A Framework for Partitioning Composite Grids

by

Jarmo Rantakokko

Report No. 204/1998

UPPSALA UNIVERSITET
Institutionen för
teknisk databehandling

UPPSALA UNIVERSITY
Department of
Scientific Computing

A Framework for Partitioning Composite Grids

Jarmo Rantakokko

February 25, 1998

Abstract

A framework is presented for partitioning of composite grids. It includes partitioning strategies found in the literature, as well as new algorithms proposed here. In particular, a multilevel graph partitioning strategy – specifically designed for structured composite grids – is proposed. Different partitioning strategies are compared in a number of case studies involving multiblock grids. One of the applications is a profile of a multi-element wing, Airbus 310. The partition properties from the different strategies and algorithms depend very much on the number of subgrids and their sizes as well as on the number of processors. The previous strategies behave well only in some cases while the new multilevel graph partitioning strategy gives an overall good performance. Finally, a set of software tools is presented, that supports the framework. The software consists of low-level operations that can be combined flexibly to compose a large class of partitioning algorithms. Some automatic partitioning methods are also provided.

Contents

1 Introduction

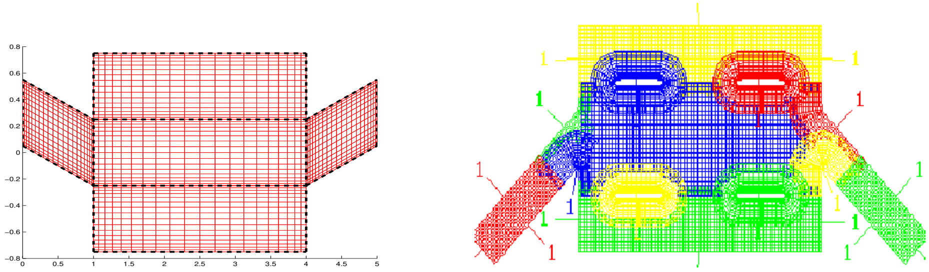
The progress of efficient and accurate numerical methods and as well efficient modeling techniques has made it possible to solve for realistic and accurate fluid flows in complex geometries. For example, the airflow around an airplane or a car can be modeled and computed numerically. The resolution in the solution is mainly limited by memory and computation time requirements. Parallel computers offer both large memory and high computational power in a cost efficient way. The most common and also the most powerful parallel computers today are distributed memory MIMD-machines, *Multiple Instruction stream Multiple Data stream*, and for the applications in mind the programming style is SPMD, *Single Program Multiple Data*. Each processor executes the same program asynchronously and the problem is solved concurrently by partitioning the data in different parts for the processors to work on.

The issue in this paper is the partitioning of composite grids, irregularly coupled regular meshes. Real-life flow simulations require high resolution and efficient numerical methods. A viable choice for the numerical approach is to model the geometry with a structured grid system, a composite grid, see Section 2. Due to the irregular data dependencies between subgrids the partitioning of composite grids is non-trivial and can not be done optimally with todays data parallel compilers, for example *High Performance Fortran* (HPF). The data parallel compilers usually distributes the data arrays one at a time over all processors. Composite grid applications have several arrays of varying lengths with irregular connections between the arrays. It is then not appropriate to partition the arrays over all processors disregarding the coupling between the arrays. Also, partitioning small arrays is not very efficient as it yields relatively much communication points compared to computation points. The HPF-2 standard and the approach in [2] use compiler directives to define processor subspaces and to map different arrays to different subsets of the processors. If the arrays have varying lengths a severe load imbalance can arise between the subsets. The definition of the sizes of the processors subspaces, i.e. the load balancing, is left to the user. The HPF-2 standard also allows for arbitrary mapping of each data element but the mapping must be defined by the user. So, there is still need for data partitioning algorithms. Also for explicit parallelization, using message passing, the natural strategy is to decompose the data over the processors. To get efficient programs the arithmetic workload must be balanced and the inter-processor communication minimized. A comparison of different programming models for multiblock computations is reported in [36]. The data partitioning is important for all programming models.

Composite grids exhibit both a coarse-grain parallelism at the subgrid level, i.e. computing on different element grids in parallel, and a fine-grain parallelism at the mesh point level, i.e. parallel computing within one element grid. Most of the previous work treats either the coarse-grain parallelism or the fine-grain parallelism. The coarse-grain parallelism is straightforward to implement with a minimal programming effort by assigning complete element grids to the processors. An amount of literature describes this approach. The strategy is also well suited for a multigrid approach [1], all grid-levels on a block can be assigned to one processor. The arithmetic workload is balanced by combining or packing the element grids to the processors with some partitioning algorithm, for example with any Greedy-like method [9] or bin packing method [26] minimizing a cost function [46, 1]. Assigning these large entities of work can cause serious load balancing problems, i.e. if the blocks and the block sizes do not match the number of processors we get a load imbalance that can be relatively large. A remedy is to use a pre-processing step to split large element grids into a number of smaller ones, introducing more internal boundaries [8, 21, 26, 36]. In [1] a detailed split algorithm is given that also considers the numerics in the problem, multigrid and boundary layer constraints in this case. Splitting grids is only a suboptimal solution. Many small subgrids are favorable with regard to the load balance but the communication and the programming overhead are increased, for example we get more serial overhead due to memory management (dependencies between neighbor blocks) and an increased number function calls (more blocks to operate on). Moreover, the convergence and the accuracy can be affected [27, 38]. Another approach is to put restraints on the grid generation to produce a balanced workload for a fixed number of processors, see for example [17]. This not proper for all applications since the blocks should be generated on basis of the geometry and refined following aerodynamic requirements. The coarse-grain parallelism is usually referred to as *domain decomposition* and is a very popular approach in computational fluid dynamics. It is relatively easy to implement and gives good parallel performance. The communication overhead between the blocks can be efficiently reduced. For example in a multistage Runge-Kutta the block-connectivity is usually applied only once in each timestep but with a result of a slower convergence to steady-state. The *actual* speedup is then reduced. For higher-order methods and time-accurate methods this is more significant since some of the accuracy is lost if not full communication is applied at the block boundaries.

The fine-grain parallelism is treated by distributing the element grids one at a time over all processors, [36, 37, 41]. The granularity can be seen as coarse if the element grids are partitioned in blocks, but the parallelism is still on the grid-point level and the communication is more extensive than on the grid level. Thus, in practice the line between the coarse-grain and fine-grain parallelism is somewhat indistinct. In [7] the two levels of parallelism are discussed, but how to extract them at the same time in the most efficient way is still left as an open question. Also [22] touches upon this subject. The meshes are classified in large, medium, and small sized meshes. The large meshes are distributed over all processors but the partitions are mapped to the processors with regard to the coupling between the subgrids. Medium sized grids are packed together and distributed as large grids. The small grids are used to fill in holes in the packing. In [4] another partitioning approach is suggested. The composite grids are partitioned geometrically by first assigning work-factors to different areas and then partitioning the geometric domain in balanced subdomains. But, if the grids are rotated so the gridlines do not match it is no longer obvious how to use this strategy in practice.

Our contribution is a multilevel graph partitioning approach where we extract both levels of the inherent parallelism in an efficient way. We consider all



(a) Multiblock grid with blocks meeting edge to edge.

(b) Hierarchical overlapping grid with higher level blocks cutting out details.

Figure 1: Composite structured grids modeling an expansion pipe.

couplings between the subgrids and at the same time balance the arithmetic workload very well. We have also suggested an optimal processor clustering strategy that can be used to define the processors subspaces in HPF-2. The work is tied together in a general strategy that sets up a framework for partitioning composite grids. The framework encompasses previous partitioning methods as well as the new algorithms proposed by us. Moreover, we have developed a set of software tools that gives support for the framework. With the software tools a large class of partitioning algorithms can be implemented. Finally, as a case study we have evaluated different partitioning strategies by partitioning a multiblock grid from a real-life application, A310 wing profile.

2 Composite grids

Numerical solution of partial differential equations with finite difference methods has proven to be efficient and accurate. Finite difference methods are related to structured grids. However, single grids are not sufficient in realistic applications and composite grid methods must be used. A composite grid is a union of structured grids. The computational domain is decomposed in logically rectangular domains. Each subdomain is discretized building a component of the composite grid. The different component grids interact with each other via interpolation. The simplest variant is so called multiblock grids where the blocks meet edge to edge, see Figure 1(a).

A more general approach is to let the grids overlap. A straightforward extension is then to allow for a hierarchy of blocks, higher level grids imposed on the base grids refining the resolution or cutting out details, see Figure 1(b). The multiblock grids constitute a very important special case of composite grids. These grids are frequently used in, e.g., aerodynamical computations. Moreover, the interpolation becomes much simpler than for overlapping grids where several grids may overlap at some point.

An alternative to using composite structured grids and finite difference methods is to use unstructured grids and finite element or finite volume methods. With unstructured grids it is much simpler to model complex geometries,

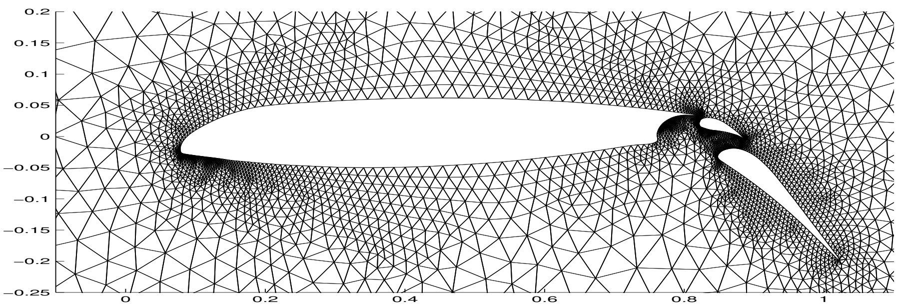


Figure 2: Unstructured finite element grid around a multi-element airfoil.

see Figure 2, but it requires more memory and more complicated data structures for book-keeping of the grid-points, limiting the size of the problem that can be solved.

For computation of flows requiring high resolution of the flow field, structured grids offer clear advantages over unstructured grids due to less memory requirements and efficient solving techniques [11, 30, 40]. The regularity of the grid structure makes it easy to implement different convergence acceleration techniques, for example multigrid schemes. Likewise, coupled equations with inviscid and viscous parts can be very naturally implemented. It is possible to use *potential, Euler,* and *Navier–Stokes* solvers on different blocks. The grid resolution can differ and the blocks can advance in time with different time-steps decided by stability conditions. However, automation of grid generation and adaptivity to the flow solution is far more developed for unstructured grids. Also, very efficient partitioning algorithms have been developed for unstructured grids. For structured grids there is a trade-off between structure and load balance which depends on the relative sizes of the subgrids and makes it more difficult to do automatic partitioning. We have developed a framework with efficient partitioning methods for structured composite grids of varying sizes, comparable with the partitioning methods for unstructured grids.

3 Partitioning strategies

3.1 General approach

Parallelization of a structured grid solver requires a structured data partitioning. To keep the efficiency and simple data structures for finite difference methods, the partitions must be rectangular blocks. Partitioning a structured composite grid is still non-trivial due to the different sizes of the subgrids and the irregular data dependencies in the interpolation between grids. Partitioning a single structured grid is straightforward and much previous work treats this problem, see for example [24, 33, 42]. A natural extension to composite grids is to apply the single grid partitioning at one grid at a time. The partitions can then be mapped differently to the processors in the different grids, with respect to coupling between the grids, minimizing the inter-processor communication [22]. Still, there will be much communication as all grids are distributed over all processors. This strategy is sufficiently efficient for computationally intensive problems (few large grids) on a modest number of processors [41]. Another natural strategy is to exploit the inherent coarse-grain parallelism and distribute different element grids to different processors. Here, the number of grids and the grid sizes are very critical for the load balance. In practice, this strategy works well only if the number of grids is much larger than the number of processors (many small grids). Combining the coarse-grain and the fine-grain parallelism gives the potential for all over efficient partitioning. In [43], a theoretical analysis of such mixed data partitioning strategies for general composite grids is carried out. Here, we will further develop the ideas and work out the practical details in the data partitioning algorithms for the multiblock case. The extension to hierarchical overlapping grids is then straightforward.

We can formulate a general partitioning strategy for composite grids including the partitioning strategies above, these can be seen as special cases of the general strategy.

Composite grid partitioning strategy: *Cluster the grids and the processors, respectively, in groups. Distribute the clusters of element grids over the clusters of processors. Within a processor cluster, partition the corresponding grids over the processors.*

Within the framework set by this strategy, we have four degrees of freedom: (i) how to choose the clusters, (ii) how to map the grid clusters to the processor clusters, (iii) how to partition within the processor clusters, and (iv) how to map the partitions to the processors. How to choose the clusters depends very much on the parallelization of the solver and the network of the parallel computer. Define N_p as the number of processors and N_g as the number of element grids, then we can specify four important special cases:

1. *N_p clusters of processors, N_g clusters of grids ($N_p < N_g$).* Complete element grids are distributed to different processors.

2. *One cluster of processors, N_g clusters of grids* The grids are one at a time partitioned over all processors.

3. *N_g clusters of processors, N_g clusters of grids ($N_g < N_p$)* The grids are partitioned in different clusters of processors.

4. *One cluster of processors, one cluster of grids* The complete composite grid is partitioned over all processors.

The two first strategies are very simple and work well for some problems. The third strategy is optimal if we have a very large number of processors. The fourth strategy is the most general all purpose partitioning strategy. It considers the neighbor relations within the composite grid and thus has a potential to give optimal partitions. We will exploit these strategies more in detail below. Other intermediate clusterings (C_g, C_p), where $1 \leq C_g \leq N_g$ and $1 \leq C_p \leq N_p$, are motivated for example for heterogeneous computer networks. Recently, computers with *Symmetric Multi Processor* nodes have emerged as cost efficient parallel computers. Here the processors are clustered physically in the nodes, sharing memory, and the communication overhead between the clusters is higher than within a cluster. Then, it is natural and efficient to exploit this processor clustering [14]. Even though our work is not limited to, we will mainly consider homogeneous computer architectures.

The sizes of the clusters are always critical for the load balance. If the clusters of element grids can not perfectly match the clusters of processors there will a load imbalance between the processor clusters [43],

$$\Lambda = 1/(1 - \max_i \delta_i), \tag{1}$$

where δ_i is the relative round-off error defined by

$$p_i = (1 - \delta_i)p_i^* \tag{2}$$

from assigning p_i processors to grid cluster i instead of the ideal number (non-integer) of processors p_i^*. The load imbalance ratio is defined as $\Lambda = r_{\max}/r$, where $r_{\max}$ is the work per processor in the most heavily loaded processor cluster and r is the global average or optimal load per processor. The analysis in [43] is carried out for the case $N_g < N_p$, more processors than element grids. Similar results can also be derived for $N_g > N_p$, see Proposition 1 below. Assigning only integer numbers of processors to the grid clusters can yield relatively large round-off errors, δ_i. The situation can be improved by allowing for joint membership in different clusters for the processors. Then, a processor can be shared between several clusters and δ_i can be decreased. Within a cluster, the reduced computing power of the shared processors must be taken into account. For example in recursive bisection, both the workload and the processors must be divided proportionally in each bisection step, i.e. divide the workload in

$$W_L = W_{\text{tot}} \cdot \frac{P_L}{P_L + P_R}, \quad W_R = W_{\text{tot}} \cdot \frac{P_R}{P_L + P_R},$$

where P_L is the computational power of the left processors set and P_R of the right processor set, respectively. This partitioning approach only reduces the arithmetic work imbalance and does not consider the inter-cluster communication. Especially, for shared processors it is important to consider the inter-cluster dependencies. We will here develop a strategy that also accounts for this.

3.2 N_p clusters of processors, N_g clusters of grids

Here, we distribute complete element grids to the processors. A necessary condition for a good load balance is that $N_p < N_g$ and preferably that $N_p \ll N_g$ so that the element grids can be combined to give an even workload on the processors.

Proposition 1 *For optimal packing, disregarding neighbor relations, the arithmetic work per processor satisfies*

$$r_{\max} \leq r + w_{\max}$$

where $w_{\max}$ is the arithmetic work on the heaviest grid.

Proof In a first step, assign grids to the processors while $r_i \leq r \quad \forall i$. Then, assign the remaining grids to the least loaded processors. It is always possible to assign a grid to a processor with $r_i \leq r$ (by definition of r). Assume that the grid corresponding to $w_{\max}$ is the final grid to assign, worst case. Then $r_i \leq r + w_{\max} \quad \forall i$. $\Box$

Example Consider a composite grid with equal workload w on all element grids. Then $r = (N_g w)/N_p$ and from the definition of the load imbalance using Proposition 1 we get

$$\Lambda = \frac{r_{\max}}{r} \leq \frac{r+w}{r} = 1 + \frac{w}{(N_g w)/N_p} = 1 + N_p/N_g.$$

We can see that the more grids we have the better load balance we can get. The worst case is when we have $N_g = N_p + 1$, then $\Lambda = 1 + (N_p - 1)/N_g$.

The major deficiency of this strategy is the lack of load balance for general composite grids. The strength of the strategy is the use of the inherent coarse-grain parallelism, yielding that the inter-processor communication becomes less intensive. Communication is then only needed in the interpolation between element grids which makes it very important to consider the neighbor relations within the composite grid. This can be achieved by putting up a connectivity graph where the element grids are vertices and then by applying a graph partitioning method, for example the Recursive Spectral Bisection method (RSB) [39]. We will refer to this strategy as the *block-rsb* method. A similar approach has been implemented in [21]. If the element grids are of very different sizes the connectivity requirements can be traded for better load balance by combining the grids more freely to the processors, as in [8]. We have implemented a *Greedy*-like method to distribute the grids to the processors. It is a variant of the Greedy-method in [18]. We will refer to this strategy as the *block-greedy* method. Compare the results from the *block-greedy* and the *block-rsb* methods in Figure 13. The *block-greedy* method distributes the blocks to the processors by minimizing the load imbalance without regarding the neighbor relations between the subgrids, while the *block-rsb* method gives less communication points at the expense of a higher load imbalance.

3.3 One cluster of processors, N_g clusters of grids

This strategy is to partition each element grid over all processors. It is straightforward to apply, we can use any partitioning algorithm for a single grid and repeat it for all element grids in the composite grid. A similar strategy is also used in HPF. Below, this strategy will be referred to as *single*-grid partitioning. The strategy is analyzed theoretically in [41, 43] and experimental results can be found in [29, 37]. For computationally intensive problems it works well and it is also very well suited for dynamic grid structures. Addition or removal of a grid will not affect the load balance. The drawbacks are that it disregards neighbor relations between the element grids and is very communication intensive. Both the internal communication, within an element grid, and the external communication, between element grids, are much higher compared to the other

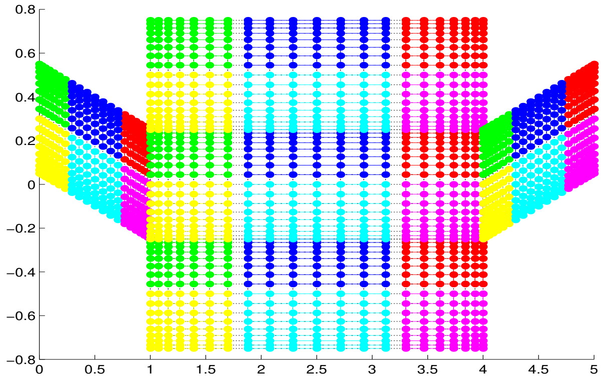


Figure 3: *Single*-grid partitioning strategy, the five subgrids are one at a time partitioned in rectangles and mapped to six processors. The number of communication points, edge-cut=351, is relatively high while the load balance, Λ=1.059, is good.

partitioning strategies. We have more internal partition boundaries within an element grid and also more partition boundaries between element grids, see Figure 3. It is possible to reduce the communication overhead by gathering the messages from all grids into a large message, as suggested in [43]. Also, a remapping of the partitions would reduce the communication overhead [22]. In this particular case it would be possible to completely eliminate the inter-processor dependencies between the subgrids. Still, the number of communication points would be very high. Finally, for many processors the ratio between the number of communication points and computation points becomes very high and with only a few grid points per processor it will also be difficult to maintain a good load balance within one grid. Thus, this strategy is not suited for massively parallel computers or for small grids.

The framework in [33], developed for single structured grids with inhomogeneous workload, is useful in this context. In the framework we do structured partitioning of arrays with varying workload, but can also do unstructured partitioning. Moreover, we can exclude parts of the grid that are not used. For composite grids we can assign different work factors on different parts of the element grids depending on the overlap of grids, interpolation formulas, boundary conditions, equations and the numerical method. In hierarchical composite grids we have grids superimposed on the base grid refining the solution in different parts of the domain or cutting out details. Then, we have superfluous parts on the base grids that can be excluded. Thus, our single grid framework is well suited for this partitioning strategy of composite grids.

3.4 Processor clustering

For the case $N_p > N_g$ and a homogeneous computer network, a compromise between the two former strategies is to cluster the processors and to partition different element grids in different processor clusters independently. Below, this will be referred to as the *cluster* strategy. The crucial issue is how to choose the number of processors in each cluster. Clearly, in an optimal partitioning the numbers of processors in the clusters are proportional to the grid sizes.

Proposition 2 *With the following rounding procedure the load imbalance in Equation (1) is minimized,*

$$
\begin{array}{ll}
1. & p_i = \lceil \frac{w_i}{w} N_p \rceil = \lceil p_i^* \rceil \quad \forall i \\
2. & q = \sum_i p_i - N_p \\
3. & q \ \textit{times}: \left\{ \begin{array}{l} \textit{find } j: \frac{p_j^*}{p_j - 1} = \min_i \frac{p_i^*}{p_i - 1} \\ p_j = p_j - 1 \end{array} \right.
\end{array}
$$

where w_i is the workload on grid i and w is the total workload on the composite grid. Moreover, $\Lambda \leq \max(p_i^* / \lfloor p_i^* \rfloor)$.

Proof After the first step, $p_i \geq p_i^*$ for all i and we have more processor power than optimal in some clusters. The total number of processors assigned is $N_p + q$, whereas there are only N_p processors actually available. We must then remove the q extra processors from the clusters. By removing processors from the cluster that gives the least load imbalance we minimize Λ. We can rewrite Equation (2) to $\delta_i = 1 - p_i/p_i^*$ and insert this in Equation (1),

$$\Lambda = 1/(1 - \max_i \delta_i) = 1/(1 - \max_i (1 - p_i/p_i^*)) = 1/\min_i (p_i/p_i^*) = \max_i (p_i^*/p_i)$$

Thus, finding $\min_i(p_i^*/(p_i - 1))$ and removing a processor from this cluster increases the load imbalance Λ as little as possible. Repeating this until all q processors are removed gives the optimal clustering. Consequently, $\Lambda \leq \max(p_i^*/\lfloor p_i^* \rfloor)$ as we can do better than truncation. $\square$

The suggested rounding procedure gives better utilization of available processors than the naive rounding, $p_i = \mathrm{nint}(p_i^*)$. Consider for example the case with $N_p = 8$ and $p_i^* = \{3.4, 2.4, 2.2\}$. The naive rounding gives $\Lambda = 1.2$ with one processor unused while our rounding procedure gives $\Lambda = 1.13$ and no unused processors. Consider also an extreme case with $N_p = 512$ and $p_i^* = \{507.1, 2.7, 2.2\}$. The optimal clustering, produced with our algorithm, is $p_i = \{506, 3, 3\}$. Here, direct rounding would give much worse results ($\Lambda = 1.1$ compared to $\Lambda_{\mathrm{opt}} = 1.002$).

Compared to the two former partitioning strategies, this strategy works well for large processor configurations while for small processor configurations the load imbalance can be very severe between the clusters, see Figure 4. By not rounding to integer number of processors, i.e. allowing for processors to be shared between clusters, we can improve the load balance. But, then the neighbor relations within the composite grid must be regarded in the partitioning phase in order to minimize the communication. We should no longer partition the grids independently. In contrast, by rounding to integer number of processors the grids can be partitioned independently as a part of only *one* grid will be assigned to each processor. The inter-grid dependencies will then also be

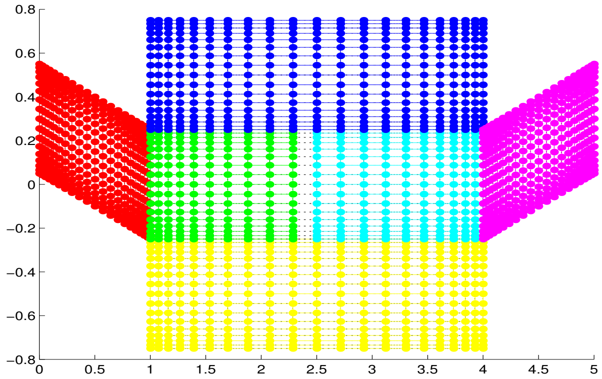


Figure 4: Processor *clustering* strategy, the processors are clustered with respect to the grids. Here, two of six processors are assigned to the center grid while the other grids have one processor each. The number of communication points is low, edge-cut=93, while the load imbalance is high, $\Lambda = 1.20$.

inter-processor dependencies which can not be minimized. Still, the mapping of grids to processor clusters and partitions to processors are important. In Figure 4, it would probably be more optimal to map another grid to the cluster with two processors. The total number of messages would then be reduced.

The processor clustering strategy is in line with the ideas presented in [7] of parallelizing the loop over subgrids for different sets of processors and then within a set of processors parallelizing the calculations for a subgrid. HPF-2 also supports this strategy by allowing definition of processor subsets and mapping of different arrays to different subsets. Our rounding procedure can then be used to determine the optimal number of processors in each subset.

3.5 One cluster of processors, one cluster of grids

Allowing for membership-exchange or joint membership for different processors in different clusters improves the load balance [13]. To achieve this and at the same time considering the data dependencies between element grids, we have below developed a strategy to partition the whole composite grid on all the processors at once. Separate processor clusters, with shared processors, can then be created from the assigned processors on each grid. In the case $N_g > N_p$ or N_g and N_p are of the same order, we usually get 1-2 processors in each cluster and many of the processors are shared between different clusters. For example, in Figure 5 we have 6 processors and we get five processor clusters, one for each grid, with two processors in each cluster/grid. In the case $N_g \ll N_p$ we get several processors in each cluster and only a few of the processors are shared between clusters.

The basic idea to partition the complete composite grid on all the processors is to divide each element grid into a number of smaller data blocks, set up a connectivity graph for the blocks and apply a graph partitioning method. Then, merge small blocks into larger rectangular blocks within each processor. Below, this will be referred to as the multilevel *graph* partitioning strategy. Multilevel

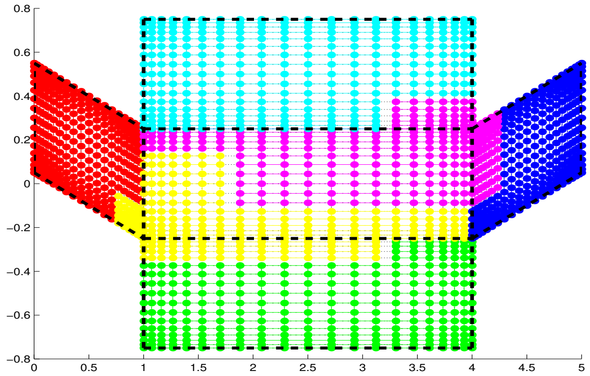
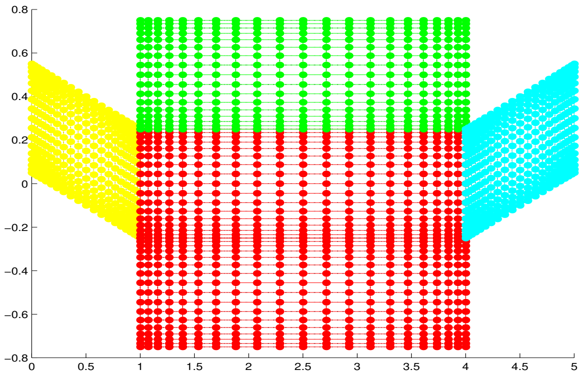


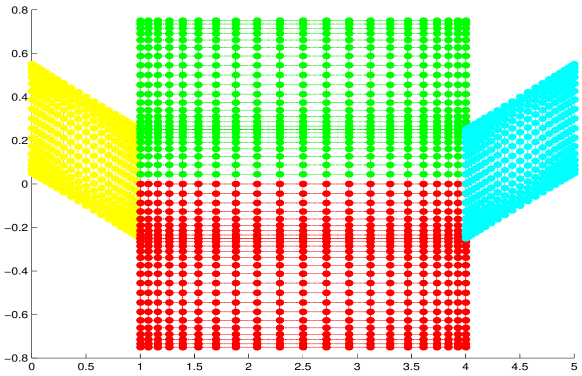
Figure 5: *Graph* partitioning strategy, the five subgrids are divided in 3x3 rectangular blocks, a connectivity graph with the blocks as vertices is set up, the graph is partitioned with the Recursive Spectral Bisection method, and finally small blocks are merged into larger rectangular blocks. The total number of blocks is reduced from 45 to 17 in the merge operation. Both the number of communication points, edge-cut=125, and the load imbalance ratio, Λ=1.059, are relatively low.

algorithms [15, 19] are currently the state of the art for *unstructured* graph partitioning. The general idea of those multilevel algorithms is to decrease the size of the graph by collapsing vertices and to partition the reduced smaller graph, with for example the Recursive Spectral Bisection method [39]. In the successive phase, the graph is expanded and the partitions are projected to the finer graph. The partitions are then further refined using a local refinement method. This will yield a fast algorithm to partition the graph. In our case, with *structured* grids, we can do the other way around: start with complete element grids as vertices on the coarsest level, refine this graph in a number of steps – by splitting the element grids into smaller data blocks – and partition the refined graph. Then, we can split the grids in even smaller data blocks and refine the partitions. If the number of processors is very large (or $N_g < N_p$), we can start by splitting the grids in smaller blocks and then proceed from this dividing. Thus, a multilevel algorithm can very easily be implemented in our framework. Moreover, we can adaptively decide how much to refine the blocks and stop the multilevel algorithm when we have an acceptable load balance. It can be difficult to find an appropriate block dividing from start in the non-multilevel cases. An example of a multilevel partitioning that gives good results in a few iterations can be seen in Figure 6. We start with complete element grids and split the blocks twice in each step. The refinement method is critical for the performance of the complete algorithm. This is discussed in [34].

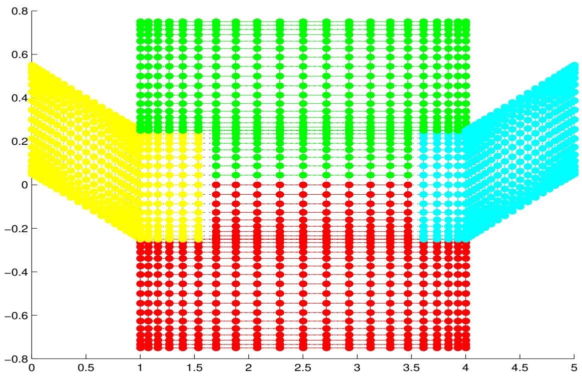
There is a trade-off between the total number of blocks in the end and the load balance. If we divide the grids in many small blocks we will approach an unstructured partitioning, i.e. the partition shapes will be composed by many small rectangular blocks and look more unstructured. If we have large blocks we can keep the structure in the partitions but then we have large units to distribute to the processors which can result in a bad load balance, especially



(a) Level 0, edge-cut 55, load Λ=1.60



(b) Level 1, edge-cut 55, load Λ=1.22



(c) Level 2, edge-cut 66, load Λ=1.03

Figure 6: Multilevel partitioning, we start with complete element grids in the initial level and split the blocks twice in each step. (4 partitions).

if the units do not match the number of processors or are of unequal sizes. If we have an inhomogeneous workload on the element grids, like in ocean modeling [33], equally sized blocks will have a different workload. We can create equally weighted blocks by recursively splitting *only* the heavy blocks until all blocks approximately have the same workload, or we can use a method like *Recursive Coordinate Bisection* to get equal weights on the blocks on one grid. When we have blocks with an even workload we can proceed as above to partition the graph of the blocks. It is more advantageous to have few blocks of different sizes and equally weighted than many small equally sized blocks. The load balance will be comparable while the structure will be better in the former case [33]. If the workload is homogeneous it is sufficient to make a uniform block partitioning of each element grid and to match the total number of blocks with the number of processors.

Our suggestion to a general partitioning strategy is a multilevel approach where we initially split the grids as discussed above depending on the application. Then, we set up a connectivity graph for the blocks and partition the graph with the RSB method. Depending on the workload requirements we further split the blocks and refine the partitioning. Finally, we merge small blocks within the same partitions. See Figure 7.

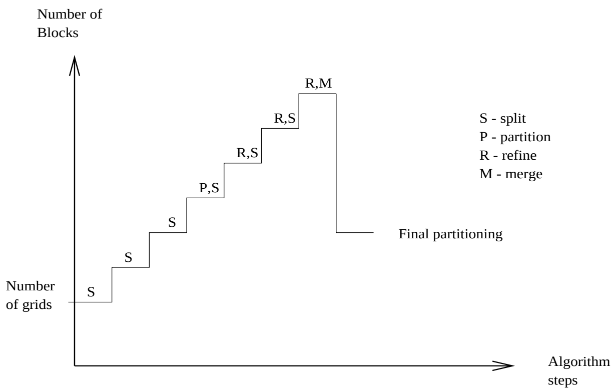
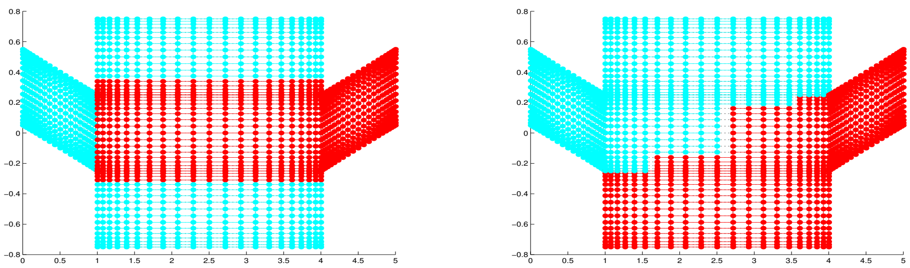


Figure 7: Graph partitioning algorithm, we initially split the subgrids in a number of levels, partition the resulting set of blocks, split again the blocks and refine the partitions, finally merge blocks within the same partitions.

For composite grids the communication differs within an element grid and between the element grids. The edge-weights correspond to the communication intensity and should be assigned accordingly. In some cases it can be better to assign the edge-weights somewhat differently. In [33] we have three ways to assign the edge-weights, *standard mode* with the weights proportional to the communication, *geometric mode* with all weights equal, and *block mode* with larger weights on smaller blocks. The last mode has a tendency to cluster small blocks together, thus minimizing the total number of blocks in the end. For composite grids we can also use these modes. The first two modes, standard and geometric, are directly applicable to composite grids. Also, the block mode applies naturally to composite grids. In the single grid case, we split the grid recursively in smaller and smaller blocks. The edge-weights are proportional to the level in which the blocks are split. For composite grids we have a natural first level, the element grid level. Then, for blocks that are neighbors and belong to different element grids we assign the lowest edge-weights. Within an element grid we proceed as in the single grid case. Assigning low weights for the inter-grid dependencies is natural since this communication is usually less frequent. But, using the block mode or the standard mode with large differences in the weights within and between the element grids can more easily result in disconnected domains for the RSB-method. The geometric mode can then in some cases give better results with connected domains and less edge-cuts, as an example see Figure 8.



(a) Block mode, edge-cut 59

(b) Geometric mode, edge-cut 38

Figure 8: Comparison of two strategies to choose edge-weights. (2 partitions). In the *block mode* the edge-weights are larger between small blocks while in the *geometric mode* all edge-weights are equal. The *block mode* strategy minimizes the total number of blocks at the expense of a higher number of communication points.

3.6 Semi-automatic partitioning

Some partitioning problems can be difficult to solve with automation, the predefined methods can fail to give a satisfactory solution, or the user may have some special requirements that can not be fulfilled with the predefined methods. Then we need additional input to solve the problem in the most efficient way, i.e. we need a semi-automatic partitioning method. By providing the user with low-level tools for partitioning, the user can create her own tailor-made solutions, see Section 5. As an example, see Figure 9 where we have partitioned a five-grid composite grid on four processors by assigning a complete element grid to each processor and then sharing the center grid between all processors.

3.7 Mapping

There is also a mapping problem in the physical computer topology. Related partitions must be mapped to neighboring processors in order to localize the communication and to avoid congestion of messages in the communication network. An appropriate mapping can make a considerable difference to the communication overhead in the parallel solution time. In most cases the mapping

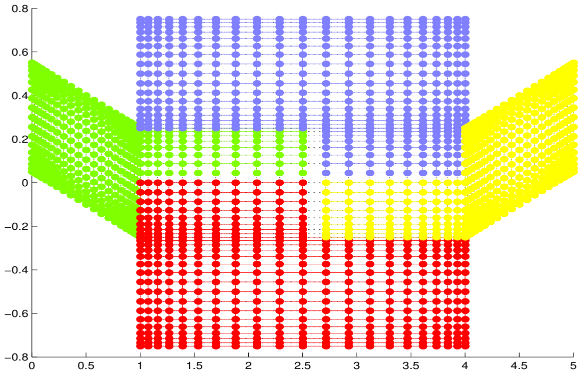


Figure 9: Example of a semi-automatic partitioning, we have assigned one complete element grid to each of the four processors. Only the center grid is partitioned, and shared between the four processors, using the Recursive Coordinate Bisection method at grid point level. The load balance, $\Lambda = 1.02$, is better than for the multilevel approach in Figure 6 while the number of communication points is somewhat higher, edge-cut=76.

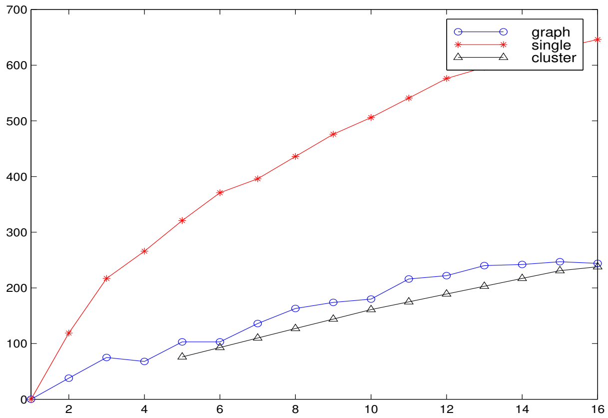
can be done as a post-processing step with a good result, especially if the partitioning is structured. But, by also considering the processor topology in the partitioning phase the parallel run time can be decreased significantly in some cases [45]. For the moment, we consider the mapping as a post-processing step and leave this topic for future work.

4 Applications

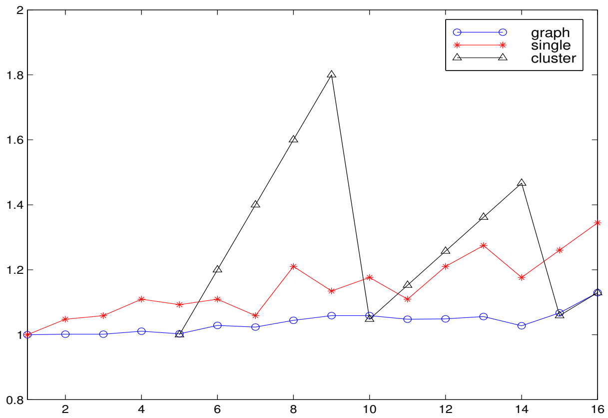
In this section we compare experimentally the partitioning strategies. As case studies, we partition three completely different composite grids and report on their respective partitioning properties. We have chosen to compare the number of communication points, *edge-cut*, the load imbalance ratio, Λ, and the number of *blocks* as a function of the number of processors.

4.1 Expansion pipe

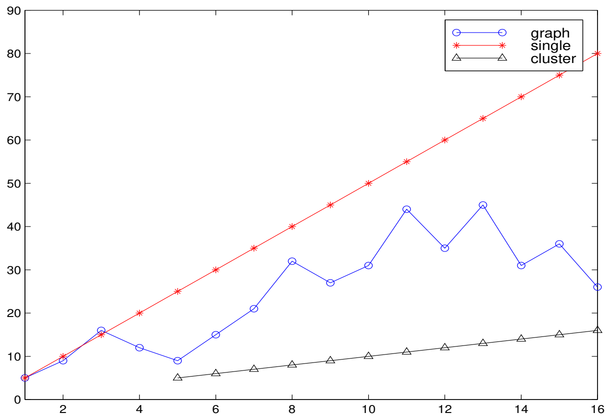
Our first example is the expansion pipe grid from Figure 1(a). The grid is used to solve the Navier-Stokes equations for airflow in the tube [12]. We have a multiblock grid with five subgrids. All subgrids are of equal sizes, 17×21 grid points in each grid. We have partitioned this grid with the *graph*, *single*, and *cluster* partitioning strategies respectively. In the graph-partitioning strategy we initially split all five grids in four blocks, totally 20 blocks. These blocks were then distributed to different partitions with the Recursive Spectral Bisection method. To get a better load balance the blocks were further split four times and the partitions refined with the local refinement method [34]. Finally, blocks were merged within the partitions. The results from this case study are presented in Figure 10. We can see that the graph partitioning strategy gives the best load balance at the expense of an increased number of blocks – and that the cluster



(a) Edge-cut as a function of N_p (processors).



(b) Load imbalance ratio Λ as a function of N_p.



(c) Number of blocks as a function of N_p.

Figure 10: Partitioning properties from partitioning the expansion pipe grid with the three different strategies, *graph*, *single*, and *cluster* respectively.

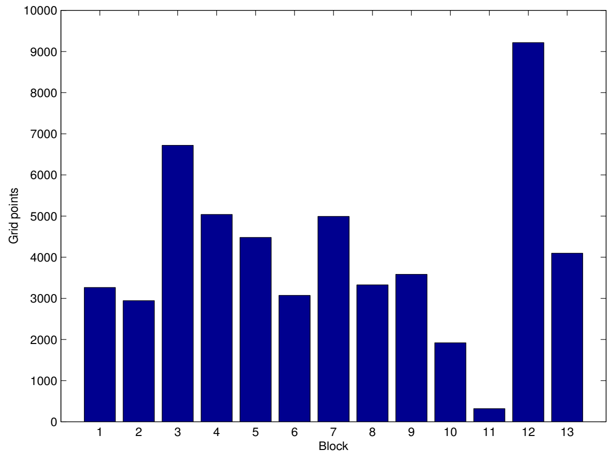


Figure 11: Number of grid points in the different subgrids the multi-element airfoil grid A310. The topology is shown in Figure 12.

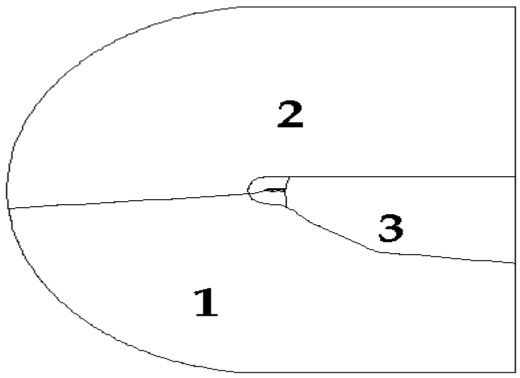
strategy is only optimal if the number of processors can be matched evenly with the number of grids. The single grid partitioning strategy is consistently non-optimal and this problem is too small for the strategy to work efficiently.

4.2 Multi-element airfoil

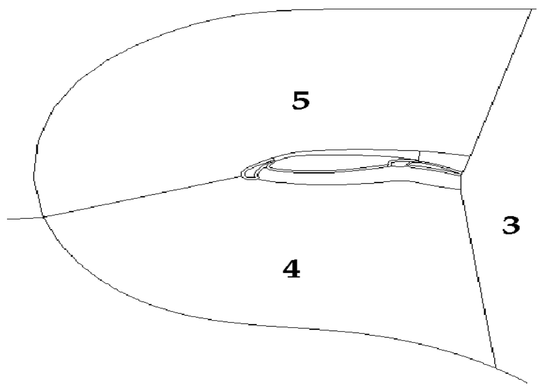
The second example is a multi-element airfoil[1] modeled with a structured multi-block grid. This is a more realistic example than the expansion pipe. We have 13 blocks of different sizes, see the Figures 11 and 12. This grid as well we have partitioned with the three strategies, *graph*, *single*, and *cluster* strategy. In addition, we have also partitioned the composite grid without splitting the subgrids. That is, we have distributed the grids as complete to the processors either by considering the neighbor relations, *block-rsb*, or by only optimizing for the load balance, *block-greedy*. Comparing these two strategies, Figure 13, we can see that the block-greedy method gives a better load balance but worse edge-cut than the block-rsb method which tries to minimize the inter-processor communication. Moreover, we can notice that these two strategies are only efficient if the number of grids is much larger than the number of processors while the graph and single grid partitioning strategies give almost optimal load balance for all number of processors. From Figure 14 we can see that the cluster strategy has comparable load balance only from $N_p \approx 70$ but then it is probably optimal since it has the smallest number of blocks and edge-cuts. The graph partitioning strategy gives an all over good compromise with a small load imbalance, low number of edge-cuts, and relatively few blocks.

As the sizes of the subgrids are very much unequal we have used a different block splitting approach here than for the expansion pipe grid above. As an

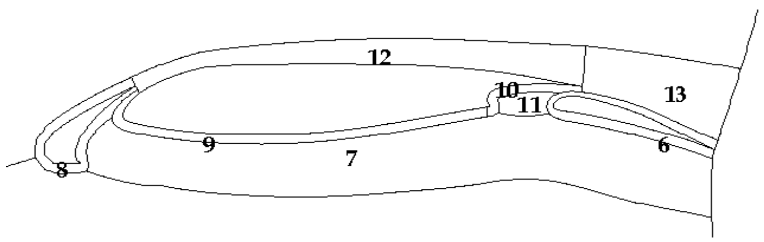
[1]This grid around the A310 wing profile was used in a European Garteur cooperation. The topology file and the plot of the topology of the grid was provided by Saab in Linköping, Sweden. The partitioning were carried out on the basis of the topology file without explicit knowledge of the coordinates.



(a)

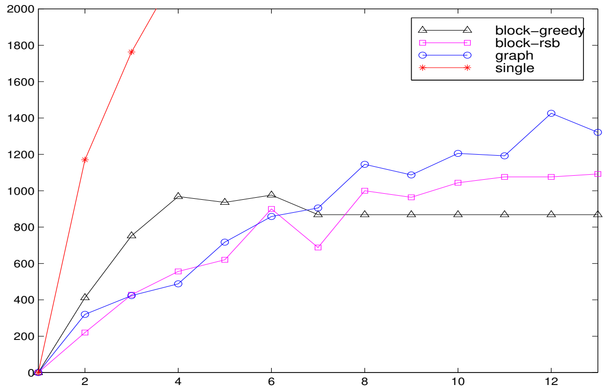


(b)

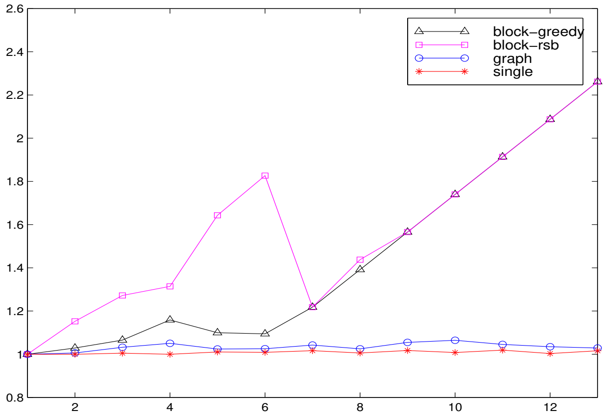


(c)

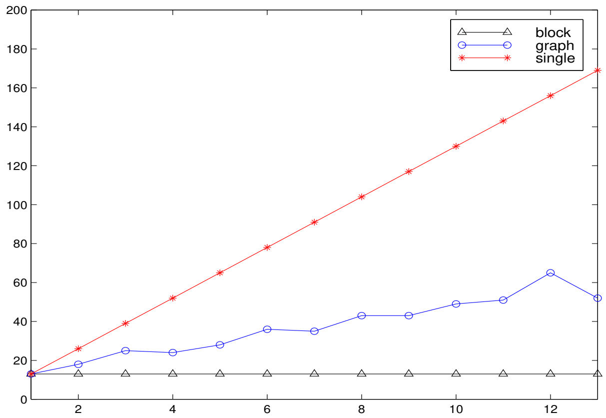
Figure 12: Topology of the multi-element airfoil grid A310. Three different resolutions are shown.



(a) Edge-cut as a function of N_p (processors).

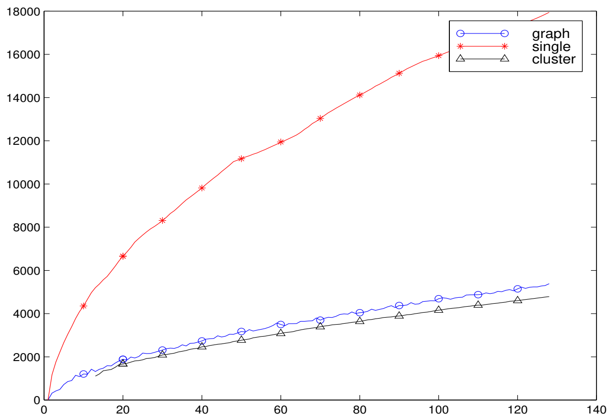


(b) Load imbalance ratio Λ as a function of N_p.

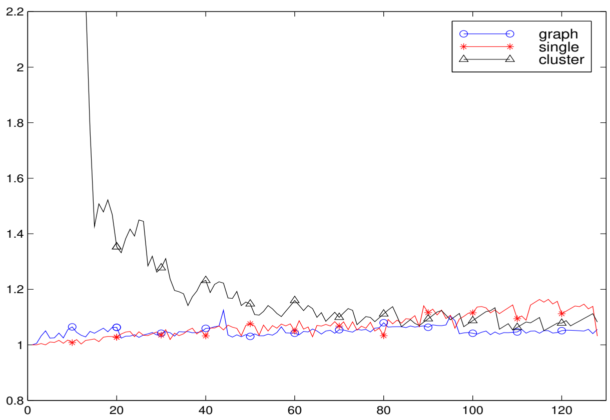


(c) Number of blocks as a function of N_p.

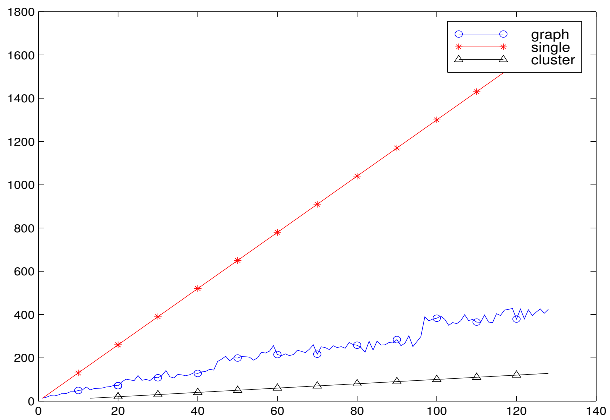
Figure 13: Partitioning properties from the multi-element airfoil case, small processor configurations ($N_p \leq N_g$).



(a) Edge-cut as a function of N_p (processors).



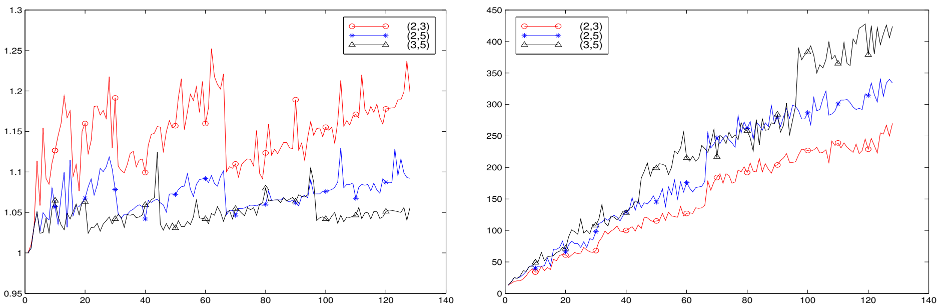
(b) Load imbalance ratio Λ as a function of N_p.



(c) Number of blocks as a function of N_p.

Figure 14: Partitioning properties from the multi-element airfoil case, large processor configurations ($N_p \geq N_g$).

initial decomposition we split the grids recursively in blocks until the number of blocks was more than three times higher than the number of processors. In each step, all grids were compared and if the size of the blocks in a grid was larger than the average block size then the blocks on the grid were further split. The initial decomposition was then partitioned with the Recursive Spectral Bisection method. Then, the partitions were refined recursively in five steps. Again, in each step only grids with blocks larger than the average size were split, and the partitions were refined with the local refinement method [34]. Finally, blocks within the same partitions and subgrids were merged. This strategy is robust and gives good results for all numbers of processors without any tuning of the parameters. We can though affect the results, mainly by changing the limit of the number of blocks in the initial decomposition and the number of refinement levels. Increasing any of these two parameters gives a better load balance but a higher number of blocks and then more serial overhead in the solver, see Figure 15. From the partitioning point of view it is arbitrary which to increase



(a) Load imbalance ratio Λ.

(b) Number of blocks.

Figure 15: Controlling of the multi-level *graph* partitioning. In the different strategies, (i, j), we initially split the grids until the number of blocks is i times higher than the number of processors before we partition the blocks. Then, we refine the partitions j times. The more blocks we have the better load balance we get.

but considering the run time it is more favorable to have more refinement levels than split levels in the initial decomposition. For a large number of blocks the arithmetic complexity in the Recursive Spectral Bisection method increases very rapidly while the complexity of the refinement method is proportional to the number of elements on the partition boundaries.

4.3 Elastic wave scattering

Our final example is somewhat different. It is an example of an extreme case where none of the strategies above would be optimal and other strategies are preferable . The application we are looking at is scattering of linear waves. For example, this could be a seismic wave propagating in a rock with a number of cracks in it. The geometry of the rock is then modeled with a structured base grid and small grids overlapping the base grid are used for the cracks. The equations are solved with explicit finite differences and polynomial interpolation is used to couple the solution between the grids. For further details see [10]. Again, we want to decompose the grids and let the processors work on a part of the data arrays. However, the cracks in the domain do complicate the situation. We get a domain with a varying workload. Depending on the the sizes and the number of the cracks we can use different strategies for the partitioning of the grids.

- *Composite grid partitioning* If the cracks are relatively large then we can use the described partitioning methods for composite grids. Still, the sizes of the overlapping grids will be much smaller than the base grid. Depending on the number of processors and the number of grids we can then either use the *graph* partitioning strategy or the processor *clustering* strategy. Graph partitioning works well for the most cases while the processor clustering is more efficient when the number of processors is much larger than the number of grids. The requirement for the processor clustering strategy to work efficiently for this application is that the computations can be balanced with at least one processor assigned to each crack grid.

- *Domain partitioning* A slightly different partitioning approach would be to treat the problem as a single domain with irregular workload and partition the base grid accordingly. That is, we set up a workmap for the domain and put extra weights at the grid points in the base grid were the cracks are centered. The computations on the cracks are carried out locally on the processor owning the centerpoint. This approach is suitable when the crack grids are very small compared to the base grid. We can use some standard partitioning method for this problem, the *Recursive Coordinate Bisection* method should work well. The standard method can be improved by including the communication cost in the method, as in [32]. The gain will depend on the relative communication cost. On a computer with a slow network and fast processors, for example a cluster of workstations, it can be worth including the communication cost. But for example on a Cray T3E with a high bandwidth and low latency the gain will be negligible, i.e. the communication cost is of the same order as the computation cost.

- *Irregular partitioning* Finally, if there are many small cracks irregularly distributed we can compare this with the parallelization problem in data assimilation, see [32]. In meteorological data assimilation we have many irregularly distributed observations, more frequent over land than water. The *Possessive Greedy* method proved to give good results for this kind of problems. With this method we partition the base grid in equally sized rectangles and distribute the cracks to the processor with a Greedy-like method that as a first choice keeps the cracks locally on the processors owning the corresponding part of the base grid. An advantage with this strategy is that base grid is regularly partitioned and the corresponding communication can be handled efficiently. For the interpolation the communication becomes global and irregular but this communication is small compared to communication on the base grid.

The quickest and the easiest solution to the partitioning problem is probably the second approach. We treat the partitioning problem as a domain with irregular workload and use the framework described in [33] to do the actual partitioning work. We first set up a homogeneous workmap for the whole domain and then include the cracks by increasing the workfactors in the base grid at the center points of the cracks. The connectivity between the crack grids and the base grid is then defined as links from the interpolation points to the center point, we assume that the processor which owns the center point is also doing all the calculations on the crack grid and the interpolation between the crack grid and the base grid. A sample of partitioning experiments are shown in the Figure 16. If the cracks are evenly distributed in the domain good results are obtained with a simple block partitioning with equally sized blocks, see Figure 16(a). The grid is 101x101 but is periodic in the y-direction. If the cracks are unevenly distributed we must use a load balancing algorithm, for example the Recursive Coordinate Bisection method, see Figure 16(b). For some settings it can be worth finding a tailor-made solution, compare the partitionings in Figure 16(c) and Figure 16(d). This is somewhat exaggerated but we get the same effect if the cracks are small and have the same orientation.

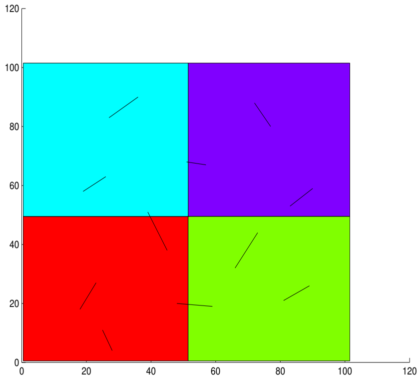
5 Software

Here we present a set of software tools that support the framework discussed in the present paper, give examples of how the tools can be used, and indicate how the tools can be integrated in a software environment to solve partial differential equations on composite grids.

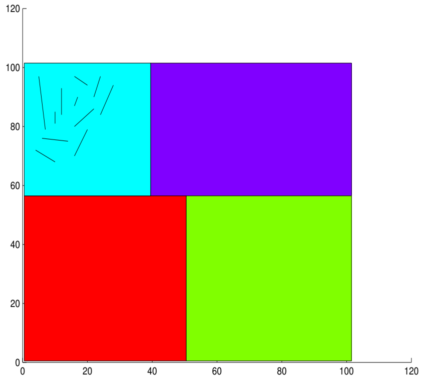
5.1 Design

For unstructured grid partitioning methods there exist a number of general software packages, for example *Chaco* [16] and *Metis* [20]. This is not the case for composite grids. The software for partitioning is usually integrated in the parallel solver environment and can not easily be extracted and adapted to other problems or applications. The algorithms in this paper for partitioning composite grids have been implemented in Fortran 90 with an object-oriented design and encapsulated in a separate Fortran 90 module. The object-oriented design encapsulates both the data structures and the functionality in one entity, a class with a clear interface. The implementation of a class is then independent of the implementation of the other classes making the programs more easy to maintain or adapt to other environments. Moreover, the class names can be chosen from the application domain increasing the readability of the programs and making it easier to communicate with the application experts.

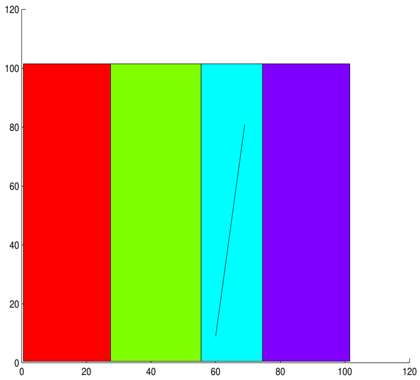
We want a functionality in the classes that enables a flexible design of partitioning algorithms and also meets the need of different users. Thus, the classes contain low-level routines with parameters that can be controlled by the user.



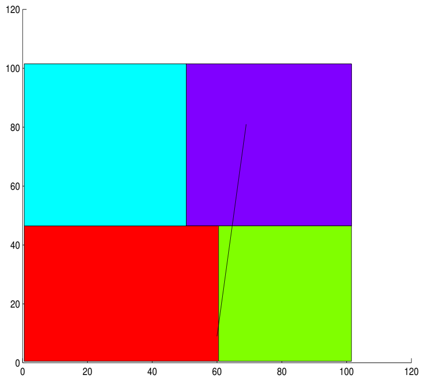
(a) Uniform partitioning, edge-cut 317, load Λ=1.00746



(b) Recursive coordinate bisection, edge-cut 303, load Λ=1.0158



(c) Recursive coordinate bisection only in the x-direction, edge-cut 303, load Λ=1.0224



(d) Recursive coordinate bisection, edge-cut 396, load Λ=1.0141

Figure 16: *Domain partitioning* of the wave scattering grids. We treat the problem as a domain with an irregular workload and partition only the base grid. The cracks are considered as perturbations on the otherwise homogeneous workload.

The routines can be combined in a flexible way to form a partitioning algorithm. The partitioning software consists of three classes, *graph, domain decomposition (dd)*, and *composite domain decomposition (cdd)*. The class *graph* needs a connectivity matrix (Laplacian matrix) and the vertex weights to produce a balanced partitioning of the vertices. Here we have implemented the Recursive Spectral Bisection method, the Greedy partitioning, and our refinement algorithm from [34]. The encapsulation of the *graph* class in a separate module makes it easy to implement other graph partitioning methods for composite grids or to connect the class, for example, to Chaco and use their highly advanced methods for our applications. The data structures within the *graph* class can now be adapted and modified without affecting the other classes. The classes *cdd* and *dd* needs to be associated to a composite grid and a grid respectively. To decouple the partitioning module from any grid module or application we use simple data objects, *cgdata* and *griddata*, for this communication. Information is passed to and from the partitioning module through these communication objects. The class *cdd* is an aggregate of several instances of the class *dd*. It handles the connections between the subgrids and includes the overarching partitioning strategies. In addition, we have the functionality from the class *dd*. The class *dd* is a further development of the routines in [33], (the main change is a different design of the interfaces). This class contains routines for partitioning a single structured grid with an inhomogeneous workload. We have demonstrated the functionality of the single grid partitioning class with good results on a real-life problem, the Baltic sea [33].

5.2 Demonstration examples

The idea is to have the ability to let the user compose her own partitioning algorithm that adapts to her application. In addition, we also provide a set of more or less automatic partitioning methods, see Example 1.

Example 1 Automatic partitioning of a composite grid. The processors are divided in clusters, one for each subgrid, and the grids are then partitioned within a cluster with the recursive coordinate bisection method.

```
subroutine partition(cg,np)
type(cgdata),intent(inout)::cg
integer,intent(in)::np
type(cdd_obj)::cdd

call createCDD(cdd,cg)
call clusterCDD(cdd,method='rcb',np)
call finalizeCDD(cdd)
call deleteCDD(cdd)
```

The user is responsible to provide information of the composite grid, the *cg* object. The *finalize* method fixes the final partitioning and fills the *cg* object with partitioning data. The user can then extract the partitioning data from *cg*.

A more advanced partitioning algorithm usually requires some control of the involved parameters. For example, the multilevel partitioning in Figure 6 requires some control of the initial partitioning, edge-weights within and between the subgrids, the number of refinement levels, and the load balance control in the refinement algorithm, to perform optimally for the specific application. The user can then tune in these parameters for her problem, see Example 2.

Example 2 Multilevel partitioning of a composite grid. The element grids are initially divided uniformly in rectangular blocks and the recursive spectral bisection method is applied to set up the blocks in different partitions. Then, the blocks are split and the partitioning is refined. Finally, small blocks on the same partitions are merged.

```
subroutine partition(cg,p1,p2,levels,mode,Wo,Wi,limit,load,np)
type(cgdata),intent(inout)::cg
integer,dimension(ng),intent(in)::p1,p2
integer,intent(in)::levels,mode,np
real,intent(in)::Wo,Wi,limit,load

call createCDD(cdd,cg)

!  Initial partitioning, for each subdomain
!  split uniformly in rectangular blocks
do i=1,cg%ng
   call domainCDD(cdd,i,dd)
   call uniformDD(dd,p1(i),p2(i))
end do
call rsbCDD(cdd,np,mode,Wo,Wi)

!  Refine the partitioning
do i=1,levels
   do j=1,cg%ng
      call domainCDD(cdd,i,dd)
      call splitDD(dd,limit)
   end do
   call refineCDD(cdd,mode,Wo,Wi,load)
end do

!  Merge blocks on the same partitions
do i=1,cg%ng
   call domainCDD(cdd,i,dd)
   call mergeDD(dd)
end do

call finalizeCDD(cdd)
call deleteCDD(cdd)
```

We have a number of control parameters: `p1(i)` and `p2(i)` – initial decomposition of grid `i`, `levels` – number of refinement levels, `mode` – how to choose the edge-weights, `Wo` and `Wi` – the edge-weights between and within the subgrids respectively, `limit` – split limit parameter, `load` – workload limit in the refinement. Some of the parameters are discussed in [33] and [34].

As we can see from the examples above, the user has a great flexibility to design her own partitioning method but can also rely on standard methods.

The partitioning classes give support for implementing partitioning algorithms for composite grids and it is straightforward to compose a highly advanced partitioning algorithm for a specific application.

5.3 Cogito

With the increasing computing power the need for advanced software techniques arises inevitably. More complicated problems with more advanced solution techniques and with high degree of automation can then be solved, for example high accuracy flow calculations around complicated geometries. This makes the software more and more complicated. Object-oriented modeling and design provides a way of describing and thinking about problems by using real-world concepts. It promotes for better understanding, cleaner designs, and more maintainable systems. Traditional techniques in scientific computing focus too heavily on low-level programming steps and the implementation remains procedure-oriented. The result is that the low-level implementation details becomes visible at the top-level of the programs. This makes the programs more complicated to read, maintain, and adapt to other applications or move to other computer platforms. *Cogito* [44] is a system of software tools, written in Fortran 90 with an object-oriented design, for PDE-solvers on composite grids. A number of projects with similar goals exist, see for example *Diffpack* [6], *PETSc* [3], and *Overture* [5]. These are C/C++ implementations. Even though Fortran 90 is not a fully object-oriented language it gives support for object-oriented programming through the *type*, *module* and *interface* concepts [23, 28].

The partitioning module for composite grids will be a part of the Cogito system, but it can also be used separately. The partitioning module is supposed to work with the classes in the grid-level [31] of Cogito, see Figure 17. Here, the class *Composite Grid (CG)* is an aggregate of the classes *Interpolation Table (IPT)* and *Grid (G)*. *Grid Functions (GF)* are associated to both the composite grid and the single grids. A *Distribution (DIST)* is associated to each grid. Grid functions are then distributed accordingly. A distribution is computed with the help of the classes in the partitioning module. The objects of the classes communicate through fixed interfaces with simple data structures, indicated with arrows in the figure. These data structures are encapsulated in communication objects, *cgdata, griddata, laplmatrix, weights,* and *part_array.* A similar approach – using communication objects – is also taken in [6].

A PDE is solved by applying mathematical/numerical operations on the grid functions, i.e. *Dplus, Add, Scale*, etc. If a distribution is attached, the operations are parallelized and the details with message passing are hidden within the class. Consequently the high level solver becomes independent of the partitioning and other parallelization details.

6 Conclusions

We have outlined a general strategy for partitioning composite grids, irregularly coupled structured meshes. Within the framework set by the general strategy

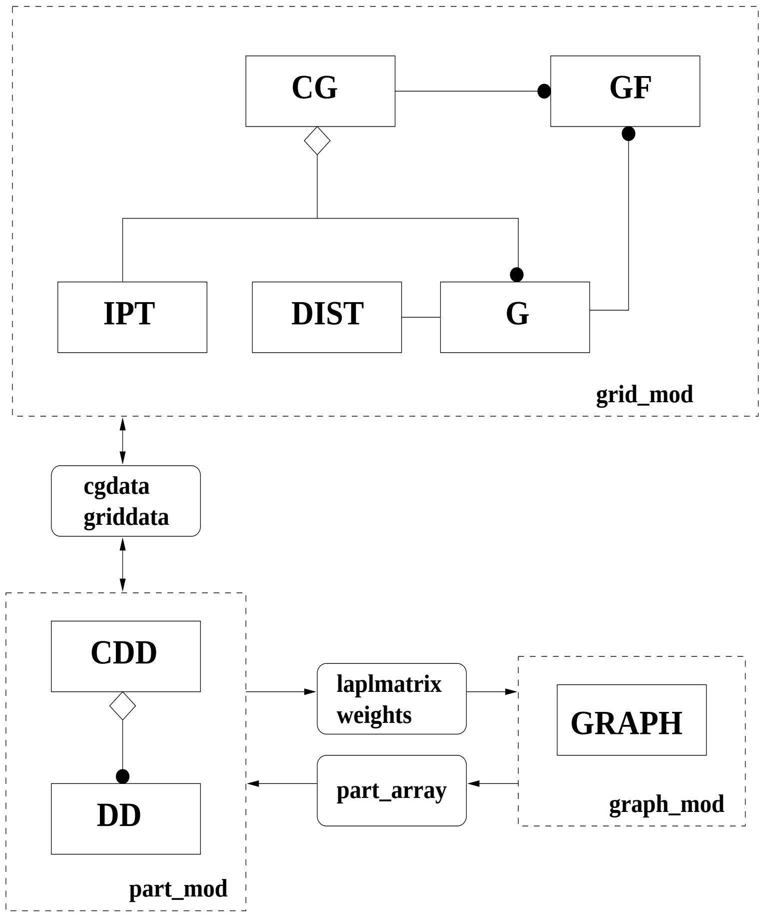


Figure 17: System design partly using the Object Modeling Technique [35]. We have a module (dashed frame) for programming PDE-solvers on composite grids, *grid_mod*, a module for partitioning composite grids, *part_mod*, and a module for graph partitioning, *graph_mod.* The separate modules communicate through simple communication objects, *cgdata, griddata, laplmatrix, weights,* and *part_array* which also are encapsulated in their own module.

we have identified four important special cases, the *block, single, graph* and *cluster* partitioning strategy. These strategies have been evaluated on two different multiblock grids. One of the applications is a multi-element airfoil used in real aerodynamical computations in industry while the other is a simple model problem. From the case studies we can conclude that, (i) the block strategy is only applicable if the number of grids is much larger than the number of processors, (ii) the cluster strategy is optimal if the number of processors is much larger than the number of grids, (iii) the single grid partitioning strategy is only efficient if the grids are large and the number of processors is modest, (iv) the graph strategy gives an all over good performance. The extension of these strategies to hierarchical overlapping grids is straightforward. Only the algorithm to set up the connections between the subgrids has to be changed.

For the *cluster* strategy we have given an optimal processor clustering algorithm. This algorithm can also be used in HPF-2 to define optimal processors subspaces. For the *graph* strategy we have developed a very flexible multilevel partitioning algorithm.

Moreover, we have developed a set of software tools for carrying out the partitioning. Some automatic partitioning methods are included in the software. However, our main ambition is not to provide universal solutions to the partitioning problem but to give the user the ability to easily create her own partitioning strategy that adapts to her own problem. The software tools mainly consist of low level operations that can be combined flexibly to create different partitioning strategies within the framework discussed here. Thus, these tools allow for the implementation of a large class of partitioning strategies for composite grids.

References

[1] A. Ålund, P. Lötstedt, M. Sillén, *Parallel single and multigrid solution of industrial compressible flow problems*, Computers & Fluids, 26:775-791, 1997.

[2] G. Agrawal, A. Sussman, J. Saltz, *An integrated runtime and compile-time approach for parallelizing structured and block structured applications*, ICASE report no. 93-77, NASA Langley Research Center, Hampton, Virginia, 1993.

[3] S. Balay, W.D. Gropp, L. Curfman McInnes, B. F. Smith, *Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries*, in Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, H. P. Langtangen editors, Birkhäuser, Boston, 1997.

[4] M.J. Berger, S. Bokhari, *A partitioning strategy for non-uniform problems on multiprocessors*, ICASE report no. 85-55, NASA Langley Research Center, Hampton, Virginia, 1985.

[5] D.L. Brown, W.D. Henshaw, D.J. Quinlan, *Overture: An Object-Oriented Framework for Solving Partial Differential Equations*, in Proceedings of IS-

COPE 97: Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science 1343, Springer-Verlag, 1997.

[6] A. M. Bruaset, H.P. Langtangen, *Object-Oriented Design of Preconditioned Iterative methods in Diffpack*, ACM Transactions on Mathematical Software, vol 23, no 1:50-80, March 1997.

[7] C. Chase, K. Crowley, J. Saltz, A. Reeves, *Parallelization of Irregularly Coupled Regular Meshes*, ICASE report no. 92-1, NASA Langley Research Center, Hampton, Virginia, 1992.

[8] G. Chesshire, V.K. Naik, *An Environment for Parallel and Distributed Computation with Application to Overlapping Grids*, IBM, T.J. Watson Research Center Yorktown Heights, NY 10598, 1993.

[9] Y.P. Chien, F. Carpenter, A. Ecer, H.U. Akay, *Load Balancing for Parallel Computation of Fluid Dynamics Problems*, in Proceedings of Parallel Computational Fluid Dynamics-New Trends and Advances, Ecer et al. editors, Elsevier Science B.V., Amsterdam, 1995.

[10] D.B. Duncan, *Difference approximations of acoustic and elastic wave equations*, Proceedings of 1995 Workshop on Numerical Methods for Wave Propagation Phenomena, Manchester, UK, 1996.

[11] A. Ecer, H.U. Akay, Y.P. Chien, N. Gopalaswamy, *Load Balancing Issues in Parallel Computing*, in proceedings of Sixth International Symposium on Computational Fluid Dynamics, Vol 1:275-280, Lake Tahoe, Nevada, 1995.

[12] R. Enander, *Grid patching and residual smoothing for computations of steady state solutions of first order hyperbolic systems*, Thesis, Dept. of Scientific Computing, Uppsala University, Uppsala, 1991.

[13] D.J. Evans, W.U.N. Butt, *Load balancing with network partitioning using host groups*, Parallel Computing, 20:325-345, 1994.

[14] S.J. Fink, S.B. Baden, *Runtime Support for Multi-tier Programming of Block-Structured Applications on SMP Clusters*, in Proceedings of ISCOPE 97: Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science 1343, Springer-Verlag, 1997.

[15] B. Hendrickson, R. Leland, *A Multilevel Algorithm for Partitioning Graphs*, in Proceedings of Supercomputing '95, 1995.

[16] B. Hendrickson, R. Leland, *The Chaco User's Guide Version 2.0*, Technical Report, Sandia National Laboratories, Albuquerque, NM, July 1995.

[17] J. Häuser, H. Wong, H.G. Paap, W. Gentzsch *Parallel computing in aerospace using multiblock grids. Part 1: Application to grid generation*, Concurrency, Vol 4:357-376, 1992.

[18] M.A. Iqbal, J.H. Saltz, S.H. Bohkari, *Performance tradeoffs in static and dynamic load balancing strategies*, Technical Report 86-13, ICASE, NASA Langley Research Center, Hampton, VA, 1986.

[19] G. Karypis, V. Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Technical Report, University of Minnesota, Department of Computer Science, Minneapolis, 1995.

[20] G. Karypis, V. Kumar, *Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering System*, Technical Report, University of Minnesota, Department of Computer Science, Minneapolis, 1995.

[21] J. De Keyser, K. Lust, D. Roose, *Run-time load balancing support for a parallel multiblock Euler/Navier-Stokes code with adaptive refinement on distributed memory computers*, Parallel Computing, 20:1069-1088, 1994.

[22] L. Liebrock, K. Kennedy, *Automatic Data Distribution for Composite Grid Applications*, Scientific Programming, vol. 6, pp. 95-113, 1997.

[23] L. Machiels, M.O. Deville, *Fortran 90: An Entry to Object-Oriented Programming for the Solution of Partial Differential Equations*, ACM Transactions on Mathematical Software, vol 23, no 1:32-49, March 1997.

[24] F. Manne, T. Sørvik, *Structured Partitioning of Arrays*, Technical report CS-96-119, Department of Informatics, University of Bergen, Norway, 1996.

[25] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tennessee, Knoxville, Tennessee, 1994.

[26] J. Mushtaq, A.J. Davies, *A multiblock solution process for the equations of fluid dynamics on a network of transputers*, Advances in Engineering Software, vol 26, no 2:133-149, 1996.

[27] C. de Nicola, R. Tognaccini, P. Visingardi, *Multiblock Structured Algorithms in Parallel CFD*, in Proceedings of Parallel Computational Fluid Dynamics-Implementations and Results using Parallel Computers, Ecer et al. editors, Elsevier Science B.V., Amsterdam, 1996.

[28] C.D. Norton, *Object-Oriented Programming Paradigms in Scientific Computing*, Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, New York, 1996.

[29] P. Olsson, J. Rantakokko, M. Thuné, *Software tools for parallel CFD on composite grids*, in Proceedings of Parallel Computational Fluid Dynamics-Implementations and Results using Parallel Computers, Ecer et al. editors, Elsevier Science B.V., Amsterdam, 1996.

[30] E. Pärt-Enander, Overlapping Grids and Applications in Gas Dynamics, Ph.D. thesis, Uppsala University, Uppsala, Sweden, 1995.

[31] J. Rantakokko, *Object-oriented software tools for composite-grid methods on parallel computers*, Technical report 165, Department of Scientific Computing, Uppsala University, Uppsala, Sweden, 1995.

[32] J. Rantakokko, *Strategies for parallel variational data assimilation*, Parallel Computing, Vol 23, Iss 13, pp:2017-2039, 1997.

[33] J. Rantakokko, *A framework for partitioning domains with inhomogeneous workload*, Technical report 194, Department of Scientific Computing, Uppsala University, Uppsala, Sweden, 1997.

[34] J. Rantakokko, *A local refinement algorithm for data partitioning*, Technical report 203, Department of Scientific Computing, Uppsala University, Uppsala, Sweden, 1998.

[35] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[36] M.L. Sawley, J.K. Tegnér, *A Comparison of Parallel Programming Models for Multiblock Flow Computations*, Journal of Computational Physics, 122:280-290, 1995.

[37] M.L. Sawley, J.K. Tegnér, C.M. Bergman, *A serial data-parallel multiblock method for compressible flow*, in Proceedings of Parallel Computational Fluid Dynamics-New Trends and Advances, Ecer et al. editors, Elsevier Science B.V., Amsterdam, 1995.

[38] K. Shimano, C. Arakawa, *Numerical Simulation of Incompressible Flow on Parallel Computer with the Domain Decomposition Technique*, in Proceedings of Parallel Computational Fluid Dynamics-New Algorithms and Applications, Satofuka et al. editors, Elsevier Science B.V., Amsterdam, 1995.

[39] H.D. Simon, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering, 2:135-148, 1991.

[40] E. Steinthorsson, D. Modiano, *Advanced Methodology for Simulation of Complex Flows Using Structured Grid Systems*, Report ICOMP-95-28, Institute for Computational Mechanics in Propulsion, Cleveland, Ohio, 1995.

[41] M. Thuné, *Straightforward partitioning of composite grids for explicit difference methods*, Parallel Computing, 17:665-672, 1991.

[42] M. Thuné, *The partitioning problem for a class of data parallel algorithms*, Parallel Computing 18:867-878, 1992.

[43] M. Thuné, *Partitioning strategies for composite grids*, Parallel Algorithms and Applications, vol 11:325-348, 1997.

[44] M. Thuné, E. Mossberg, P. Olsson, J. Rantakokko, K. Åhlander, K. Otto, *Object-oriented construction of parallel PDE solvers*, in Modern Software

Tools in Scientific Computing, E. Arge, A. M. Bruaset, H. P. Langtangen editors, Birkhäuser, Boston, 1997.

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

[46] D. Quagliarella, *Optimal Domain Decomposition for Parallel Multiblock Flow-field Solvers using Genetic Algorithms*, in Proceedings of Parallel Computational Fluid Dynamics-New Algorithms and Applications, Satofuka et al. editors, Elsevier Science B.V., Amsterdam, 1995.