Computing (2007) 81: 53–75
DOI 10.1007/s00607-007-0243-1
Printed in The Netherlands

Computing

Adaptive full domain covering meshes for parallel finite element computations

S. Vey[1] and **A. Voigt**[2]

[1]Crystal Growth Group, Research Center Caesar, Ludwig-Erhard-Allee 2, Germany
[2]Institut für Wissenschaftliches Rechnen, TU Dresden, Dresden, Germany

Received 4 May 2006; Accepted 15 August 2007; Published online 15 October 2007

Summary

In this work, we present a new parallelization concept for adaptive finite element methods. Compared to classical domain decomposition approaches, the concept of adaptive full domain covering meshes reduces the parallel communication overhead. Furthermore, it provides an easy way to transform sequential codes into parallel software by changing only a few lines of source code.

AMS Subject Classifications: 65Y05.

Keywords: parallel computing; finite element method; adaptivity; partition of unity.

1. Introduction

In traditional parallelization concepts each process computes within a partition Ω_i of the full problem domain Ω. At partition boundaries a copy of needed neighboring data (shadow data) is stored to reduce the communication between the processes. But communication must be done anytime when the neighboring data changes. So, the parallel structure must be considered in many parts of the code. Therefore, parallelizing a sequential code is a complex matter if not only the solver is parallelized but the whole problem including setup, grid generation, assembly, solving, error estimation and visualization. Using adaptive full domain covering meshes can circumvent this complexity. Here, each process computes a solution on the whole domain Ω. But outside of the local partition Ω_i a relatively coarse mesh is used. At the end of computation the different processes combine their solutions into one global solution by a partition of unity method.

Bank and Holst presented a similar technique in [1]. Their approach consists of the following steps: First, the problem is solved on a relatively coarse mesh and

Correspondence: Simon Vey, Crystal Growth Group, Research Center Caesar, Ludwig-Erhard-Allee 2, 53175 Bonn, Germany (E-mail: simon.vey@caesar.de)

local error estimates are computed. Next, partitions with approximately the same error are created, assuming that equal errors will lead to approximately equal future work. Then each process computes on the whole domain, but refinements are limited largely to its own partition. Finally, after parallel computations, a global solution is constructed on the union of the refined partitions. This can be done e.g., by a parallel multigrid solver or by a partition of unity method.

In contrast to our approach, no repartitioning is provided there, but only one partitioning based on local error estimates at the beginning. This makes it hard to obtain a good load balance, because local error estimates are only a rough estimate for the future work to be done in some region. Furthermore, repartitioning becomes useful in time dependent problems to consider instationary solution properties.

Mitchell introduced a concept called full domain partitions in [2]. It combines the adaptive finite element method with a parallel solver and a load balancing step in each iteration. Outside of the local partition the coarsest possible compatible mesh is used for each process. The global solution is built directly by a specialized parallel multigrid solver. So, no partition of unity is needed. A load balanced repartitioning is done in every iteration.

The concept of adaptive full domain covering meshes presented in this paper combines the benefits of both approaches. No specialized solvers or other specialized modules are needed. Our repartitioning strategy allows one to handle time dependent problems in a straightforward way. Load balance quality and repartitioning overhead can be balanced against each other. Furthermore, the concept of Mesh Structure Codes (Sect. 4.1) presents a very efficient way to exchange mesh information between the processes and further reduces the need for communication.

We introduce our approach in the context of adaptive multidimensional simulations (AMDiS), a finite element toolbox for the solution of systems of partial differential equations (PDEs) that is written in C++. Problem formulations can be done on a high level of abstraction in a dimension independent way. Numerical issues are kept away from the user as far as possible. More information about AMDiS can be obtained from [3].

The remainder of this paper is organized as follows: Section 2 shortly describes adaptivity and the adaptive mesh structure used in AMDiS. This is the context, in which the approach is implemented. Section 3 gives a survey of the concept followed by implementation aspects presented in Sect. 4. In Sect. 5, we give a short source code example that should demonstrate the simplicity of parallelizing a sequential program. Afterwards we present some numerical results in Sect. 6. We close this paper by drawing conclusions in Sect. 7.

2. Adaptive meshes

In AMDiS, the mesh adapts to solution properties within the adaptation loop. So, a given error tolerance can be reached with a minimal amount of unknowns. One iteration of an adaptation loop consists of four steps. In the first step (*assemble step*) a linear equation system is built with one equation for each unknown of the actual discretization. In the second step (*solve step*) this system is solved by an iterative

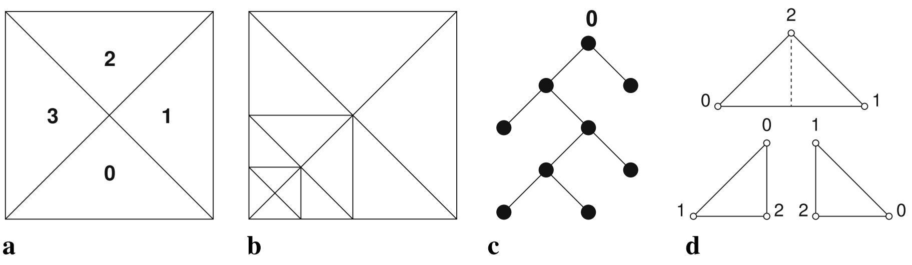

Fig. 1a. A two-dimensional macro mesh. (**b**) Some refinements of it. (**c**) Binary tree of macro element 0. (**d**) A bisected triangle with local node numeration of parent and children

solver. The result is an approximate solution of the PDE. Non-adaptive methods stop here. In adaptive codes the quality of the current approximation is rated by estimating the global error and local element-wise error indicators (*estimate step*). If the global error estimation exceeds a given error tolerance, those elements with a high local error indicator will be refined (*adapt step*). In regions with a very low error indicator the mesh may be coarsened, which is important for time dependent problems. With this adapted mesh a new iteration is started. If the global error estimation fulfills the tolerance criterion, the adaptation loop stops and the adaptive computation is finished. Otherwise, a new iteration will be started.

AMDiS meshes consist of simplicial elements which are lines in 1D, triangles in 2D and tetrahedra in 3D. If an element has to be refined during the adaptation loop, it will be bisected into two elements of the same dimension. The refinement algorithm is described in [4] in more detail. The two new elements are called children of the original parent element. For each element of the coarsest mesh (*macro mesh*) a binary tree arises during adaptation. To avoid hanging nodes (vertices of an element which are not vertices of the neighbor element), it may be necessary to first refine a neighbor of an element before refining the element itself. So, a single refinement can cause some propagation refinements of elements in the neighborhood. In Fig. 1 a triangular macro mesh consisting of four elements (a), some refinements of this macro triangulation (b), and the corresponding binary tree for macro element 0 (c) are shown. Figure 1d illustrates the bisection of a triangle and the element-wise node numeration of parent and children in the case of linear basis functions with one node at each element vertex.

3. Overview

The main idea of the parallelization approach presented in this work is the concept of adaptive full domain covering meshes. Starting with a very coarse macro mesh, a partitioning mesh is obtained by some global or adaptive refinement steps. The leaf elements of the resulting mesh build the partitioning level which is the basis for all future domain decompositions. After partitioning the mesh, each process computes on the whole domain, but refinements are allowed only on the local partition including a certain overlap region and some necessary propagation refinements. The overlap is needed to construct a global solution after the adaptation loop. The

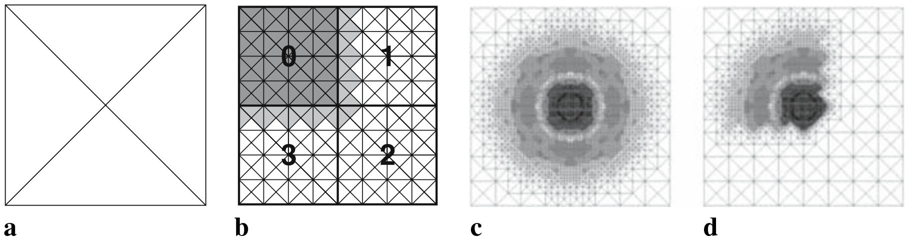

Fig. 2a. A triangular macro mesh, (**b**) domain decomposition after six global refinements and overlap for partition 0, (**c**) composite mesh after adaptation loop, (**d**) local mesh of process 0 after adaptation loop

propagation refinements are needed to preserve mesh compatibility. Figure 2 illustrates the concept of adaptive full domain covering meshes. To simplify matters, in this example the partitioning did not change during the adaptation loop. In Sect. 4.4, we show how repartitionings within the adaptation loop are handled.

Due to adaptive refinements, the load may get out of balance during the adaptation loop. Then a new load balanced repartitioning will be necessary. The load balancing can be done before or after adapting the local meshes. If it is done before the adaptation step (predictive load balancing), the load situation after the adaptation has to be estimated considering the local error estimates at the elements. In this work we applied regular load balancing in which the new partitioning is computed after mesh refinements. This leads to a little more communication, because a finer mesh has to be redistributed after repartitioning, but on the other hand the load balance is more accurate. After the adaptation loop the global solution is constructed by a parallel partition of unity of all local solutions. Algorithm 1 describes the parallel adaptation loop on a high abstraction level. In the initialization step of line 1 the partitioning level is constructed and the first domain decomposition is done.

Algorithm 1 parallel adaptation loop

```
 1: initialize parallelization
 2: assemble, solve, estimate
 3: while tolerance not reached do
 4:    adapt mesh
 5:    if load out of balance then
 6:       repartition mesh
 7:    end if
 8:    assemble, solve, estimate
 9: end while
10: build global solution
```

Initialization, repartitioning and building the global solution are described in more detail later in this paper. Without these three steps the algorithm would describe the usual non parallel adaptation loop for stationary problems in AMDiS, which is described in [3].

In Fig. 3, the domain notation used in this paper is shown. Ω stands for the whole problem domain and Ω_i for the local partition of process i. The local partition

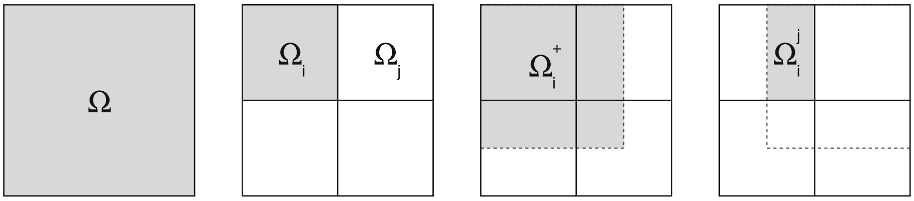

Fig. 3. Domain notation used in this paper

of process i including overlap is denoted by Ω_i^+. Ω_i^j is the sub domain of Ω_i that belongs to the overlap of Ω_j ($\Omega_i^j := \Omega_i \cap \Omega_j^+$).

4. Implementation

The parallelization was realized on top of the *message passing interface* (MPI), which is the standard for message passing in distributed memory environments.

In Sect. 4.1, the concept of Mesh Structure Codes is introduced which is an efficient way to exchange mesh information between the processes. Mesh Structure Codes can be used to construct global element and node indices, addressed in Sect. 4.2. Section 4.3 describes the *three level approach* which allows one to decouple partitioning level, overlap level and global coarse grid level. The repartitioning algorithm is the topic of Sect. 4.4. Section 4.5 addresses the construction of a global solution by a parallel partition of unity method. Finally, in Sect. 4.6 the parallelization of time dependent problems and systems of PDEs is discussed.

4.1 Mesh structure codes

At some points in the computation, e.g., when the global solution should be built or a repartitioning is performed, the current mesh states have to be communicated between the parallel processes. Therefore, a compact mesh representation is needed, which can be used for MPI communication. The concept of Mesh Structure Codes provides such a compact representation.

Like mentioned in Sect. 1 all processes start with the same coarse macro triangulation, which is refined in different ways during parallel computation. The only allowed refinement operation is bisection of elements. Every element has either two children or no children. The same holds for the nodes of the corresponding binary tree. For each node we store a 1 if the corresponding element is refined and a 0 otherwise. To obtain a unique node order, we perform a pre-order traversal on the tree: First, the root of the tree is visited, then the left sub tree and, finally, the right sub tree is traversed in pre-order recursively. The resulting binary sequence can be interpreted as an (unsigned long) integer value which can be sent over MPI very efficiently. If the number of mesh elements exceeds the capacity of one integer variable, an array of integers has to be used. Using this code the receiver can easily reconstruct the senders mesh or fit the local mesh to it. Figure 4 illustrates how to construct the Mesh Structure Code of an adaptively refined triangle. The corresponding binary

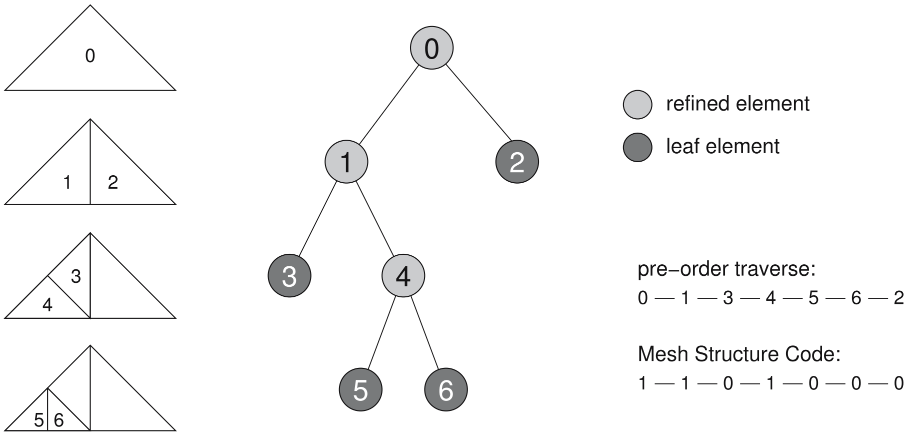

Fig. 4. Mesh Structure Code of an adaptively refined triangle

code 1101000 is represented by the integer value 104 together with the position of the first relevant bit within the binary representation. In this way, the code 1101000 can be distinguished from the code 0...01101000. The Mesh Structure Code of a mesh is the concatenation of the Mesh Structure Codes of its macro elements. The Mesh Structure Code of the mesh showed in Fig. 1b is 110110000 – 0 – 0 – 101101000 (the "-" character is used for a higher readability to seperate the macro elements). Since the construction of Mesh Structure Codes is not expensive, no update procedure for them is needed. A Mesh Structure Code is constructed for the current mesh based on the hierarchical mesh data structure described in Sect. 2. It will be deleted after its usage.

Another feature of Mesh Structure Codes is the possibility to build the composition of multiple meshes on a binary level. Therefore, in Algorithm 2 we firstly define a recursive algorithm to extract a sub code $subTreeCode$ which represents the sub tree starting from the node corresponding to position pos in the Mesh Structure Code $structureCode$. If $subTreeCode$ is not empty at the beginning, the result is pushed to the back of $subTreeCode$. The return value of the algorithm is the first position within $structureCode$ after the extracted sub tree. The first recursive call of **getSubTreeCode** delivers the first child's (or left) sub tree code, the second call the second child's (or right) sub tree code. If $subTreeCode$ is not given ($subTreeCode == NULL$), the algorithm just skips the corresponding sub tree and returns the first position after it. This feature will be used, e.g., in Algorithm 4 to skip unused sub trees in the Mesh Structure Code. In Algorithm 3 **getSubTree-Code** is now used to merge two codes into a composite Mesh Structure Code. The result of this procedure is a code which represents the composition of the two meshes represented by $structureCode1$ and $structureCode2$.

To synchronize the local meshes during the parallel computation, each process computes its local Mesh Structure Code. After that the codes are exchanged between the processes via MPI. Now each process knows the Mesh Structure Code of each other process and can build the composite code by merging all local codes. Then the local mesh can be locally adapted to fit to the composite code in some region.

Algorithm 2 getSubTreeCode($structureCode$, pos, $subTreeCode$)

```
1: if $structureCode[pos] == 0$ then
2:   $pos = pos + 1$
3:   if $subTreeCode \neq NULL$ then
4:     push 0 to back of $subTreeCode$
5:   end if
6: else
7:   $pos = pos + 1$
8:   if $subTreeCode \neq NULL$ then
9:     push 1 to back of $subTreeCode$
10:  end if
11:  $pos$ = getSubTreeCode($structureCode$, $pos$, $subTreeCode$)
12:  $pos$ = getSubTreeCode($structureCode$, $pos$, $subTreeCode$)
13: end if
14: return $pos$
```

Algorithm 3 merge($structureCode1$, $structureCode2$)

```
1: $result$ = empty structure code
2: $size1$ = binary length of $structureCode1$
3: $size2$ = binary length of $structureCode2$
4: $pos1 = 0$, $pos2 = 0$
5: while $(pos1 < size1)$ and $(pos2 < size2)$ do
6:   if $structureCode1[pos1] == structureCode2[pos2]$ then
7:     push $structureCode1[pos1]$ to back of $result$
8:     $pos1 = pos1 + 1$
9:     $pos2 = pos2 + 1$
10:  else
11:    if $structureCode1[pos1] == 0$ then
12:      $pos1 = pos1 + 1$
13:      $pos2$ = getSubTreeCode($structureCode2$, $pos2$, $result$)
14:    else
15:      $pos1$ = getSubTreeCode($structureCode1$, $pos1$, $result$)
16:      $pos2 = pos2 + 1$
17:    end if
18:  end if
19: end while
20: return result
```

Usually the local mesh is adapted only within the local partition including the corresponding overlap region. To skip the parts of the code that are not needed for local mesh refinements, Algorithm 2 can be used again. Now, the result in $subTreeCode$ can be ignored and only the new position returned by the algorithm is of interest.

4.2 Global indexing

Different refinement and coarsening orders on the processes can lead to different element and node numerations on the processes, even if the final mesh structure is the same. To create global numerations, which are necessary for inter process communication, the concept of Mesh Structure Codes described in Sect. 4.1 can be used.

Global element indices: First the composite Mesh Structure Code is built on each process by exchanging the local codes and merging them. In the little example

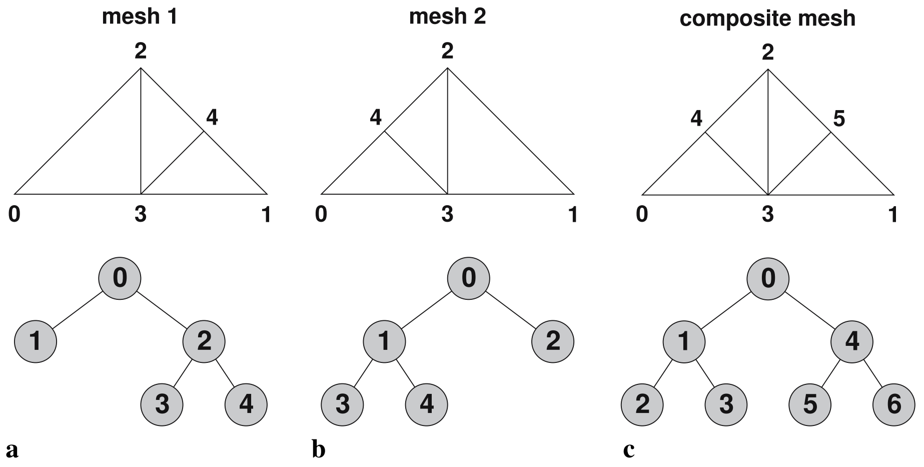

Fig. 5. Node indices and binary trees with element indices for two differently refined meshes (**a, b**) and corresponding global node and element indices for the composite mesh (**c**)

Table 1. Global element indices for composite mesh structure code 1100100 and corresponding element indices for mesh 1 and mesh 2

Composite code	1	1	0	0	1	0	0
Global element index	0	1	2	3	4	5	6
Mesh 1 element index	0	1	–	–	2	3	4
Mesh 2 element index	0	1	3	4	2	–	–

illustrated in Fig. 5 the local codes 10100 and 11000 are merged into the composite Mesh Structure Code 1100100. Now a pre-order traversal is performed on the local mesh of each process. Simultaneously the composite Mesh Structure Code is traversed. If the current element in the local mesh is a leaf element but the structure code entry is 1 (for *refined element*), the corresponding sub tree of the code is skipped using Algorithm 2. The global index of the current local element is always its position in the composite Mesh Structure Code. In Algorithm 4 this procedure is shown. The mapping from local to global element indices in our example can be seen in Table 1.

Algorithm 4 create global element numeration

```
1: Create composite mesh structure code: $code$
2: $pos = 0$
3: **for all** elements $el$ of local mesh (pre-order) **do**
4:    $globalElementIndex(localIndex_{el}) := pos$
5:    **if** $el$ is leaf element **then**
6:       $pos :=$ **getSubTreeCode**($code, pos, NULL$)
7:    **else**
8:       $pos := pos + 1$
9:    **end if**
10: **end for**
```

Table 2. Global node indices in local element order according to global element indices

Global element index	0	1	2	3	4	5	6
Global node indices	0, 1, 2	2, 0, 3	3, 2, 4	0, 3, 4	1, 2, 3	3, 1, 5	2, 3, 5

Table 3. Global and local node indices

Global node index	0	1	2	3	4	5
Mesh 1 node index	0	1	2	3	–	4
Mesh 2 node index	0	1	2	3	4	–

Global node indices: To create a global node numeration the composite Mesh Structure Code is used to create a binary tree, corresponding to the structure of the composite mesh. At each node of this binary tree the list of global node indices will be stored in local element order as a result of the algorithm. In our example for global element 5 the list of global node indices (3, 1, 5) is stored. Once such a binary tree is created, it is easy to obtain a mapping from local to global node indices. To create this binary tree, we can use the local node indices on macro elements as global indices, too, because the numeration at macro level is the same for each process. If the node indices of an element are known, the indices of its children can be constructed depending on the element type. For linear Lagrange elements with one node at each vertex and parent node indices (p_0, p_1, p_2), the indices of the first child are $(p_2, p_0, newIndex)$ and those of the second child are $(p_1, p_2, newIndex)$. The counter $newIndex$ is set to the first number which is not used for the macro mesh nodes. It is incremented by one for each refined element. Now the whole binary tree for every macro element can be constructed and filled with global node indices in a recursive way. Table 2 lists the global node indices for the composite mesh elements for our example. This leads to the mappings illustrated in Table 3.

4.3 Three level approach

As mentioned in Sect. 3, the domain decomposition is done on a fixed partitioning mesh. This partitioning mesh should be defined on a relatively coarse level for two reasons. First, the partitioning process is faster for fewer elements. Second, in time dependent problems it should be possible to coarsen the mesh in regions where such a fine mesh is not longer needed.

On the other hand a large overlap would lead to a bad parallel speedup behavior, because on overlap regions more than one process computes on a fine grid. And defining the overlap on the coarse partitioning level would lead to large overlaps.

A third point is that the quality of the local solution of process i on Ω_i partially depends on the mesh level outside of Ω_i, see, e.g., [5]. So, these three levels should be determined separately. First, the *partitioning level* is created and the domain decomposition is computed. Then the mesh is refined uniformly on the whole domain Ω to create the *global coarse grid*, which now is set to the coarsest mesh for future computations on Ω. In a third step the *local coarse grid* is constructed by uniform refinements within Ω_i, which builds the coarsest mesh for future computations within the partition. To ensure a smaller overlap due to local coarse grid construction, not only

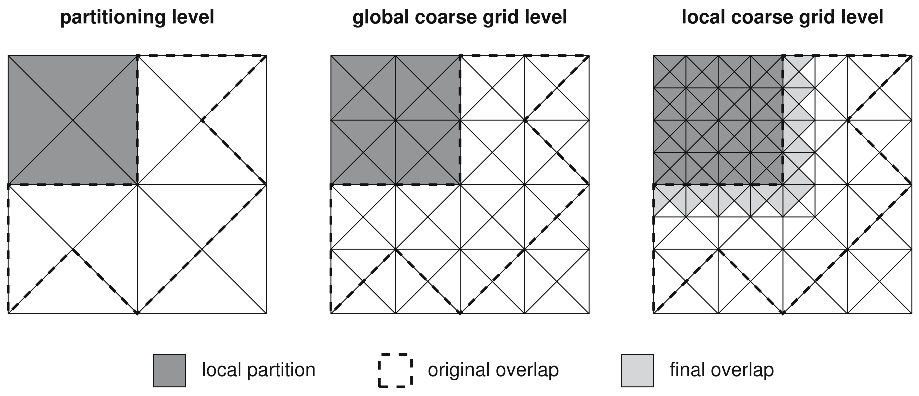

Fig. 6. Example for the three levels of partitioning. The partitioning mesh is globally refined twice to get the global coarse grid level. Then another two global refinement steps applied on Ω_i and its direct neighbor elements (in each step) result in the local coarse grid level

refinements within Ω_i are performed, but in every refinement step all elements with element distance 1 to Ω_i are refined, too. Then on the resulting mesh the overlap computation is performed.

In Fig. 6 an example for this three level approach is shown. Notice the size of the final overlap compared to the size the overlap would have on partitioning level.

In Algorithm 5, we can now take a closer look at the initialization procedure needed for the parallel adaptation loop, introduced in Sect. 3.

Algorithm 5 initialize parallelization (on process i)

```
1: create partitioning level
2: create initial partitioning
3: set element weights on initial $\Omega_i$
4: create new partitioning (in parallel)
5: mark elements of new $\Omega_i$
6: create global coarse grid level
7: create local coarse grid level
8: create overlap
9: create parallel estimator
10: create parallel marker
```

After the creation of the partitioning level, an initial randomized partitioning is done. This first partitioning is needed to compute the first useful domain decomposition in parallel. The element weights set in line 3 are used for the partitioning. Every element of the partitioning mesh is weighted by 1 at this point because we want a partitioning with the same number of elements in each partition. At later stages of the computation, when further refinements of the partitioning mesh exist, the element weights are set to the number of leaf elements belonging to one partitioning element. In Sect. 4.4 setting the element weights for repartitioning is addressed in more detail.

In the last two steps a parallel marker and a parallel estimator are created for each process. The marker is responsible for marking elements for coarsening and refinement depending on local error estimates. The parallel marker is responsible for marking elements for coarsening and refinement depending on local error estimates – with the requirements that only elements within Ω_i^+ can be adapted, and coarsening is limited by the local coarse grid. Propagation refinements are done by the refinement module automatically. So, no markings have to be done for them. The parallel estimator extends an arbitrary sequential estimator by communicating needed global values like estimation sums or maxima after each estimation step.

The parallel marker and estimator are created only once at the beginning of the parallelization. They use information about the current partitioning which is stored at the elements. This information is set in line 2 of Algorithm 5 and adapted within the adaptation loop after each repartitioning (line 6 of Algorithm 1).

4.4 Domain decomposition and repartitioning

For the domain decomposition the parallel graph partitioning library ParMETIS, described in [6], is used. First, a dual graph of the mesh that should be partitioned has to be constructed. Then the nodes of the dual graph, which corresponds to the elements of the mesh, are decomposed considering weight constraints on the graph nodes. The algorithm also tries to minimize the number (or edge weight sum) of graph edges that are cut by domain boundaries. Furthermore, a diffusive repartitioning of adaptively refined meshes is supported.

The goal of the domain decomposition is to decompose the domain Ω into n partitions (n is the number of processes in the parallel computation), such that the work load is approximately the same on every process. Each of the partitions should be connected and the boundaries between the partitions should be minimized to reduce the communication overhead and the size of the resulting overlap. In this work, we use node weights to enable partitioning of a fine mesh on a coarser level (see Sect. 4.3). So far we do not use edge weights for partitioning. The partitioning algorithm in ParMETIS works in parallel. This means that an initial arbitrary partitioning must exist before the first call of ParMETIS. Each process then generates a new partition number for every element of its old partition. Redistribution of the new partitioning information is not part of ParMETIS and has to be done by the calling application. In the context of this work, every process has to collect its new partition elements from all other processes and mark them, including all descendants, as elements of the local partition.

When an element is refined, the partition status (*IN*, *OUT* or *OVERLAP*), is handed down to its children. Coarsening is only allowed if the resulting coarse element has a defined partition status. So, the partitioning mesh is the coarsest possible mesh for all future computations.

During the parallel adaptation each process adapts its local mesh due to local error indicators. In general, this leads to a more and more unbalanced load between the processes. Repartitioning then is applied to recover the optimal load balance. Like mentioned in Sect. 3, the basis of repartitioning is a fixed repartitioning mesh.

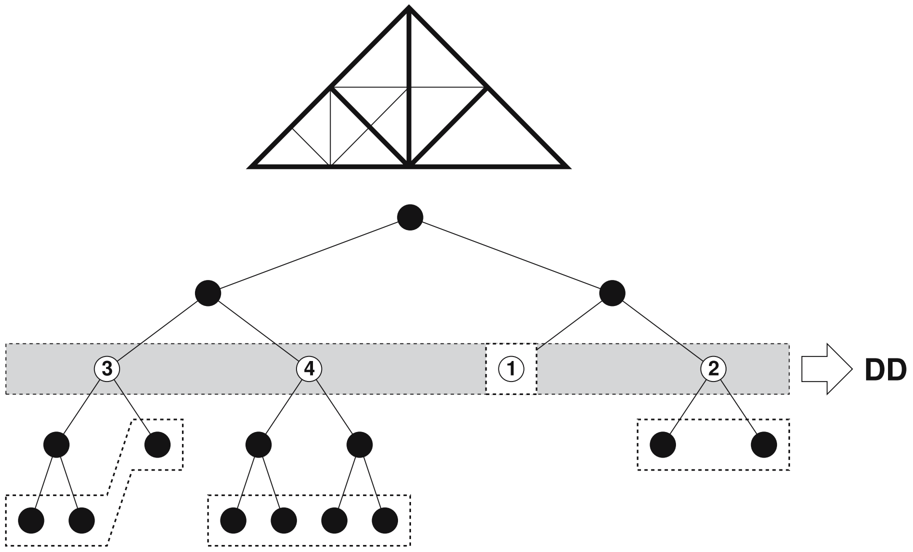

Fig. 7. Element weights for an adaptively refined macro triangle. The partitioning mesh is built by the elements of level two

An optimal load balance is assumed if all partitions have the same number of leaf elements within their local partitions. One could imagine other criteria which could count the number of degrees of freedom or the number of all tree elements of the hierarchical mesh (not only leaf elements). Furthermore, one could consider the work to be done outside of the local partition on each process. To count the number of leaf elements within Ω_i probably is not the most accurate approximation, but it is very easy to implement and fast to execute.

In Fig. 7, the element weights used for domain decomposition are shown for an adaptively refined macro triangle. One pre-order mesh traversal is needed to obtain these element weights. If a partitioning element is reached during the traversal, this element is stored as the current partitioning element. If a leaf element is reached, the weight of the current partitioning element (initially set to zero) is incremented by one. Note that in pre-order traversal the partitioning elements are visited before all of the corresponding leaf elements. The mesh of process i is not necessarily the finest on Ω_i because other processes have an overlap with Ω_i, in which they can refine also. Therefore, before the element weights are set, local Mesh Structure Codes are exchanged and merged, and the local mesh on process i is adapted to the composite mesh within Ω_i.

The goal of repartitioning is to optimize the load balance and to reduce total computation time. But the repartitioning itself is an expensive procedure. To repartition after every mesh adaptation is not necessarily the best choice. Repartitioning is useful only if the time loss due to load imbalance is higher than the time needed for the total repartitioning process. To consider this aspect, we introduce a mechanism which after each mesh adaptation step decides whether to repartition or not. First, on every process the sum over all element weights is computed, and then the sum average over all processes is built. After that every process compares its local sum to this average. If the difference is too large for at least one of the processes, a repartitioning is scheduled. More precisely, a repartitioning is done if any of the local weight sums sum_i satisfies one of the following inequalities:

$$\text{sum}_i > rt_{\text{high}} \cdot \text{sum}_{\text{average}} \quad rt_{\text{high}} > 1 \tag{1}$$

$$\text{sum}_i < rt_{\text{low}} \cdot \text{sum}_{\text{average}} \quad 0 < rt_{\text{low}} < 1, \tag{2}$$

where rt_{high} and rt_{low} stand for upper and lower repartitioning thresholds and $\text{sum}_{\text{average}}$ is the average over all local weight sums.

Algorithm 6 repartition on process i

```
1: adapt to composite mesh on $\Omega_i$
2: set element weights on $\Omega_i$
3: **if** repartitioning useful **then**
4:     compute new partitioning (in parallel)
5:     mark elements of new $\Omega_i$
6:     create local coarse grid level
7:     create overlap
8:     adapt to composite mesh on new $\Omega_i^+$
9:     exchange values
10:    coarsen outside of $\Omega_i^+$
11: **end if**
```

4.5 Building the global solution

After the parallel adaptation loop, each process i has computed a solution on a fine mesh within Ω_i^+ and on a relatively coarse mesh outside of this region. We build one global solution u_{PU} out of the N rank solutions u_i by a partition of unity method (see, [7]):

$$u_{PU}(x) := \sum_{i=0}^{N-1} \gamma_i(x) u_i(x) \quad \forall x \in \Omega, \tag{3}$$

where

$$\gamma_i(x) := \frac{W_i(x)}{\sum_{j=0}^{N-1} W_j(x)}. \tag{4}$$

Equation (4) ensures $\sum_{i=0}^{N-1} \gamma_i(x) = 1$ for all $x \in \Omega$, and $\gamma_i(x) \geq 0$ if $W_i(x) \geq 0$ for all $i \in [0 : N-1]$ and for all $x \in \Omega$. We define

$$W_i(x) := \sum_{\phi \in \Phi_i^c} \phi(x), \tag{5}$$

where Φ_i^c is the set of all linear basis functions of the local coarse grid level of partition i located at vertices with an overlap distance to Ω_i smaller than the given overlap size. This choice leads to functions W_i which are constant 1 within Ω_i, constant 0 outside of Ω_i^+, and have a linear slope in the overlap region. Figure 8 shows

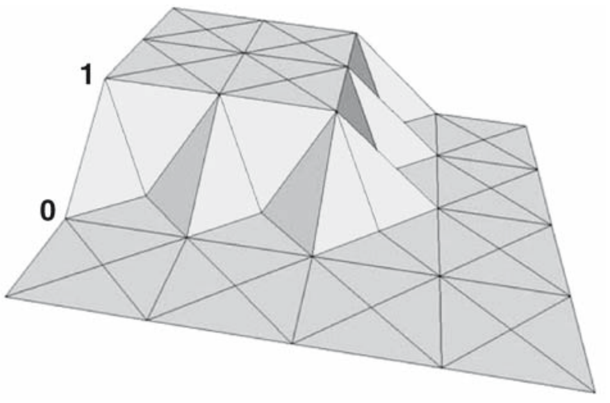

Fig. 8. Global view of W_0

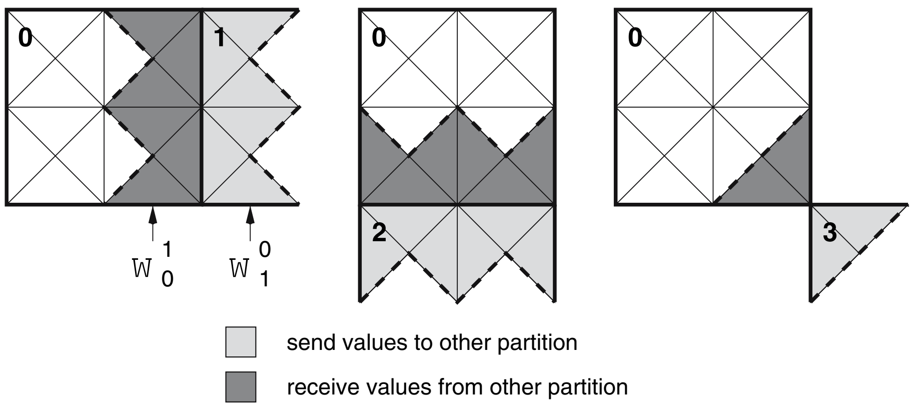
0
1
W_0^1
W_1^0
0
2
0
3
send values to other partition
receive values from other partition

Fig. 9. Necessary communication for process 0

such a function in the two dimensional case for an overlap size of 1. After construction of the γ_i functions on the local coarse grids, Eq. (3) can be evaluated at each discretization point of the fine composite mesh to obtain the final solution.

For a parallel computation of u_{PU}, each process i computes the partition of unity within its local partition Ω_i. For this purpose the process needs the local solution u_j of process j in $\Omega_i^j = \Omega_i \cap \Omega_j^+$ for all $j \neq i$. In Fig. 9, the communication scheme for one process is illustrated.

In [8], an upper bound for the error in H^1 semi norm resulting from the partition of unity is given. Assume $u \in H^2(\Omega)$, then $\|u - u_{PU}\|_{H^1} \leq C(h + H^2)$, where h is the maximal edge size of mesh i in Ω_i and H is the maximal edge size of mesh i in $\Omega \setminus \Omega_i$. In particular, if $h \leq \sqrt{H}$:

$$\|u - u_{PU}\|_{H^1} \leq C(h). \tag{6}$$

As in [1], we do not require this constraint in the simulations in Sect. 6. However, the results still fulfill our tolerance requirements on the H^1-error with the analytic solution, which was set to be 10^{-3}.

4.6 Time dependent and vector valued problems

The parallelization of time dependent problems is straight forward. Because repartitioning is provided already in the stationary adaptation loop, the partitioning will also adapt to mesh changes over time automatically (see Sect. 4.4). The basis for repartitioning is always the relatively coarse partitioning level, defined at the beginning of computation. In time dependent problems not only refinement but also coarsening of elements can occur. But the coarsest possible mesh is defined by the local coarse gird level and the global coarse grid level respectively introduced in Sect. 4.3. Further coarsening decisions are ignored. After each timestep the global solution is constructed by a partition of unity. And after each repartitioning the solution of the last timestep is sent from the former owner process of a fine element to the new owner process of this element, like it is done for all relevant values located at mesh nodes or elements.

If a system of PDEs should be solved in one vector valued problem, all PDEs are discretized on one common mesh and result in one linear equation system. So, all aspects concerning the mesh, e.g., the partitioning, are handled in exactly the same way as in the scalar case. Aspects concerning the values defined on the mesh, like value exchange and partition of unity, must be treated separately for each component, but also in the same way as in the scalar case.

5. Code example

In this section, we want to give an impression of how easy it is for the user to parallelize a given sequential code. The most work is done by a parallel problem class, which extends the original problem, and adds the needed parallelization abilities to it. Instead of the original problem this parallel problem is handed over to the adaptation loop. Before the loop is started an *initParallelization* routine has to be called. And after the loop has finished an *exitParallelization* routine has to be called. In the following example the relevant code lines of a stationary scalar problem called *parallelellipt* are shown:

```
#include "mpi.h"                                     // added
#include "ParallelProblem.h"                         // added
...
int main(int argc, char* argv[])
{
  MPI::Init(argc, argv);                             // added
  ...
  ParallelProblemScal parallelellipt
    ("ellipt->parallel", &ellipt);                   // added

  AdaptStationary *adaptationLoop =
    NEW AdaptStationary("ellipt->adapt",
                        &parallelellipt,           // modified
                        adaptInfo);
  parallelellipt.initParallelization(adaptInfo);  // added
  adaptationLoop->adapt();
  parallelellipt.exitParallelization(adaptInfo);  // added
  ...
  MPI::Finalize();                                // added
}
```

In this example *ellipt* is the name of the original sequential problem. After compiling and linking this code for MPI use, it can be started in parallel by *mpirun*. Needed parallelization parameters, like partitioning level or partitioning thresholds, can be set in a parameter file. Otherwise, they are set to predefined default values.

6. Numerical results

In this section, we present some numerical results that demonstrate the possibilities and the limits of our approach.

6.1 Varying local coarse grid level

To analyze the effect of the local coarse grid level, introduced in Sect. 4.3, we look at the following 2D Poisson problem:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= g \quad \text{on } \partial\Omega \end{aligned} \tag{7}$$

with

$$\begin{aligned} f(\vec{x}) &= n\,(\sin(2\pi n x) + \sin(2\pi n y)) \\ g(\vec{x}) &= \frac{1}{4\pi^2 n}\,(\sin(2\pi n x) + \sin(2\pi n y)) \end{aligned} \tag{8}$$

with $\vec{x} = (x, y)$. The right-hand side f and boundary function g are constructed such that the problem results in a sine-shaped solution. The scaling parameter n determines frequency and amplitude of the solution. In this example we set $n = 4$ and solve on $\Omega = (0, 1)^2$. Figure 10a shows the corresponding solution and partition boundaries for 4 partitions (thick lines) and 16 partitions (thin lines). Because of the periodic shape, in each partition nearly the same sub-problem has to be solved (except for different boundary situations). So, we obtain an optimal load balance in each iteration and no repartitionings become necessary. We perform the initial partitioning on a mesh consisting of 64 triangles (four global refinements of four macro elements), and measure the speedup to the corresponding sequential code for different choices of the local coarse grid level. The adaptation tolerance was set to 5×10^{-4}.

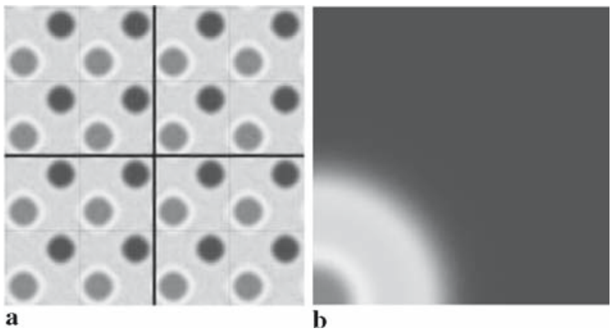

Fig. 10. Solutions of Eq. (7). (**a**) f and g according to Eq. (8) with $n = 4$. (**b**) f and g according to Eq. (9)

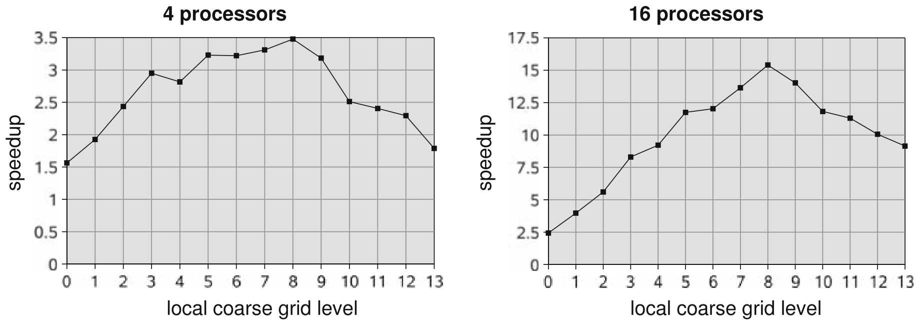

Fig. 11. Speedup factors for different local coarse grid levels with 4 and 16 processes. The local coarse grid level describes the number of uniform refinements within Ω_i starting from the partitioning level (see Sect. 4.3)

Figure 11 shows the resulting speedup factors for computations with 4 and with 16 processes. In both cases a local coarse grid level of 0 results in a very bad speedup. The reason is that here the overlap for each partition was computed at the relatively coarse partitioning level and covers a large part of Ω. On those overlap regions more than one process computes the solution on a fine mesh. In the worst case, where the overlap of each partition covers the whole remaining domain, every process would perform exactly the same computations on the same mesh on whole Ω. Including the overhead for partitioning and building the global solution, this would result in a speedup factor smaller than one. Increasing the local coarse grid level, the size of the resulting overlap will be decreased. For a local coarse grid level of 8 we obtain a speedup of 3.48 with 4 processes and a speedup of 15.41 with 16 processes. For higher values the speedup becomes worse again, because the number of needed adaptive iterations to reach the desired tolerance is getting smaller. So, the needed time is getting smaller, too, and the relative parallelization overhead grows.

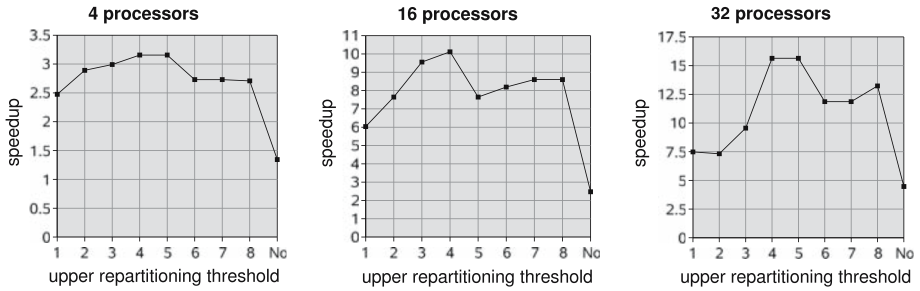

Fig. 12. Speedup for varying repartitioning values with 4, 16 and 32 processes. The lower repartitioning threshold is always set to the reciprocal of the upper threshold. The last value in each case describes the speedup when no repartitioning was performed at all

6.2 Varying repartitioning thresholds

While the example in Sect. 6.1 was constructed to avoid any repartitionings, the next example explores the influence of repartitioning thresholds, described in Sect. 4.4. For this purpose we use Eq. (7) again and define f and g as follows:

$$\begin{aligned} f(\vec{x}) &= -\left(400\vec{x}^2 - 40\right) e^{-10\vec{x}^2} \\ g(\vec{x}) &= e^{-10\vec{x}^2}. \end{aligned} \tag{9}$$

This problem can be seen as worst case scenario for the parallelization approach, because the irregularity of the solution can never be resolved by all processes equally.

The solution on $\Omega = (0, 1)^2$ is shown in Fig. 10b. The fact that the source is located in a corner of the domain leads to successive refinements towards this corner and to load imbalance between the partitions. We use the eight times global refined macro mesh as the partitioning level and set the local coarse grid level to 4. Then we solve the problem with 4, 16 and 32 processes with a tolerance of 5×10^{-4} and with varying values of upper and lower repartitioning thresholds. Then we compare the computation times with the sequential case. In Fig. 12, the results are shown for upper repartitioning thresholds between 1 and 8. The lower repartitioning threshold was always set to the reciprocal of the upper one. *No* on the x-axis means that no repartitioning was performed during the whole computation. This was realized by setting the upper threshold to 10,00,000 and the lower one to 0. As one can see, the optimal value for the upper threshold in this example is four in all cases. The benefit compared to an upper threshold of one increases when more processes are used. Whereas the overall relative speedup is worse if more processes are used. The worst speedup is achieved in each case when no repartitioning was performed. This is further illustrated by comparing our approach with [1].

The optimal choice for the repartitioning thresholds is probably problem specific. Therefore, we added an algorithm which adapts the thresholds depending on the relationship between the time used for the last repartitioning and the elapsed computation time since the last repartitioning. The resulting speedup was similar to that with the fixed optimal thresholds.

6.3 Comparison with the approach of Bank and Holst

Bank and Holst make the assumption that partitions with approximately equal error lead to approximately equal work for each process. They note in [1] that this is a fragile assumption, but show that it works well for several examples.

However, for the problem defined in Sect. 6.2 this assumption does not hold. Using the approach of Bank and Holst, partitioning is done only once at a relatively coarse mesh at the beginning of the computation. Element weights are the local error estimates of this mesh. We use four processes. The partitioning results in four partitions with approximately the same estimation sums. The highest errors were estimated in the lower left corner of the domain. This leads to small partitions 0, 1 and 2 around this corner, compared to the much larger partition 3, see Fig. 13a.

Like described in [1] we avoid any communication within the adaptation loop to decouple the iterations of the different processes. So, decisions concerning which elements are marked for refinement and when the total tolerance is reached, can be based only on local process information. We distribute the total tolerance equally between the four processes, which is reasonable if the assumption holds that equal errors lead to equal work loads.

In Fig. 13b the mesh after the parallel computation is shown. Figure 13c shows the mesh after the corresponding sequential computation, which differs from the parallel computation. In this example, the reached speedup factor was 0.6, which actually is a slow down for the parallel case. One reason for this bad result is that it needs much less work to reduce the error on the small partitions 0, 1 and 2 with very few elements, than on the large partition 3, where the same error is distributed on many more elements. The final mesh of partition 3 has over 3,25,000 elements, whereas the meshes of partition 0 and 1 have about 7,000 elements, and the mesh of partition 2 has 20,000 elements. So, nearly the whole work is done by process 3.

This effect is enhanced by distributing the tolerance equally between the processes. We force each process to reduce the error by the same factor, which is not what happens in the sequential case. Bank and Holst stop the computations at each process, when a given target number of elements or degrees of freedom is reached. This, however, does not guarantee a solution, which fulfills the given tolerance criterion. Furthermore, a comparison with the sequential case can not be done. Therefore, we use the tolerance to stop the computation.

For this example our approach provides a far better speedup than the Bank and Holst approach does. In Sect. 6.2, we showed that, with the right repartitioning thresholds, a speedup of over 3 can be reached. Even if no repartitionings are applied, the speedup is still better than with the Bank and Holst approach, see Fig. 12.

6.4 Scaled problem domain

The previous examples showed that the relative speedup (speedup divided by number of used processes) for a given problem gets worse, when the number of processes grows. One way to avoid that, is to adapt the local coarse grid level according to the number of used processes. In this section, we analyze another kind of scalability: We use a higher number of processes to solve an accordingly more complex problem.

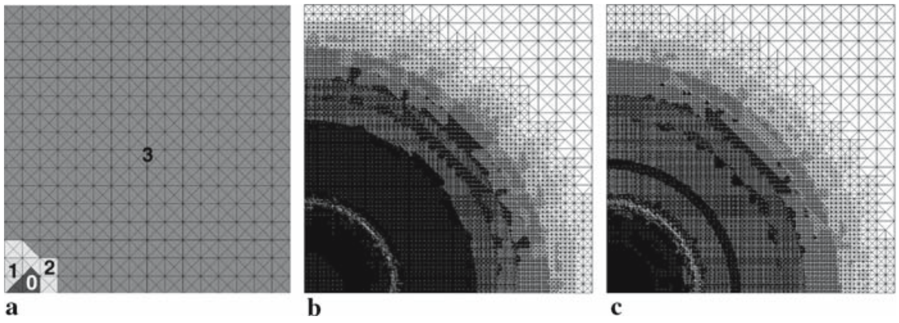

Fig. 13a. The four partitions with approximately equal estimation sums. (**b**) The composite mesh after the parallel computation. (**c**) The mesh after sequential computation

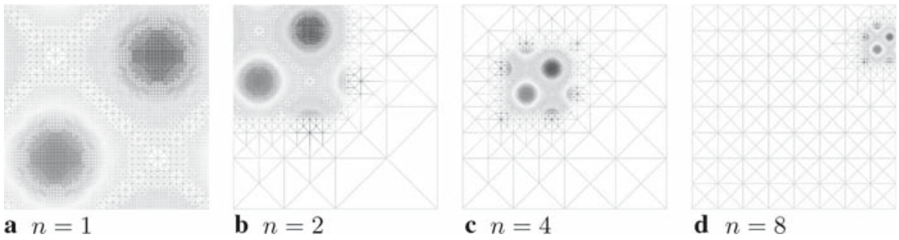

Fig. 14. Final mesh of partition 0 in the different cases. The value of n corresponds to the square root of the used process number

Table 4. Time factor compared to the sequential case

p	1	4	16	64
Factor	1	1.279	1.496	1.589

This is a realistic scenario for many problems in materials science, where the overall goal is not a reduction of computing time but the increase in domain size, see, e.g., [9]. We use the problem defined by Eqs. (7) and (8) in Sect. 6.1 again, but now we use different values for n. To double the value of n is equivalent to double the x and y expansion of Ω and continue the problem formulation on the extension. We use $n = \sqrt{p}$ with p the number of processes for $p \in \{1, 4, 16, 64\}$. The partitioning level is set to $log_2(p)$, the local coarse grid level is set to 8. In Fig. 14, the final meshes for partition 0 in the different cases are shown.

In Table 4, the resulting time factors compared to the sequential case are shown. Solving the problem, which is 64 times more complex than the original one, needs only about 1.6 times the time when using 64 processes.

6.5 Moving source

In this section, we give an example of a simple time dependent problem solved in parallel with four processes. The problem is defined by

$$\begin{aligned} u_t(\vec{x},t) - \Delta u(\vec{x},t) &= f(\vec{x},t) \quad \text{in } \Omega \\ u(\vec{x},t) &= g(\vec{x},t) \quad \text{on } \partial\Omega. \end{aligned}$$

The function u_t is the time derivative of u. The functions f and g are chosen such that the source is moving from the lower left corner to the upper right corner of $\Omega = (0,1)^2$ while the time t proceeds from 0 to 1. Furthermore, the source amplitude grows from 0 to 1 until $t = 0.5$ and falls back to 0 until $t = 1.0$ following a sine function. We use a fixed timestep of 0.1 and an implicit time strategy, which adapts in each timestep until the tolerance of 10^{-4} is reached. Also element coarsening is allowed if local error indicators are sufficiently small. We use a partitioning level of 6 and a local coarse grid level of 6. Upper and lower repartitioning thresholds are set to $\frac{3}{2}$ and $\frac{2}{3}$, respectively. In Fig. 15, we can see, how the solution changes over time. The meshes at the different timesteps are adapted to the current solution. On the right part of the figure we see the partitions at the beginning of the corresponding timesteps. One can see that the partitioning follows the mesh changes to obtain

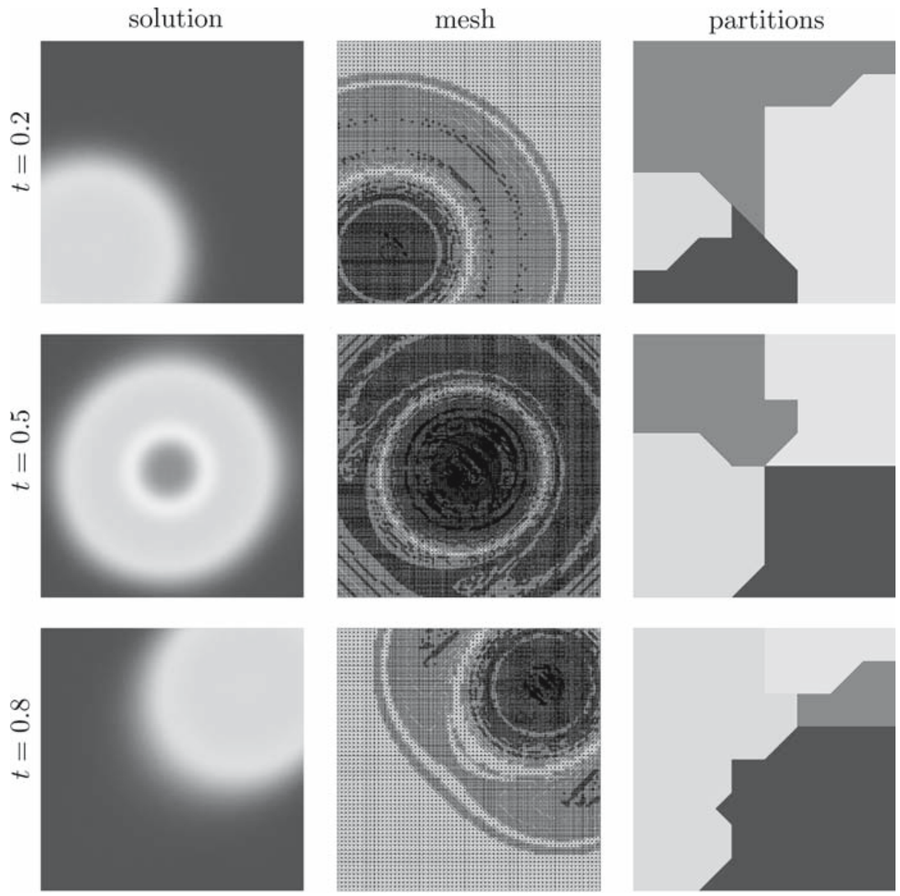

Fig. 15. Solution, resulting mesh and corresponding partitions at different timesteps

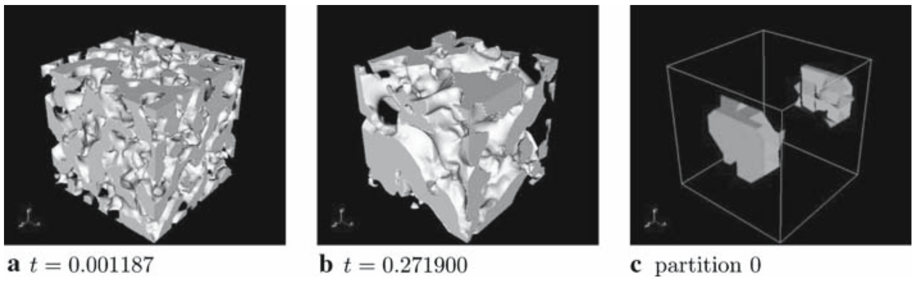

Fig. 16a–b. Evolution of u at two timesteps. The value $u = 0$ is denoted by opaque volume, $u = 1$ by transparent volume. (**c**) Volume of partition 0, connected through periodic boundary conditions. The simulations were performed by A. Rätz

a good load balance in each timestep. The speedup compared to the sequential solution of the same problem was 3.65.

6.6 Higher-order problem in 3D

This is an example of a time dependent three dimensional parallel computation for a system of PDEs. Here we look at a classical model for spinodal decomposition of a binary alloy, the Cahn–Hilliard equation. The model is used to describe coarsening dynamics in phase separation processes, which occur in quenched alloys. For numerical approaches we refer to [10–12]. The Cahn–Hilliard equation is a fourth-order problem that reads

$$u_t = \Delta\left(-\epsilon\Delta u + \epsilon^{-1}G'(u)\right), \tag{10}$$

with $G(u) = 18u^2(1-u)^2$ a double well potential and ϵ a small parameter. The values $u = 0$ and $u = 1$ are the two stable steady states, representing the two phases. As initial conditions we use a small zero mean perturbation of $u = 0.5$. We write (10) as a system of two second-order equations

$$\begin{aligned} u_t &= \Delta w, \\ w &= -\epsilon\Delta u + \epsilon^{-1}G'(u), \end{aligned} \tag{11}$$

discretize in space by linear finite elements, linearize the derivative of the double-well potential, apply a semi-implicit time-discretization and solve the resulting linear system by an iterative solver.

We solve the problem on $\Omega = (-1,1)^3$, discretized by a fix mesh consisting of about 6.3 million tetrahedra, and we apply periodic boundary conditions on $\partial\Omega$. This discretization is too fine to be covered by one processor. Here we used 24 processes. Figure 16a and b shows the evolution of u at different time steps. The solution quickly separates Ω into two regions Ω_0 and Ω_1, where u takes the values of 0 and 1, respectively. The remaining part of Ω lies on an interface of width $O(\epsilon)$ between the two regions. At later stages, Ω_0 and Ω_1 change shape so that the surface of the interface between the two regions decreases while maintaining the volume of Ω_0 and Ω_1. In Fig. 16c, the volume of partition 0, connected through a periodic boundary, is visualized.

7. Conclusion

With the concept of adaptive full domain covering meshes presented in this paper, sequential codes can be parallelized in an easy way producing little communication needs. Mesh Structure Codes are used to exchange mesh information in a very efficient way. The parallel speedup can be optimized by varying parallelization parameters like local coarse grid level and repartitioning thresholds. If the problem complexity increases together with the number of used processes, a good scalability of the approach can be observed. Time dependent problems and systems of PDEs can be treated in a straightforward way. The number of processes used in the examples are moderate. If the number of processes drastically increases, the need for communication in traditional parallelization concepts becomes an issue. We believe our concept of adaptive full domain covering meshes to be a useful tool to overcome this problem as the need for communication is reduced to a minimum.

References

[1] Bank, R. E., Holst, M.: A new paradigm for parallel adaptive meshing algorithms. SIAM Rev **45**(2), 291–323 (2003)
[2] Mitchell, W. F.: Parallel adaptive multilevel methods with full domain partitions. Appl Num Anal Comput Math **1**, 36–48 (2004)
[3] Vey, S., Voigt, A.: Amdis – adaptive multi-dimensional simulations. Comput Visual Sci **10**, 57–67 (2007)
[4] Schmidt, A., Siebert, K. G.: Design of adaptive finite element software. LNCSE, vol. 42. Springer, Heidelberg (2005)
[5] Ribalta, A., Vey, S., Voigt, A.: Error reduction in adaptive full domain covering meshes for parallel computing. Num Math (in review)
[6] Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. Concurrency and Computation: Practice Experience **14**, 219–240 (2002)
[7] Babuska, I., Melenk, J. M.: The partition of unity method. Int J Numer Meth Engng **40**, 727–758 (1997)
[8] Holst, M.: Applications of domain decomposition and partition of unity methods in physics and geometry. Proc. 14th Int. Conf. on Domain Decomposition Methods, pp. 63–78, 2002
[9] Backofen, R., Rätz, A., Voigt, A.: Nucleation and growth in a phase field crystal (PFC) model. Phil Mag (in review)
[10] Barrett, J. W., Blowey, J. F.: Finite element approximation of the Cahn–Hilliard equation with concentration dependent mobility. Math Comput **68**, 487–517 (1999)
[11] Kim, J., Kang, K.K., Lowengrub, J.: Conservative multigrid methods for Cahn–Hilliard fluids. J Comput Phys **193**(2), 511–543 (2004)
[12] Feng, X. B., Prohl, A.: Numerical analysis of the Cahn–Hilliard equation and approximation for the Hele–Shaw problem. Interfaces Free Bound **7**(1), 1–28 (2005)