Partitioning strategies for composite grids *

Michael Thuné
Department of Scientific Computing
Uppsala University
Box 120
S-751 04 Uppsala, Sweden

michael@tdb.uu.se

May 1996

Abstract

Two different strategies are discussed, for the partitioning of composite grids, in the context of parallel computers of MIMD type with distributed memory. The two strategies are compared theoretically, with respect to arithmetic load balance and communication overhead. Moreover, the ability to handle dynamically changing grids is considered.

Composite, structured grids are the natural choice for finite difference methods on complicated geometries, which can not be covered by a single, structured grid. A composite grid is a union of (in general curvilinear) structured grids. So called multi-block grids, frequently used, e.g., in aerodynamical computations, constitute a special case. The discussion in the present paper is valid also if the blocks are unstructured, and for hybrids such as dragon grids.

Composite grids are more complicated to partition than single grids. The present paper points out the difficulties in detail, giving some quantification of the overhead that may result from a naive implementation of composite grid partitioning. These results indicate what are the crucial implementation issues in the present context.

Keywords: partition, data distribution, composite grid, multi-block, finite difference, parallel computing

1 Introduction

1.1 Background

The development of parallel high-performance computers has been motivated to a large extent by the demands raised by computationally heavy applications, such as fluid dynamics, combustion, and electromagnetics. The mathematical

*The research described in this paper has been supported by a faculty grant from Uppsala University.

models in these cases are systems of time-dependent partial differential equations. One of the dominating classes of numerical methods for such systems is finite difference methods. In this context, it is standard to use a data parallel approach. Parallelization is equal to partitioning and mapping of the computational grid.

Finite difference methods are related to structured grids. The partitioning of a single structured grid is straightforward [16]. However, single grids are not sufficient in realistic applications. Normally, in order to cover a complicated geometry, it is necessary to use *composite* structured grids, for which partitioning is less obvious.

A composite grid is a union of (in general curvilinear) structured grids. The simplest variant is so called multi-block grids (e.g., [6]), where the different grids (or blocks) meet edge to edge, with no overlap. A more general approach is to have overlap between blocks (e.g. [1], [3], [14], [15]). A straightforward extension is then to allow for a hierarchy of blocks. First, a set of blocks is laid out to cover the computational domain. Then, smaller blocks are inserted locally, for example to improve the resolution in the neighbourhood of a discontinuity. This can be repeated, so that further levels of local refinement blocks are introduced. In the following, the individual blocks will be referred to as *element grids*.

In applications like those mentioned above, it is often appropriate to adapt the grid dynamically to changes in the solution (for example, in order to resolve sharp gradients). This can be achieved by inserting new refinement blocks on top of the initial set of element grids.

The different element grids will interact with each other via interpolation. We consider parallel computers of MIMD type with distributed memory. On such platforms, the exchange of interpolation data between grids will lead to communication between processors. The communication pattern for this interaction will be irregular (in contrast to the communication pattern for interaction between different partitions of a single element grid). This makes the partitioning problem more intricate for composite than for single grids.

1.2 Partitioning strategies for composite grids

In general, most previous work treats *algorithms* for partitioning of single grids. In the present contribution, such algorithms are assumed to be available as basic building blocks. Our focus is on *strategies* for utilizing these algorithms in the context of composite grids. In fact, the ideas are not restricted to structured grids. We could equally well assume composite, *unstructured* grids. This means that our results also apply to hybrids, such as *dragon* grids [9].

For the partitioning of composite grids, two general strategies are conceivable:

Strategy 1 (Generalized single-grid partitioning) Each element grid is partitioned over *all* processors.

Strategy 2 (Clustering) Clusters of element grids are partitioned over clusters of processors.

(Note that Strategy 1 can be regarded as a special case of Strategy 2, with *one* single cluster.)

Strategy 1 was studied in previous work ([17], [18]). It is relatively straightforward to apply, given a partitioning algorithm for single grids. Most important, Strategy 1 is very well suited for *dynamic* grid structures. As each grid is distributed over all processors, the addition or removal of a grid will not at all affect the *arithmetic* load balance (i.e. disregarding communication).

The major *deficiency* of Strategy 1 is that it disregards neighbour relations within the composite grid. Consequently, the communication pattern for interpolation between element grids will be very irregular. The primary reason for introducing Strategy 2 is to avoid this drawback.

Another reason is that Strategy 1 is less suitable on massively parallel computers, since it may then not be feasible to partition a small grid over all processors.

The purpose of the present investigation is to compare the two strategies theoretically. We aim at a nuanced evaluation, pointing at the advantages and drawbacks of each strategy. The important issues of arithmetic load balance and communication overhead are discussed. First, non-hierarchical composite grids are considered, then the results are generalized to hierarchical grids.

In the latter case, Strategy 2 can be interpreted in different ways. We propose four different variants. Their relative pros and cons are discussed.

For Strategy 2, the relation between number of processors, p, and number of element grids, n, is of importance. If $p > n$, then it is natural to assign a group of processors to each element grid. In case $p < n$, the task is rather to assign a group of element grids to each processor. In the present paper, we consider the first case, i.e. processor clustering. We will show that for this approach to be competitive, it is essential to allow for a non-integer number of processors in each cluster. Moreover, with Strategy 2 the mapping problem comes into focus, whereas in the context of Strategy 1 mapping can be considered as being implicit.

1.3 Related work

There is a vast literature on the partitioning problem, but few authors have discussed the partitioning problem for composite grids. The most interesting example from our point of view is LOCO [4], which is a framework for data parallel computing. It can for example handle multi-block codes, also in the dynamic case.

The literature on partitioning algorithms (or load balancing algorithms) for single grids, can broadly be divided into three groups:

- *General graph partitioning algorithms.* Here, the execution time formula is disregarded. Instead, the different types of work load are captured as weights on nodes and edges, and the graph is partitioned in order to balance the work load. Important contributions are [2], [20], [11]. For unstructured-grid problems, the Recursive Spectral Bisection (RSB) method by Pothen et al. [10] is currently considered as state of the art.

However, this algorithm is relatively time-consuming, which is problematic in the dynamic case. A remedy to this can be to design an iterative repartioning algorithm, which takes the original RSB partitioning as initial guess [19].

- *Graph partitioning algorithms for structured grids.* Given a structured grid, it is possible to design specially tailored algorithms, which are more efficient than the general ones. One group of algorithms are variants of the Greedy algorithm [8]. The straightforward approach which I have analyzed is another example [17], [18]. A similar approach is adopted in High Performance Fortran [7].

- *Heuristic dynamic load balancing.* A third type of load balancing is related to dynamic scheduling algorithms. Here, some initial distribution of work is carried out, possibly using one of the previously mentioned algorithms. Then, one or more schedulers are introduced, and they take the responsibility for migrating tasks between processors. In the data parallel context, the task will correspond to a set of data to be operated on. Processors will send reports to the scheduler(s) about their load, and on the basis of this information the scheduler will make migration decisions. This type of techniques is reviewed in [12].

2 The two strategies in the context of non-hierarchical composite grids

For a comparison between the two partitioning strategies, the following aspects are relevant:

1. Arithmetic overhead due to load imbalance.

2. Communication overhead.

3. The ability to handle *dynamic* grid structures, i.e. where element grids are added and deleted as the computation evolves.

(Implementation aspects could have been added. However, we leave them as a topic for separate study.)

It is apparent that Strategy 1 is good for dynamic grid structures, whereas Strategy 2 is more problematic from this point of view. For the moment, we will focus on the other two aspects, where Strategy 2 is expected to have advantages over Strategy 1. We consider twodimensional grids. The extension to 3D is straightforward and does not alter the conclusions.

First, non-hierarchical composite grids will be considered. We will synonymously use the term *multi-block* grids. It is not required that the grid lines match each other at grid interfaces. Thus the issue of interpolation between grids is present also in the multi-block context.

Following [13], we define arithmetic overhead, $R_a(p)$ and communication overhead, $R_c(p)$, as follows:

$$R_a(p) = p\frac{t_a(p)}{t(1)} - 1, \tag{1}$$

$$R_c(p) = p\frac{t_c(p)}{t(1)}. \tag{2}$$

Here, $t_a(p)$ is the time for arithmetic work, and $t_c(p)$ the time spent on communication, when the algorithm is executed on p processors. $t(1)$ is the execution time for the same algorithm on a single processor. We assume a fixed problem size M.

The definitions of overhead are related to efficiency. Use the common definition of efficiency (based on algorithmic speedup):

$$E(p) = \frac{1}{p}\left(\frac{t(1)}{t_a(p) + t_c(p)}\right).$$

This can be formally rewritten as:

$$E(p) = \frac{1}{1 + R_a(p) + R_c(p)}. \tag{3}$$

2.1 Arithmetic load balance

In order to measure the degree of imbalance in arithmetic work, we introduce an imbalance ratio Λ:

$$\Lambda = r_{\max}/r.$$

Here, $r_{\max}$ is the work per processor in the most heavily loaded processor cluster. The average load is denoted by r. Consequently, $\Lambda = 1$ means perfect balance. In fact, $\Lambda = 1 + R_a(p)$, since, in our context, $t(1) = prt_f$, and $t_a(p) = r_{\max}t_f$, where t_f is the average time for the execution of a floating point operation.

2.1.1 Arithmetic load imbalance within a cluster

There are two sources of arithmetic load imbalance in our context. One is the distribution of load *within* a cluster. This is the only source of imbalance if Strategy 1 is used (the "cluster" then being the complete set of available processors). With Strategy 2, the distribution of load *between* clusters is an additional source of imbalance.

Consider first the distribution of load within a cluster. This means studying the distribution of a single element grid over a cluster of p_c processors, where $p_c = p$ for Strategy 1, $p_c < p$ for Strategy 2. For both strategies, there has to be an algorithm for the single-grid distribution. The load balance will depend on the choice of this algorithm. However, this choice is not our concern for the present. We want to study effects that are purely due to differences between Strategy 1 and Strategy 2. Thus, we focus on a case where there exists an (almost) optimal single-grid distribution.

Definition 1 *The* work factor *at a grid point is the number of arithmetic operations necessary in order to compute the grid function values at that grid point.*

Consider an element grid with work factor ω at all grid points. In this case, the *normal* partitioning will be optimal:

Definition 2 *Given a partition shape, the* normal partitioning *is the set of partitions obtained by distributing the grid points as evenly as possible over the partitions, one partition per processor, and minimizing the lengths of the partition interfaces.*

The two natural choices of partition shapes in 2D are slices (partition across one dimension) and rectangles (partition across both dimensions). Then, the minimization of partition interface lengths is trivial: make slices across the longest side of the grid, and make the rectangles as square as possible, respectively.

Let the element grid have dimensions $N_x \times N_y$. Let the imbalance ratios be Λ_1 for Strategy 1, Λ_2 for Strategy 2. The normal partitioning with slices makes p_c slices across the side with length $N_* = \max(N_x, N_y)$. For Strategy 1, $p_c = p$. For Strategy 2 let the largest value of p_c, for any cluster, be denoted by $\bar{p_c}$.

Proposition 1 *In the case of normal partitioning with slices, and assuming $\bar{p_c} << p$ (which is the typical situation, although exceptional cases can be envisaged) the two imbalance ratios will be approximately equal when $N_* >> p$.*

In the worst case, $N = p + 1$, we have*

$$\Lambda_1 \approx 2\Lambda_2.$$

Proof Assuming $N_* \bmod p_c \neq 0$, some slices will get $s_{\max} = \lfloor N_*/p_c \rfloor + 1$ grid lines, whereas the other slices get $\lfloor N_*/p_c \rfloor$ grid lines. For perfect load balance, we would need $s = N_*/p_c$ grid lines per partition. Since we assume equal work factors at all grid points, we get $\Lambda = s_{\max}/s$. With $N_* >> p$ this quantity is close to 1, for both Strategy 1 and Strategy 2.

The worst case (for Strategy 1) occurs when $N_* = p + 1$. Then, Λ_1 is close to 2, whereas Λ_2 will remain close to 1. $\Box$

Remark: As rectangles are obtained by slicing in two directions, a similar result is obtained in that case, the factor being 4 instead of 2.

Proposition 1 applies when $N_* > p$, and then speaks in favour of Strategy 2. However, in particular on massively parallel computers, it is likely that there are element grids with $N_* < p$. Thus, the relevant conclusion is that there is need for a mechanism for deciding the *maximal* number of processors that is reasonable to assign to a certain grid. The following proposition gives guidelines for this decision.

Proposition 2 *Assume an element grid as above. Assume that the normal partitioning is to be applied, to distribute the grid over p processors. Two different partition shapes are considered: slices and rectangles. Assume that in the latter case perfect squares can be formed.*

Let ρ be the maximal arithmetic overhead that can be accepted, in the sense that the efficiency must not be lower than $1/(1+\rho)$. Sufficient conditions for this to hold are:

$p \leq \rho N_*$ *for slices,*

$p \leq (\sqrt{1+\rho}-1)^2 N$ *for squares.*

Proof We prove the proposition only for squares. The proof for slices is straightforward.

The rectangles are formed by making p_x slices in the x-direction and p_y slices in the y-direction (where $p_x p_y = p$). Then, in the worst case:

$$R_a(p) = \Lambda - 1 = \frac{(\lfloor N_x/p_x \rfloor + 1)(\lfloor N_y/p_y \rfloor + 1)}{N/p} - 1 < p/N + p_y/N_y + p_x/N_x.$$

Now, introduce the aspect ratio $\phi = \max(N_x, N_y)/\min(N_x, N_y)$. Furthermore, assume (without loss of generality), that $\max(N_x, N_y) = N_x$. Then, the previous expression can be reformulated as

$$R_a(p) < p/N + p\phi/(p_x N_x) + p_x/N_x.$$

Squares imply that $p_x = \sqrt{\phi p}$. Moreover, note that $N_x = \sqrt{\phi N}$. Consequently,

$$R_a(p) < p/N + 2\sqrt{p/N}.$$

From this follows that $R_a(p) \leq \rho$ if $\sqrt{p/N} \leq \sqrt{1+\rho} - 1$, which completes the proof. □

An example: assume that $\rho = 0.1$ (i.e. an efficiency of 91% is acceptable). A square element grid of dimension 100×100 should then be assigned at most 10 processors in the slice case, 23 processors for rectangles. (NB.: The theorem gives sufficient, but not necessary conditions. For example, $p = 25$ gives $R_a(p) = 0$ for the square 100×100 grid.)

Both Strategy 1 and Strategy 2 should take into consideration Proposition 2 (or similar formulae, for other partition shapes). Assuming this modification of the strategies, there will be no significant difference between them, from the aspect of arithmetic load imbalance within clusters.

2.1.2 Arithmetic load imbalance between clusters

Next, consider the additional load imbalance which can appear with Strategy 2, due to round-off in the formation of processor clusters. Let g_i denote element grid i, and p_i^* the ideal number of processors for g_i. In practice, we can not use exactly p_i^* processors, since p_i^* will in general be a real number. After rounding, the number of processors assigned to g_i is p_i. The relative round-off error δ_i is defined by:

$$p_i = (1 - \delta_i)\, p_i^*$$

The imbalance ratio Λ can be expressed in terms of these round-off errors.

Proposition 3 *For Strategy 2, applied to a non-hierarchichal composite grid, the imbalance ratio, due to imbalance of arithmetic load between clusters, fulfills:*

$$\Lambda = 1/\left(1 - \max_i \delta_i\right) \tag{4}$$

Proof Let ω_i be the arithmetic work load related to g_i, and let C_i denote the processor cluster assigned to g_i. Moreover, let r_i be the workload per processor in C_i, i.e.

$$r_i = \omega_i / p_i.$$

Let ω be the total workload (in terms of arithmetic operations) on the composite grid, i.e. $\omega = \sum \omega_i$. The optimal number of processors in C_i is

$$p_i^* = (\omega_i/\omega)\, p. \tag{5}$$

With this many processors, the workload per processor in C_i would be $r = \omega/p$. That is, $\omega_i/p_i^* = r$. The formula for r_i can then be written:

$$r_i = \omega_i/p_i = (\omega_i/p_i^*)\,(p_i^*/p_i) = r/(1 - \delta_i).$$

Consequently, $\Lambda = \max r_i/r = 1/\left(1 - \max_i \delta_i\right)$, which completes the proof. $\square$

With an integer number of processors in each cluster, the right-hand side in (4) can be unacceptably large.

Assume that rounding—of p_i^* to get p_i—is made to d decimals. From the defintion of δ_i, and using (5), we get

$$|\delta_i| \le 0.5 \times 10^{-d}/[(\omega_i/\omega)\, p]. \tag{6}$$

Then, according to (4),

$$\Lambda \le 1/\left(1 - 0.5 \times 10^{-d}/[(\min_i \omega_i/\omega) p]\right).$$

That is, the load imbalance will increase as $(\min_i \omega_i/\omega)p$ decreases. The crucial situations are when p is small, and when some grid is very small (in terms of arithmetic work) in comparison with the total composite grid.

With $d = 0$, i.e. an integer number of processors in each cluster, the following decrease in efficiency can occur, for normal values of $\gamma = 1/[(\min_i \omega_i/\omega)p]$. (For example, $\gamma = 0.9$ means that $(\min_i \omega_i/\omega)p = 1.11$, which is by no means exceptional.)

γ	0.1	0.5	0.9
$E = 1/\Lambda_u$	0.95	0.75	0.55

Table 1: Efficiency as a function of $\gamma = 1/[(\min_i \omega_i/\omega)p]$. Here, Λ_u denotes the upper limit of Λ, and E is the efficiency corresponding to that load imbalance, in the absence of communication overhead.

The conclusion is that Strategy 2 must be implemented to allow for non-integer numbers of processors in the clusters. (One possibility is to use the idea of joint membership [5].) Then, the arithmetic load imbalance between clusters will be acceptable. For example, with $d = 1$, $\gamma = 0.9$, we get $E = 0.95$. However, for exceptional values of γ, the decrease in efficiency can be significant, even if $d = 1$. There has to be additional mechanisms for handling such situations.

2.2 Communication overhead

For a discussion about communication overhead in our context, it is appropriate to make a distinction between two sources of communication.

- *Internal communication.* This is communication for the exchange of data between different partitions of the same element grid. This is normally nearest-neighbour communication and follows a regular pattern.

- *External communication.* This is communication due to the exchange of interpolation data between different element grids. Here, the communication may be non-local, and the communication pattern more irregular.

2.2.1 Internal communication

Consider first the overhead due to internal communication. Such communication takes place *within* a cluster. Use the same notation as in the previous section. Assume that there are n element grids, and assume for simplicity that all grids are of the same size, N. Let $S + T$ be the internal-communication time for *one* partition. This involves at least one message to each neighbour partition. Here, S and T denote the accumulated startup and transfer time, respectively. The transfer time can be expressed as a function of grid size and number of processors, $T = C(N, p)$.

With Strategy 1, each processor will have n partitions, one from each grid. The internal communication can then be implemented in two ways. The *obvious* alternative is to make the communication for one partition at a time. The *optimal* implementation is to collect the communication for *all* partitions into a small number of messages. For example, if the communication network allows messages to be sent in one direction at a time, the internal communication could be handled with only one message in each direction.

The communication overhead $R_c(p)$ (see (2)) is a sum of two terms, $R_s(p)$ and $R_{tr}(p)$, corresponding to message startup and transmission, respectively. We use superindices to distinguish between the overhead for Strategy 1 and Strategy 2.

Assume that all element grids have the same arithmetic work load. This corresponds to an average case, with p/n processors in each cluster.

Proposition 4 *Under the previous assumptions, the following holds. For the obvious implementation of Strategy 1:*

$$R_s^{(1)}/R_s^{(2)} = n.$$

For the optimal implementation of Strategy 1:

$$R_s^{(1)}/R_s^{(2)} = 1.$$

For both implementations of Strategy 1:

$$R_{tr}^{(1)}/R_{tr}^{(2)} = nC(N,p)/C(N,p/n).$$

A startup overhead ratio n is a severe drawback for Strategy 1, since message startup is normally a bottleneck. The optimal implementation is a remedy, but requires more careful programming.

The ratio of transfer overheads is the same for both implementations of Strategy 1. The exact size will depend on the partition shape. For the common shapes in our context, slices and rectangles, the following holds.

Corollary 1 *With assumptions as above,*

$$R_{tr}^{(1)}/R_{tr}^{(2)} = n$$

for slices.

With the additional assumption that the grid has dimensions $\sqrt{N} \times \sqrt{N}$, and that $\sqrt{p}$ is an integer, the transfer ratio is

$$R_{tr}^{(1)}/R_{tr}^{(2)} = \sqrt{n}$$

for the normal partitioning with rectangles.

Proof In the slice case, the partition interface length does not depend on the number of processors. Consequently, $C(N,p/n) = C(N,p)$, and the first part of the corollary follows.

For rectangles, the partition interface length will increase as the number of partitions decreases. With our assumptions, the partition interfaces will have length $\sqrt{N}/\sqrt{p}$. Thus, $C(N,p/n) = \sqrt{n}C(N,p)$, and the second part of the corollary follows. □

2.2.2 External communication

Next, consider the overhead due to external communication. This communication can be organized in different ways. The two extremes are: (a) one message per interpolation point, and (b) one message per processor that needs data from any of my partitions. Obviously, (a) is prohibitively inefficient, and (b) is optimal. However, (b) is relatively complicated to implement.

In our analysis, we will consider an intermediate alternative, procedure (c). Assume the following implementation of this communication. Each processor in the cluster has partitions from n_c grids, one partition per grid. For each partition, the processor sends one set of messages for each grid that needs interpolation data from the partition. Such a set contains one message to each processor that needs data from this partition.

For Strategy 2, in the non-hierarchical case, (c) is equivalent to the optimal (b) (since in this case there is precisely one partition per processor). For Strategy 1, however, (b) would normally be better than (c).

Strategy 1 can be regarded as a special case, with $n_c = n$, $p_c = p$. With Strategy 2, in the current, non-hierarchical case, there is precisely one element grid per cluster, i.e. $n_c = 1$. Moreover, $1 < p_c < p$ for each cluster.

A precise analysis of the external communication is very difficult. In the following we will consider two special cases, a *worst* and a *best* case. This will indicate upper and lower bounds for the actual external communication overhead we can expect to observe in real cases.

Assume that the external communication is implemented according to procedure (c). Assume in addition the following model for the time spent on external communication:

$$t_{ij} = \sum_{k=1}^{e_{ij}} [\alpha + m_{ijk}\beta].$$

Here, t_{ij} is the time spent by processor j in sending interpolation data from its partition of grid i. (Rather: from the grid functions defined on this partition.) e_{ij} is the number of messages being sent for this purpose, and m_{ijk} is the length of message k. α is the message startup time, and β the transfer time per unit length. Finally, assume that each grid has N points.

Since each processor has n_c partitions, the total time spent on external communication will be:

$$t_e(p) = \max_j \sum_{i=1}^{n_c} t_{ij} \leq n_c e_c[\alpha + m_c\beta],$$

where e_c is the maximal value of e_{ij} for any processor in the cluster, and m_c is the maximal message length.

The worst case for e_c is when the processor in charge of a certain partition must send to all processors holding partitions of the adjacent grid. This will be the case for example when two adjacent element grids have been sliced in opposite directions. As far as message length is concerned, the worst case is when two adjacent element grids have been sliced in the same direction, leading to communication along the whole side of the element grid.

As a best case, consider the situation when all element grids have equally sized rectangular partitions (i.e., the grid is partitioned across both directions). Here, for each partition, there will be at most two messages to send in the external communication. The message length will be (approximately) $\sqrt{N}/\sqrt{p_c^{(i)}}$, assuming almost square grids. (Here, and below, superindex (i) corresponds to Strategy i, $i = 1, 2$.)

Proposition 5 *With assumptions as above, we get the following relations between communication overheads for Strategy 1 and Strategy 2, with respect to external communication:*

In the worst case:

$$R_s^{(1)}/R_s^{(2)} \quad = \quad \mathcal{O}(n^2)$$

$$R_{tr}^{(1)}/R_{tr}^{(2)} = \mathcal{O}(n^2)$$

In the best case:

$$\begin{aligned} R_s^{(1)}/R_s^{(2)} &= \mathcal{O}(n) \\ R_{tr}^{(1)}/R_{tr}^{(2)} &= \mathcal{O}(\sqrt{n}), \end{aligned}$$

Proof Disregarding internal communication implies $t_c(p) = t_e(p)$, and

$$\begin{aligned} R_s^{(i)} &= \mathcal{O}(pn_c^{(i)}e_c^{(i)}) \\ R_{tr}^{(i)} &= \mathcal{O}(pn_c^{(i)}e_c^{(i)}m_c^{(i)}). \end{aligned}$$

We already noted that $n_c^{(1)} = n$, and $n_c^{(2)} = 1$. It remains to estimate $e_c^{(i)}$ and $m_c^{(i)}$.

Worst case. In a really extreme case, the processor holding a certain partition $\mathcal{P}$ may have to send data to all partitions of all element grids. Since, with Strategy 1, each processor holds one partition of every element grid, the processor holding $\mathcal{P}$ would send, in this extreme case, $(p-1)(n-1)$ messages in the external communication. With Strategy 2, the processor holding partition $\mathcal{P}$ would send one message to each processor outside its own cluster, i.e., $p - p_c$ messages (where p_c is the number of processors in the cluster containing $\mathcal{P}$). At worst, $p_c = \mathcal{O}(1)$.

To summarize, in the worst case

$$\begin{aligned} e_c^{(1)} &= \mathcal{O}(pn) \\ e_c^{(2)} &= \mathcal{O}(p) \end{aligned}$$

Remember that for message length, the worst case is when two adjacent element grids have slices in the same direction, leading to communication along the whole side of the element grid. The message length will then be

$$m_c^{(1)} = m_c^{(2)} = \sqrt{N}.$$

Here, we have assumed a non-naive partitioning, so that slices are always across the longest side of the grid. Then, the length of the slice interface will be at most $\sqrt{N}$.

Consequently, in the worst case

$$\begin{aligned} R_s^{(1)}/R_s^{(2)} &= \mathcal{O}((pn)^2)/\mathcal{O}(p^2) = \mathcal{O}(n^2) \\ R_{tr}^{(1)}/R_{tr}^{(2)} &= \mathcal{O}(n^2). \end{aligned}$$

Best case. Here, for each partition, there will be at most two messages to send in the external communication. Thus,

$$\begin{aligned} e_c^{(1)} &= \mathcal{O}(1) \\ e_c^{(2)} &= \mathcal{O}(1) \end{aligned}$$

The message length will be (approximately) $\sqrt{N}/\sqrt{p_c^{(i)}}$, assuming almost square grids. Consequently, in the best case

$$\begin{aligned} m_c^{(1)} &= \mathcal{O}(1/\sqrt{p}) \\ m_c^{(2)} &= \mathcal{O}(\sqrt{n}/\sqrt{p}) \end{aligned}$$

It follows that

$$\begin{aligned} R_s^{(1)}/R_s^{(2)} &= \mathcal{O}(pn)/\mathcal{O}(p) = \mathcal{O}(n) \\ R_{tr}^{(1)}/R_{tr}^{(2)} &= \mathcal{O}(n)\mathcal{O}(1/\sqrt{p})/\mathcal{O}(\sqrt{n}/\sqrt{p}) = \mathcal{O}(\sqrt{n}). \end{aligned}$$

This completes the proof. □

Remark: The proposition does not take into account the fact that external communication *between* clusters (as in Strategy 2) is likely to be more efficient than external communication *within* a cluster (Strategy 1). In fact, the model for communication time, that was used in the proof, is too generous towards Strategy 1, because it does not take into account the risk of link contention. Link contention could appear with Strategy 2 as well, but with a smaller probability than for Strategy 1.

Consequently, our result says that even if no link contention would occur, the external communication would be much faster with Strategy 2 than with Strategy 1.

2.3 Case study

The results of the previous section show that there is a significant difference between the two strategies, with respect to external communication overhead. However, it remains to see whether this difference is of *importance* in the sense that it corresponds to an inacceptable loss of efficiency.

To investigate this in more detail we make a case study. Consider the best case, as discussed above. Here, $R_s^{(1)} = nR_s^{(2)}$. Moreover,

$$R_s^{(2)} = 2p\alpha/t(1).$$

Next, we estimate $t(1)$, which is the execution time (for making one time step with the numerical method) if one processor is used. With n blocks, N points per block, and ω floating point operations per point (per time step), we get

$$t(1) = nN\omega t_f,$$

where t_f is the average execution time for a floating point operation. Assuming in addition that p/n is an integer, m, we get

$$R_s^{(2)} = \frac{2}{N\omega}\frac{\alpha}{t_f}m.$$

Finally, assume that the block has 100×100 points, and that ω is of the order 10^2 (which is realistic for a CFD computation). Then,

$$R_s^{(2)} = 2\frac{\alpha}{t_f} m 10^{-6}.$$

The quotient α/t_f depends on the computer platform. We consider two typical cases: Cray T3D (small communication overhead), and a cluster of workstations (large communication overhead).

Case 1: Cray T3D $\alpha = 1\mu s$, $t_f = 14ns$.

$$R_s^{(2)} = 1.4m \times 10^{-4}$$

The efficiency for Strategy i, with respect to external communication startup, i.e., assuming that no other overhead was at hand, would be $1/(1 + R_s^{(i)})$. The table below shows the results for Strategy 2, for different values of m. We see that in all cases, the efficiency is over 98%. The effect of external communication is negligible.

m	$E = 1/(1 + R_s^{(2)})$
1	0.9999
10	0.9986
100	0.9862

Table 2: Efficiency, E, for Strategy 2, with respect to external communication, on a computer with small communication latency. The number of processors per cluster is m.

For Strategy 1, $R_s^{(1)} = nR_s^{(2)} = 1.4nm \times 10^{-4} = 1.4p \times 10^{-4}$. The table below shows the efficiency for Strategy 1.

p	$E = 1/(1 + R_s^{(1)})$
100	0.9862
1000	0.8772

Table 3: Efficiency, E, for Strategy 1, with respect to external communication, on a computer with small communication latency. The number of processors is p.

Thus, the reduction in efficiency is small also for Strategy 1, but it is not negligible, as was the case for Strategy 2.

Case 2: Cluster of workstations $\alpha = 100\mu s$, $t_f = 14ns$.

$$R_s^{(2)} = 1.4m \times 10^{-2}$$

The results for Strategy 2, for different values of m, are shown below.

m	$E = 1/(1 + R_s^{(2)})$
1	0.9862
10	0.8772
100	0.4167

Table 4: Efficiency, E, for Strategy 2, with respect to external communication, on a computer with large communication latency. The number of processors per cluster is m.

This points to a drastic drop in efficiency for Strategy 2 as the cluster size increases.

The situation is even worse for Strategy 1.

p	$E = 1/(1 + R_s^{(1)})$
100	0.4167
1000	0.0667

Table 5: Efficiency, E, for Strategy 1, with respect to external communication, on a computer with large communication latency. The number of processors is p.

2.4 Conclusions for non-hierarchical composite grids

To summarize, the comparison between Strategy 1 and Strategy 2 for non-hierarchical composite grids has given the following results.

- There are differences between the two strategies in terms of arithmetic load balance. However, these differences will be accommodated by special treatment within the framework of each strategy.

  That is, it is necessary to implement *both* algorithms so as to allow for small grids to be partitioned over less than p processors. Proposition 2 gives a guideline for the choice of number of processors, in the case of slice and rectangle partitions. Moreover, Strategy 2 must be implemented so that a cluster can have a non-integer number of processors.

- The communication overhead is significantly larger for Strategy 1 than for Strategy 2. This is true for both internal and external communication. The case study concerning the overhead for external communication indicates that Strategy 1 is not recommendable on a computer with many processors and large communication latency.

3 Hierarchical composite grids

3.1 Four interpretations of Strategy 2

In the context of hierarchical grids, Strategy 2 can be interpreted in several ways. For the situation we consider, $p > n$, i.e., when there are more processors than element grids, we discuss four alternatives, all adhering to the general pattern of Strategy 2. They are referred to as Strategy 2.1 until Strategy 2.4.

For the presentation of these, some notation is needed. Let G be a two-dimensional composite grid. (The generalization to three dimensions is evident). In general, G is hierarchical, with L levels of element grids. On level 1, we have a number of grids laid out to cover the geometry of the computational problem. Levels 2 until L consist of superimposed element grids, for resolving sharp gradients. Let $g_i^{(j)}$ denote element grid i on level j. This grid, and all its superimposed element grids, form a local composite grid, $G_i^{(j)}$.

Remark 1: The notation above indicates that each element grid on level $j+1$ has a unique local main grid on level j. This is not the case in general. However, for each element grid on level $j+1$, we can assign *one* of its underlying grids on level j to be the local main grid. Then, the notation applies.

Remark 2: Some grid generation systems cut holes in the underlying grids. The motivation is that the superimposed grid is more important, so it is a waste of computing time to compute on the covered parts of the underlying grid. The grid being structured, the hole will be stair-shaped. We will assume that the grid is logically rectangular, which is not true for stair-shaped grids. However, partitioning of a stair-shaped grid can be performed *as if* the grid was rectangular, with work load 0 in the cut-out points. (Alternatively, the stairshaped grid could be decomposed, before partitioning, into a number of—almost—rectangular pieces.)

All four strategies begin by assigning a number of processors to each local composite grid on level 1. Let this initial step be represented by a function $P(G,p)$, which returns an array c, where c_i is the number of processors in the cluster for $G_i^{(1)}$. Then, the question is how to partition $G_i^{(1)}$ over its cluster of processors. In the following, the different strategies are informally explained.

Strategy 2.1 (Domain partitioning)

> Each element grid $g_i^{(1)}$ on level 1 covers a separate subdomain. In Strategy 2.1 the *subdomains* are partitioned. If a subdomain is partitioned into slices, for example, the result will be a partion which consists of one slice of the corresponding $g_i^{(1)}$, and in addition of irregular pieces of superimposed grids on other levels of $G_i^{(1)}$.
>
> Strategy 2.1 can be regarded as a straightforward generalization to the hierarchical from the non-hierarchical case. G is partitioned *as if* it was non-hierarchical, containing only the level 1 grids. However, the *work factors* of the superimposed grids in $G_i^{(1)}$ are taken into account when the partitions for $g_i^{(1)}$ are computed.

Strategy 2.2 (Hybrid strategy)

> Here, after execution of the initial step, each local composite grid $G_i^{(1)}$ (on level 1) is distributed over its cluster of c_i processors, according to Strategy 1. Thus, Strategy 2.2 can be regarded as a hybrid between Strategy 2 and Strategy 1.

Strategy 2.3 (Subclustering, overlapping clusters)

> For each local composite grid $G_i^{(1)}$, divide $g_i^{(1)}$ into c_i partitions, and assign one partition to each processor in the corresponding cluster. Each *superimposed* grid in $G_i^{(1)}$ is an element grid $g_k^{(2)}$ on level 2. Divide the c_i processors into subclusters, one for each local composite grid $G_k^{(2)}$.
>
> Repeat this procedure until all levels of G have been partitioned.
>
> Strategy 2.3 yields overlapping clusters, in the sense that the final clusters contain partitions of several element grids, on different levels.

Strategy 2.4 (Subclustering, disjoint clusters)

> For each local composite grid $G_i^{(1)}$, divide the c_i processors into subclusters, one for $g_i^{(1)}$, and one for each local composite grid $G_k^{(2)}$.
>
> Repeat this procedure until each element grid has been assigned a unique cluster. Then, partition the element grids over their respective cluster.

Remark 3: Mapping is not discussed above. In order to create optimal partitions, it is necessary to take into account the communication pattern, which depends on the mapping. The simplest alternative is to partition without knowing the exact communication pattern (but still taking communication into account, in less detail). Then, as a subsequent step, mapping is performed, so as to minimize communication, given the set of partitions. A more ambitious alternative would be to iterate between partitioning and mapping.

Remark 4: Note that the previous discussion is not restricted to structured grids. In principle, it is possible to have unstructured element grids.

3.2 An asset of the four algorithms

The previous comparisons with Strategy 1—for the non-hierarchical case—has shown the general merits and deficiencies of Strategy 2. However, for hierarchical grids, the different interpretations of Strategy 2 emphasize these general properties more or less strongly.

From the point of view of arithmetic overhead due to load imbalance *between* clusters (cf. §2.1.2), further subclustering means additional round-off errors, which increases the potential imbalance. In particular, it will be necessary to include special treatment to avoid subclustering of very small clusters.

The conclusion is that the strategies with subclustering, Strategy 2.3 and Strategy 2.4, are the most problematic ones in terms of arithmetic overhead. Strategy 2.4 is the worst alternative in this respect, as it generates the smallest clusters.

As for communication overhead, the picture is different. This overhead increases with the number of grids per cluster. Strategy 2.4, with only one grid in each cluster, is the best from this point of view, and Strategy 2.2 is the worst alternative.

For the subclustering algorithms, the external communication will depend on the mapping of processors to clusters. If this mapping can be done so that neighbour relations in the processor graph are preserved, then Strategy 2.3 and Strategy 2.4 will have a clear advantage in terms of communication.

Strategy 2.1 is special from the aspect of communication, because it disregards the hierarchical structure of the composite grid. Thus, there will be only one partition per processor, but this partition can contain pieces of many grids, from different levels of the hierarchy. This is excellent as far as external communication is concerned, because most of the interpolation between grids on different levels can be done without any communication at all. The communication between grids on the same level will be of the same complexity as for Strategy 2 in general.

However, Strategy 2.1 for hierarchical grids will be rather complicated to implement. Moreover, it is difficult to apply when there are superimposed grids going across the boundaries of underlying grids.

To summarize, each algorithm has its shortcomings. From the crucial aspect of communication overhead, the strategies based on subclustering have an advantage. This presumes a good mapping of processors to clusters. From the aspect of arithmetic overhead, subclustering is problematic. With special treatment to avoid nasty situations, the problems can be alleviated. However, this makes the implementation of the subclustering algorithms tricky, and involves an overhead of various checks during run-time.

4 Dynamic grids

At the outset, we claimed that the main advantage of Strategy 1 is its ability to treat dynamic grids. Here, by a dynamic grid we mean a hierarchical composite grid where superimposed element grids are added or deleted as the computations evolve. When Strategy 1 is used, each new grid will be partitioned over all processors, and this will have only marginal effects on the arithmetic overhead. There will never be need for redistribution of the grid.

If we consider only this aspect, Strategy 1 is clearly better than Strategy 2. However, our discussions in the previous sections have shown that Strategy 2 can give considerable advantages for static grids, mainly in terms of communication overhead. Thus, it is meaningful to investigate in more detail the ability of Strategy 2 to treat dynamic grids.

For all versions of Strategy 2, the dynamic evolvement of the grid structure will disturb the load balance *between* clusters. There is a need for a mechanism for deciding when a major redistribution is necessary. Moreover, each strategy must be supplied with a redistribution algorithm. However, in the following we discuss only how to handle a changing grid structure, *without* making a complete redistribution.

Strategy 2.1 is the most problematic alternative. As it disregards the hierarchical structure of the grid, it has no ability at all to handle the addition or removal of a superimposed grid. For this reason, the strategy will be omitted from further discussion.

Strategy 2.2 is a hybrid between Strategy 1 and Strategy 2. Within a cluster, an element grid can be added or removed without significant effects on the load balance. However, this approach will disturb the load balance *between* clusters. The effect can be severe, if the PDE solution has some dramatic behaviour in one subdomain (cluster) but is smooth in most subdomains. The same holds for Strategy 2.3 and Strategy 2.4.

Strategy 2.3 leads to a complicated structure of overlapping clusters. The final clusters (the leafs in the cluster tree), contain L partitions. One of them comes from a grid on level L (the highest superimposed level), another from this grid's local main grid on level $L-1$, and so on. Now, an element grid can be added on any grid level $l > 1$. Using the notation introduced in § 3.1, assume that $g_i^{(l-1)}$ is the element grid on which the new grid is superimposed. Then, the entire local composite grid $G_i^{(l-1)}$ will have to be repartitioned. This implies that the cluster for $G_i^{(l-1)}$ must be substructured in a new way, and that the grids in $G_i^{(l-1)}$ will have to be redistributed over the new subclusters. This having been done, the load balance within the cluster will be essentially unchanged. However, there will be an imbalance *between* clusters.

The removal of an element grid, within the framework of Strategy 2.3, is more straightforward. We can simply remove the grid from the processors it has been mapped to. The resulting distribution will then not conform strictly to Strategy 2.3, but this has to be accepted. Removing a grid will not significantly disturb the load balance within the grid's cluster. Thus, making a repartitioning would be unmotivated. Clearly, the removal of an element grid would disturbe the load balance *between* clusters. However, this situation would not be improved by repartitioning *within* a cluster.

Strategy 2.4 leads to final clusters with only one grid per cluster, and one partition per processor. The addition of an element grid could be done in the same way as for Strategy 2.3. (An alternative would be to put the new grid in the cluster of its local main grid $g_i^{(l-1)}$, splitting this cluster into two. This would be straightforward, but is likely to degrade the load balance severely.)

Within the framwork of Strategy 2.4, when an element grid is removed, the processors in the corresponding cluster will become idle. This calls for repartitioning.

The previous discussion speaks in favour of Strategy 2.2. The strategies using subclustering will distribute each element grid over fewer processors, which means that the addition or removal of a grid will have a larger effect on the arithmetic load balance. This, in turn, will lead to more frequent redistributions. Moreover, for an orthodox application of Strategy 2.3 or Strategy 2.4, it is necessary to make a local redistribution each time a new grid is added.

5 Conclusions and continuation

We have compared two general strategies for the partitioning of composite grids. The purpose has been to make a nuanced evaluation, pointing at the advantages and drawbacks of each strategy.

Strategy 1 is straightforward to implement, and in situations where it behaves well, it is certainly to prefer. This is further stressed for dynamic grid structures. However, for massively parallel processors, or in general when there are many element grids which are small compared to the number of processors, Strategy 1 is less appropriate. In such situations, Strategy 2 is better, but has to be implemented with care in order to behave well.

Our investigation points at crucial implementation issues:

- Both strategies need special mechanisms for element grids that are too small relative to the number of available processors. Such a mechanism should include results like those in Proposition 2, for deciding *if* a grid needs special treatment. In addition, there is need for an algorithm to handle the actual distribution in these special cases.

- Strategy 2 needs a special mechanism to avoid the formation of too small processor clusters. The problem is similar, but not identical to the previous one. Results like that of Proposition 3 are needed in order to decide when special treatment is called for.

- In principle, Strategy 2 is superior to Strategy 1 with regard to communication. However, for this to be true in practice, the mapping problem has to be solved. This is more complicated for Strategy 2 than for Strategy 1.

- For dynamic grid structures, Strategy 2 is problematic. To a certain extent, the addition or removal of element grids can be handled via local redistribution *within* processor clusters. However, in any case Strategy 2 has to be equipped with a redistribution algorithm, and a mechanism for deciding when a complete redistribution becomes necessary.

For hierarchical composite grids, Strategy 2 can be interpreted in several ways. We have discussed four alternatives. Of those, Strategy 2.2 stands out as the most promising for further study. It can be regarded as a hybrid between Strategy 1 and Strategy 2, thus balancing the advantages and drawbacks of both strategies in an attractive way.

Strategy 1 could in fact be regarded as a special case of Strategy 2. Consequently, Strategy 2 forms a general framework for the partitioning of composite grids. Given a certain composite-grid problem and a specific computer platform, the problem is to make the appropriate choices within this framework. The results presented here can serve as a basis for the automatization of this decision process.

References

[1] M.J. Berger, J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comp. Phys., 53 (1984), 484-512.

[2] M.J. Berger, S. Bokhari, *A partitioning strategy for non-uniform problems on multiprocessors*, IEEE Trans. Computers, V-85(5):570–580, 1987.

[3] G. Chesshire, W. D. Henshaw, *Composite meshes for the solution of partial differential equations*, J. Comp. Phys., 90 (1990), pp. 1-64.

[4] J. de Keyser, D. Roose, *Load balancing data parallel programs on distributed memory computers.* Parallel Computing, 19 (1993), pp. 1199-1219.

[5] D.J. Evans, W.U.N. Butt, *Load balancing with network partitioning using host groups*, Parallel Computing, 20 (1994), pp. 325–345.

[6] J. Häuser et al., *Parallel computing in aerospace using multi-block grids. Part 1: Applications to grid generation*, Concurrency, 4 (1992), pp. 357-376.

[7] High Performance Fortran Forum, *High Performance Fortran Language Specification.* Rice University, Houston, 1993.

[8] M. A. Iqbal, et al., *Performance tradeoffs in static and dynamic load balancing strategies.* Technical Report 86-13, ICASE, NASA Langley Research Center, Hampton, VA, 1986.

[9] K.-H. Kao, M.-.S. Liou, *Direct replacement of arbitrary grid-overlapping by non-structured grid*, ICOMP-94-07, NASA Lewis Research Center, Cleveland, Ohio, 1994.

[10] A. Pothen, et al., *Partitioning sparse matrices with eigenvectors of graphs.* SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430-452.

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

[12] S. L. Smith, *Adaptive asynchronous parallel algorithms in distributed computations.* Ph.D. thesis, Graduate School of the University of Colorado, 1991.

[13] K. Solchenbach, *Grid applications on distributed memory architectures: Implementation and evaluation*, Parallel Computing,7 (1988), pp. 341-356, North-Holland.

[14] J.L. Steger, J.A. Benek, *On the use of composite grid schemes in computational aerodynamics*, Comp. Meth. Appl. Math. Eng., 64 (1987), pp. 301-320.

[15] J.F. Thompson, *Composite grid generation code for general 3-D regions — the EAGLE code*, AIAA Journal, 26 (1988), pp. 271-272.

[16] M. Thuné, *A partitioning strategy for explicit difference methods*, Parallel Computing, 15 (1990), pp. 147-154.

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

[18] M. Thuné, *A partitioning algorithm for composite grids*, Parallel Algorithms and Applications, 1 (1993), pp. 69-81.

[19] C. Walshaw, M. Berzins, *Dynamic load balancing for PDE solvers on adaptive unstructured meshes.* Report 92.32, School of Computer Studies, University of Leeds, Leeds, 1992.

[20] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations.* Concurrency, 3 (1991), pp. 457-481.