 6

Experimental results

The adaptive algorithm is applied to a number of two-dimensional test problems ranging from scalar problems to non-linear systems of PDEs. The experiments aim to evaluate the ability of the method to estimate the local errors in the solution and to select appropriate space and time steps accordingly. Two examples from Paper A are discussed here.

6.1 A scalar problem

First, consider the following scalar problem:

$$u_t + a_1 u_x + a_2 u_y = 0, (x, y) \in [0, 1] \times [0, 1], \tag{6.1}$$

where the velocity vector $a = (a_1, a_2)^T = 2\pi(-\sin\psi, \cos\psi)^T, \psi = \arctan y/x$. The analytical solution is simply a rotation of the initial solution about the origin. At time $t = 1$ the exact solution is the initial solution.

In this experiment, a Gaussian pulse located at $(0.75, 0.5)$ is used as the initial solution. The grid is partitioned into 8×8 blocks, and there are three refinement levels present in the grid. Figure 6.1 shows the initial ($t = 0$) and final grid ($t = 1$) for the scalar problem. The grids are close to identical.

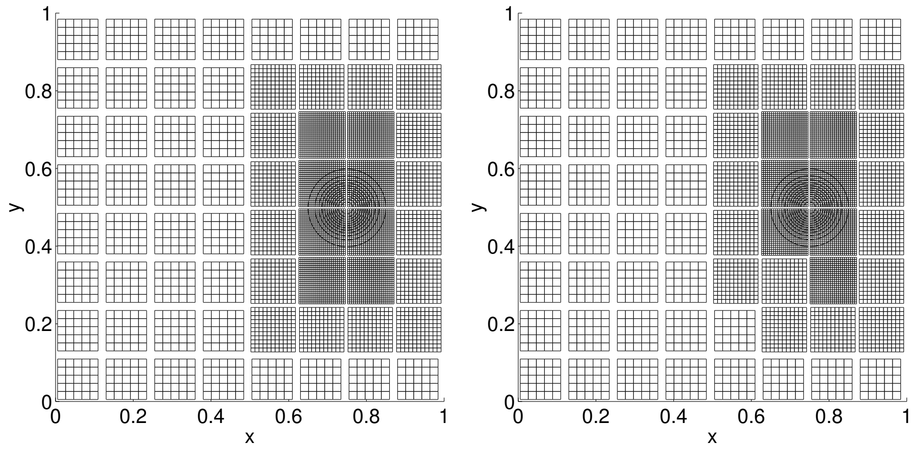

Figure 6.1: The adaptive grid and the isolines of u and $t = 0$ (left) and $t = 1$ (right) for the scalar problem (6.1). The grids are almost identical.

Since the analytical solution to this problem is known, the estimated local and global errors can be compared to the measured errors. Figure 6.2 shows the exact (dashed) and estimated (solid) local errors in space τ_s (left) and time τ_t (right) plotted along $x = 0.75$. The error tolerance ϵ is 0.01 in this experiment. The estimated and exact errors are very close. Figure 6.3 shows that also the estimated global error (solid) is close to the exact error (dashed).

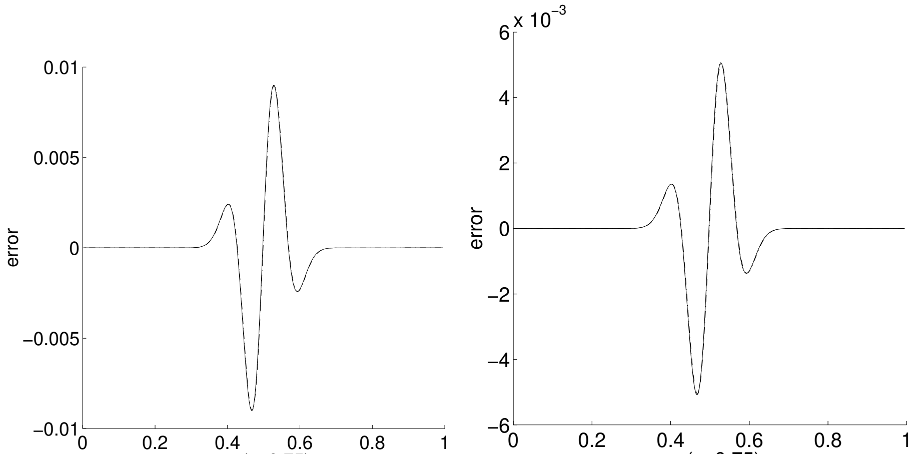

Figure 6.2: The exact (dashed) and estimated (solid) local errors in space (left) and time (right) along $x = 0.75$. The estimates are very close the the exact errors.

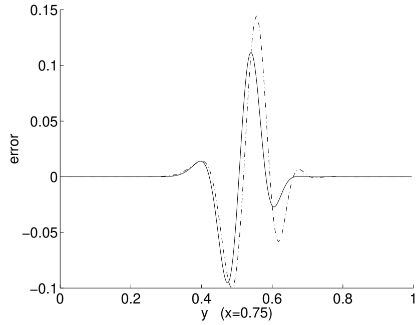

Figure 6.3: The exact (dashed) and the estimated estimated global error (solid) at $t = 1$.

Since the solution is just a rotation of the initial solution, the behavior of this application does not change significantly during the simulation. Hence, the time step can be expected to approach a constant value which is also illustrated in Figure 6.4. The total number of cells in the adaptive grid shows a slightly oscillating behavior, see Figure 6.4. This is due to the look-ahead strategy described in Chapter 4 which refines neighbouring blocks slightly prior to the pulse moving across the block boundary. The average number of cells including the cells used for estimating the spatial error is $4.05 \cdot 10^4$. Achieving the same resolution using a uniform grid with the same grid size in all blocks would require more than $1.47 \cdot 10^5$ cells.

The time step is almost constant in this case and could have been chosen a priori after some analysis. A constant time step requires less administration but it is seldom known in advance that such a step is sufficient to satisfy a local error constraint.

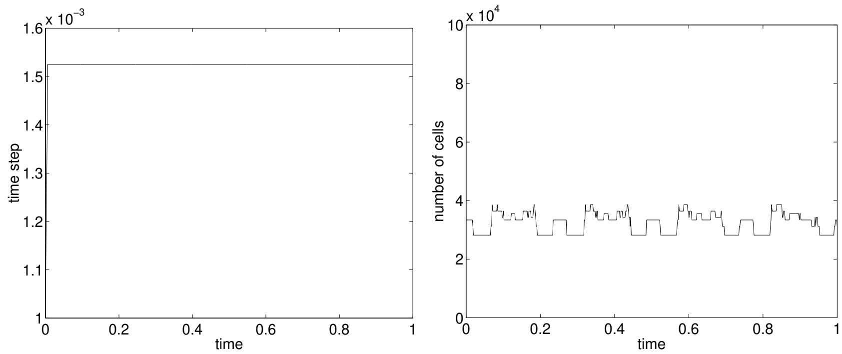

Figure 6.4: The time step (left) and total number of cells (right) used during the simulation.

6.2 The Euler equations

Inviscid, compressible flow is governed by the Euler equations. Let ρ be the density, u and v the Cartesian velocity components, E the total energy, and p the pressure. Then the equations are

$$\begin{aligned} \rho_t + (\rho u)_x + (\rho v)_y &= 0, \\ (\rho u)_t + (\rho u^2)_x + (\rho uv)_y + p_x &= 0, \\ (\rho v)_t + (\rho uv)_x + (\rho v^2)_y + p_y &= 0, \\ (\rho E)_t + (\rho uH)_x + (\rho vH)_y &= 0. \end{aligned} \tag{6.2}$$

The system is closed by the definitions of E and the enthalpy H and the gas constant γ:

$$E = \frac{p}{(\gamma - 1)\rho} + \frac{1}{2}(u^2 + v^2), \; H = E + \frac{p}{\rho}.$$

The solution is computed in a channel $(x, y) \in [0, 0] \times [4, 1]$ with periodic boundary conditions at $x = 0$ and $x = 4$ and with non-slip conditions ($v = 0$) at $y = 0$ and $y = 1$. The initial solution is

$$\begin{aligned} \rho(x, y, 0) &= 1 + \Delta\rho, \\ u(x, y, 0) &= 0.5, \\ v(x, y, 0) &= 0, \\ p(x, y, 0) &= 1 \end{aligned}$$

where $\Delta\rho = 0.1 \exp(-20((x - 1.0)^2 + (y - 0.5)^2))$ is a Gaussian pertubation in the density. The pertubation is advected downstream from its initial position $(1, 0.5)$.

The space derivatives are approximated by the second order Osher scheme [26] and time derivatives by BDF-2. The solution is computed on 8 processors using 16×4 blocks and two refinement levels. Figure 6.5 shows the adapted grid and the density ρ at $t = 0$ (upper) and $t = 4$ (lower).

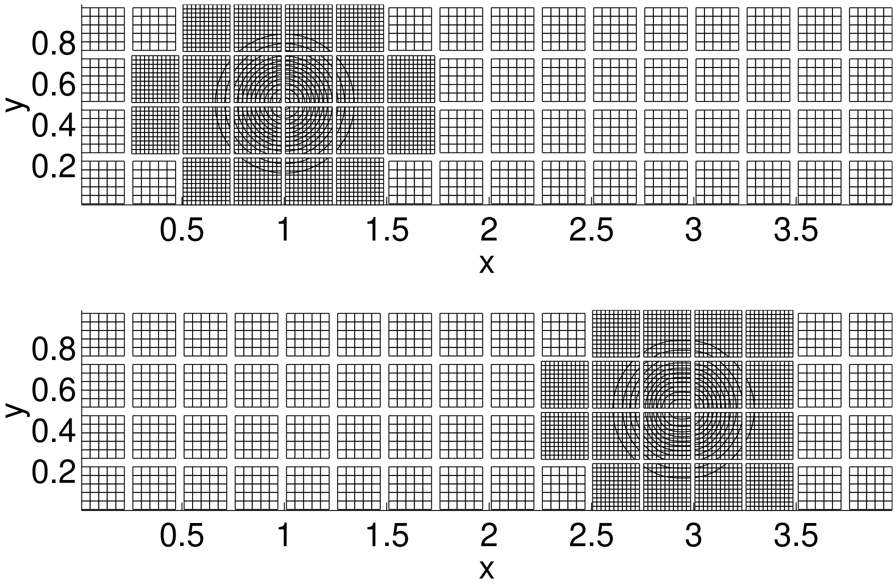

Figure 6.5: Isolines of the density ρ at $t = 0$ (upper) and $t = 4$ (lower).

The total number of cells and the time steps used during the simulation is shown in Figure 6.6. The cell consumption is on average $1.75 \cdot 10^4$. A uniform grid with the same resolution would require $3.69 \cdot 10^4$ cells. The gain in cell consumption is relatively moderate since only two refinement levels were needed to resolve the pertubation.

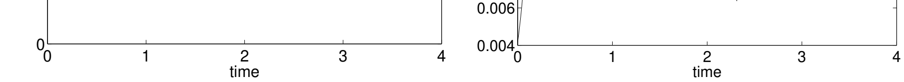

Figure 6.6: The total number of cells in the adaptive grid (left) and the time steps (right) used in the simulation.

Chapter 7

Conclusions

A second order, space and time adaptive method has been derived. The method dynamically adapts the grid and time step so that the local numerical errors are kept below a user defined error tolerance. An estimate of the global error in the solution is also computed. The estimated errors are close to the exact errors. Using error estimates as the basis for refinement and coarsening gives a quantitative control of the errors in the solution.

The grid is adapted by successive refinement in blocks. There may be jumps in the step size at the block boundaries where special care is needed to ensure stability and second order accuracy. The number of blocks used in the grid has an impact on both the performance and the cell efficiency of the method. The blocking overhead is mainly due to inter-block communication within processor partitions. A method of predicting the optimal number of blocks is suggested. The algorithm is based on a model of the execution time and the input is measurements of the time for arithmetic operations and communications as well as a typical refinement pattern.

The distribution of the cells over the processor set is determined at block level by applying grid partitioning packages such as ParMETIS [21] directly to the block graph. This yields low overhead costs for computing the new partition and for administrating the partitions compared to cell level partitioning.

The implicit time discretisation generates a system of equations which is iteratively solved in every time step by GMRES. For problems with different time scales, preconditioning is needed for fast convergence. Preconditioning is addressed in [18].

The method is used to successfully solve a number of two-dimensional examples in parallel. Achieving the same resolution with a uniform grid and constant time step requires 1.5-2 times more computational work and 2-3 times more memory. In addition, an estimate of the global error in the solution is computed.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Per Lötstedt and to my colleagues Dr. Lina Hemmingsson-Frändén and Dr. Alison Ramage* for all their hard work and patience. This research was supported by the Swedish Foundation for Strategic Research via the program of Industrial Computational Mathematics.

* Department of Mathematics, University of Strathclyde, Glasgow

Bibliography

[1] J. Bell, M. Berger, J. Saltzman, M. Welcome, Three-dimensional adaptive mesh refinement for hyperbolic conservation laws, *SIAM J. Sci. Comput.*, 15 (1994), p. 127–138.

[2] M. J. Berger, Stability of interfaces with mesh refinement, *Math. Comp.*, 45, p. 301-318.

[3] M. Berger, P. Colella, Local adaptive mesh refinement for shock hydrodynamics, *J. Comput. Phys.*, 82 (1989), p. 64–84.

[4] M. Berger, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, *J. Comput. Phys.*, 53 (1984), p. 484–512.

[5] M. Berger, A. Jameson, Automatic adaptive grid refinement for the Euler equations, *AIAA J.*, 23 (1985), p. 561–568.

[6] A. Bodman, A blockwise adaptive method on an unstructured grid, MSc thesis, *UPTEC F 01 041*, Uppsala University, Uppsala, 2001.

[7] P. Colella, M. R. Dorr, D. D. Wake, Numerical solution of plasma fluid equations using locally refined grids, *J. Comput. Phys.*, 152 (1999), p. 550–583.

[8] R. L. Davis, J. F. Dannenhoffer, III, 3-D adaptive grid-embedding Euler technique, AIAA paper 93-0330, 1993.

[9] J. De Keyser, K. Lust, D. Roose, Run-time balancing support for a parallel multiblock Euler/Navier-Stokes code with adaptive refinement on distributed memory computers, *Parallel Comput.*, 20 (1994), p. 1069–1088.

[10] D. De Zeeuw, K. G. Powell, An adaptively refined Cartesian mesh solver for the Euler equations, *J. Comput. Phys.*, 104 (1993), p. 56–68.

[11] K. Eriksson, D. Estep, P. Hansbo, C. Johnson, Introduction to adaptive methods for differential equations, *Acta Numerica*, (1995), p. 105–158.

[12] L. Ferm, P. Lötstedt, S. Söderberg, Error control in the solution of flow problems, Proceedings of ISCFD99, *Computational Fluid Dynamics Journal*, 9 (2001), 506-514.

[13] L. Ferm, P. Lötstedt, Blockwise adaptive grids with multigrid acceleration for compressible flow, *AIAA J.*, 37 (1999), p. 121–123.

[14] L. Ferm, P. Lötstedt, Adaptive error control for steady state solutions of inviscid flow, Report 2000-003, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, 2000, *SIAM J Sci. Comput.* to appear.

[15] R. Hagmeier, Grid adaption based on modified anisotropic diffusion equations formulated in the parametric domain, *J. Comput. Phys.*., 115 (1994), p. 169–183.

[16] E. Hairer, S. P. Nørsett, G. Wanner, *Solving ordinary differential equations*, 2nd ed., Springer-Verlag, Berlin, 1993.

[17] L. Hemmingsson-Frändén, P. Lötstedt, A. Ramage, S. Söderberg, Implicit solution of hyperbolic equations with space-time adaptivity, *BIT*, 42:1, p. 128–153.

[18] L. Hemmingsson-Frändén, P. Lötstedt, A. Ramage, S. Söderberg, Preconditioned implicit solution of linear hyperbolic equations with space-time adaptivity, in preparation.

[19] M. Holmström, Solving hyperbolic PDEs using interpolating wavelets, *SIAM J. Sci. Comput.*, 21 (1999), p. 405–420.

[20] R. D. Hornung, J. A. Trangenstein, Adaptive mesh refinement and multilevel iteration for flow in porous media, *J. Comput. Phys.*, 136 (1997), p. 522–545.

[21] G. Karypis, K. Schloegel, V. Kumar, PARMETIS–Parallel graph partitioning and sparse matrix ordering library, Version 2.0, Univ. of Minnesota, Minneapolis, MN, 1998.

[22] S. Li, L. Petzold, Y. Ren, Stability of moving mesh systems of partial differential equations, *SIAM J. Sci. Comput.*, 20 (1998), p. 719–738.

[23] F. Liu, S. Ji, G. Lao, An adaptive grid method and its application to steady Euler flow calculations, *SIAM J. Sci. Comput.*, 20 (1998), p. 811–825.

[24] P. Lötstedt, S. Söderberg, Parallel solution of hyperbolic PDEs with space-time adaptivity, in *Finite Volumes for Complex Applications II*, eds. R. Vilsmeier, F. Benkhaldoun, D. Hänel, Hermes Science, Paris, 769-776, 1999.

[25] S. Muzaferija, D. Gosman, Finite-volume CFD procedure and adaptive error control strategy for grids of arbitrary topology, *J. Comput. Phys.*, 138 (1997), p. 776–787.

[26] S. Osher, F. Solomon, Upwind schemes for hyperbolic systems of conservation laws, *Math. Comp.*, 38 (1982), p. 339–377.

[27] K. Otto, Analysis of preconditioners for hyperbolic partial differential equations, *SIAM J. Numer. Anal.*, 33 (1996), p. 2131–2165.

[28] N. G. Pantelelis, A. E. Kanarachos, The parallel block adaptive multigrid method for the implicit solution of the Euler equations, *Int. J. Numer. Meth. Fluids*, 22 (1996), p. 411–428.

[29] K. G. Powell, A tree-based adaptive scheme for solution of the equations of gas dynamics and magnetohydrodynamics, *Appl. Numer. Math.*, 14 (1994), p. 327–352.

[30] K. G. Powell, P. L. Roe, T. J. Linde, T. I. Gombosi, D. L. De Zeeuw, A solution-adaptive upwind scheme for ideal magnetohydrodynamics, *J. Comput. Phys.*, 154 (1999), p. 284–309.

[31] Y. Saad, M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comput.*, 7 (1986), p. 856–869.

[32] P. Skogqvist, High order adaptive difference methods for combustible flows, PhD thesis, Dept of Numerical Analysis and Computer Science, Stockholm University, Stockholm, Sweden, 2001.

[33] K. Srinivasan, S. G. Rubin, Solution-based grid adaptation through segmented multigrid domain decomposition, *J. Comput. Phys.*, 136 (1997), p. 467–493.

[34] J. Steensland, S. Söderberg, M. Thuné, Comparison of dynamic load balancing techniques for a parallel SAMR algorithm, in T. Sørevik, F. Manne, R. Moe, A. H. Gebremedhin, eds., Applied Parallel Computing – New Paradigms for HPC in Industry and Academia, Lecture Notes in Computer Science 1947, *Springer-Verlag*, Heidelberg, 2001, p. 160–169.

[35] J. Steensland, Vampire homepage, http://www.caip.rutgers.edu/~johans/vampire/, 2000.

[36] B. Strand, Numerical studies of hyperbolic IBVP with high-order finite difference operators satisfying a summation by parts rule, *Appl. Numer. Math.*, 26 (1998), p. 497-521.

[37] P. Wesseling, Principles of Computational Fluid Dynamics, *Springer Verlag*, Berlin, 2001.

[38] J. Wu, H. Ritzdorf, K. Oosterlee, B. Steckel, A. Schüller, Adaptive parallel multigrid solution of 2D incompressible Navier-Stokes equations, *Int. J. Num. Meth. Fluids*, 24 (1997), p. 875–892.