Parallel Multigrid in an Adaptive PDE Solver Based on Hashing

Michael Griebel and Gerhard Zumbusch[a]

[a]Institut für Angewandte Mathematik, Universität Bonn,
Wegelerstr. 6, D-53115 Bonn, Germany

Partial differential equations can be solved efficiently by adaptive multigrid methods on a parallel computer. We report on the concept of hash-table storage techniques to set up such a code. The code requires substantial less amount of memory and is easier to code in the sequential case. The parallelization takes place by a space filling curve domain decomposition intimately connected to the hash table. The new data structure simplifies the parallel version of the code substantially way and introduces a cheap way to solve the load balancing and mapping problem.

1. INTRODUCTION

We consider partial differential equations, e.g. an elliptic scalar differential equation on a two-dimensional domain. For reasons of efficiency, we use an optimal order solution algorithm: The dicretized equation system is solved by a multigrid method. In order to accelerate the solution procedure further, adaptivity is employed to achieve a given error tolerance with less unknowns. The grid is adapted to the solution and is refined only in regions of the domain where necessary. A third way to speed up the computation is parallel computing. We partition the data and distribute it to several processors and we assign the operations on that data preferably to the processor who owns the data. We intend to put all three methods (multigrid, adaptivity, parallelism) efficiently together. This requires some additional considerations.

While state-of-the-art computer codes use tree data structures to implement such a method, we will propose hash tables instead. Hash table addressing gives more or less direct access to the data stored (except of the collision cases), i.e. it is proven to possess a $\mathcal{O}(1)$ complexity with a low constant if a statistical data distribution is assumed. Hash tables allow to deal with locally adapted data in a simple way. Furthermore, they need no additional storage overhead for logical connectivities like tree-type data structures which are usually used in adaptive finite element codes. Finally, they are easy to program and to handle and allow a straightforward implementation. We studied their efficiency and compared it to that of other data structures like trees and linked list. It turned out that the hash table technique was in all considered cases superior with respect to storage requirements and with respect to computing time. Furthermore, and this is an additional advantage of the hash table methodology, it allows relatively easy for parallelization with simple load balancing.

2. DATA STRUCTURES FOR ADAPTIVE CODES

2.1. Adaptive Cycle

In order to adapt the grid to the solution without a priori knowledge, where to refine, we use an iterative procedure: Solving the problem on one grid and employing an error estimator gives information where a finer grid is needed to resolve the solution. We start the iterative procedure with a very coarse grid. In each step we add new nodes. If a final error tolerance is matched, we have obtained our solution on a fine adapted grid. This basic approach to adaptivity is depicted in figure 1.

```
    solve linear system
    estimate local errors
    flag nodes for refinement
    create new nodes and
    enforce geometric conditions
until global error $\leq \epsilon$
```

Figure 1. The adaptive algorithm.

The sequence of solving and refining the grid leads in a natural way to a sequence of nested grids. We can use the solution on one grid as a starting guess for the iterative solution on the refined grid. Furthermore, the sequence of grids can be used as different levels inside a multigrid method. So, the two components, adaptive refinement and iterative solution by a multigrid method fit nicely together.

2.2. Tree Data Structures

The efficient implementation of an adaptive multigrid code is not straightforward. Since a multigrid solver has optimal $\mathcal{O}(n)$ complexity for n unknowns, the same complexity is desirable for its implementation on refined grids and for the grid refinement procedure itself. One way to construct such an optimal order algorithm is to use tree data structures. This is described in more detail in [1,12,9]. Different trees represent the hierarchies of nodes, edges and elements. All entities of a tree down to one level altogether represent one grid. Refining the finest grid means adding new leaves to the tree. Moreover, to run multigrid is actually to traverse the tree several times and to perform operations on all nodes.

In order to administrate the nodes (unknowns), edges (stiffness matrix) and elements (grid), the leaves of the trees have to be linked. This results in a number of pointers, both for the tree and for the links between the trees. A typical value is 400 to 1000 bytes of memory per unknown for a scalar two-dimensional problem (double precision, see [7,2]). In three dimensions, even more memory per node is needed. Consequently, there is much more memory required for the administration of the numerical data than for the data itself; the storage overhead is substantial.

Now, performing numerical operations on one grid often requires a complete tree traversal. But such a complete tree traversal for e.g. one `daxpy` operation adds a certain number of index operations and administration operations to the few floating point operations we are interested in, which degrades the overall floating point performance. Besides, this also results in fragmented memory access, which degrades performance even further. Of course it is possible to eliminate some of the tree traversals by establishing additional data structures like linked lists or sparse matrices but only at the expense of additional memory.

2.3. Hash Table Data Structures

Looking for a different way to manage adaptive grids, we propose to use hash storage techniques. *Hash tables* are a well established method to store and retrieve large amounts of data [6, chap. 6.4]. The idea is to map each entity of data to a *hash-key*, which is used as an address in the hash table. The entity is stored and can be retrieved at that address in the hash table, which is implemented as a linear array of cells (buckets). The mapping is done by a (deterministic) hash function. Since there are many more possible different entities than different hash keys, the hash function is not injective. Algorithms to resolve collisions are needed. Furthermore it may also happen that some entries in the hash table are left empty, because no present entity is mapped to that key.

In general, access to a specific entry in the hash table can be performed in constant time. However, this is only true if the hash function scatters the entries broad enough and there are enough different cells in the hash table. But altogether, using hash tables in a statistical sense, access is cheaper than random access in a sorted list or a tree. The basic principle of the hash table approach is illustrated in figure 2.

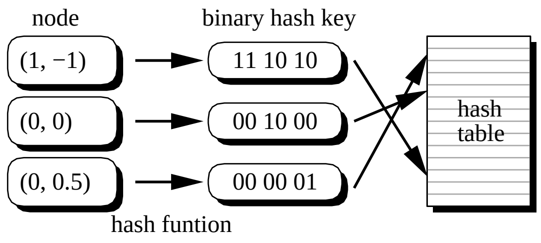

Figure 2. Storing nodes in a hash table.

We presently use the hash table implementation of an extended version of the C++ standard template library (STL) [11], which provides automatic resizing. There, the number of cells is kept proportional to the number of entries and we only have to bother with a well suited hash function.

Iterating over the contents of a hash table is simpler than traversing trees of elements or nodes. STL iterators simplify the coding even further. In a general statistical setting the random access on some data in a hash table is constant, compared to logarithmically in a tree structure [6]. In total an amount of 74 bytes (double precision) of memory per unknown is used in our implementation, which is less than standard tree based codes mentioned above use. This is due to the lack of pointers in the hash table code.

3. THE ADAPTIVE CODE

3.1. Adaptive Finite Differences

We take a strictly node-based approach. The nodes are stored in a hash table. Each node represents one unknown. Neither elements nor edges are stored. We use a one-irregular grid with 'hanging' nodes, see figure 3, who's value is determined by interpolation. This is equivalent to the property that there is at most one 'hanging' node per edge. Additionally we consider only square shaped elements. The one-irregular condition is a kind of a geometric condition for the adaptive grid.

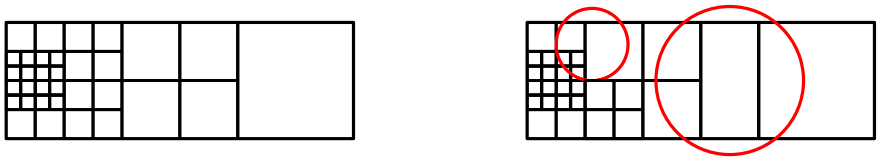
Figure 3. A one-irregular grid (left) and a grid containing several mistakes (right). Look for the edge with multiple 'hanging' nodes and for the rectangular shaped element, indicated by the labels.

The partial differential equation is discretizatized by finite differences. We set up the operator as a set of difference stencils from one node to its neighboring nodes in the grid, which can be easily determined: Given a node on a level, its neighbors can be only on the same level, or one level up or down. The distance to the neighbor is determined by the level they share.

So pure geometric information is sufficient to apply the finite difference operator to some vector. We avoid the storage of the stiffness matrix or any related information. For the iterative solution of the equation system, we have to implement matrix multiplication, which is to apply the operator to a given vector. A loop over all nodes in the hash table is required for this purpose.

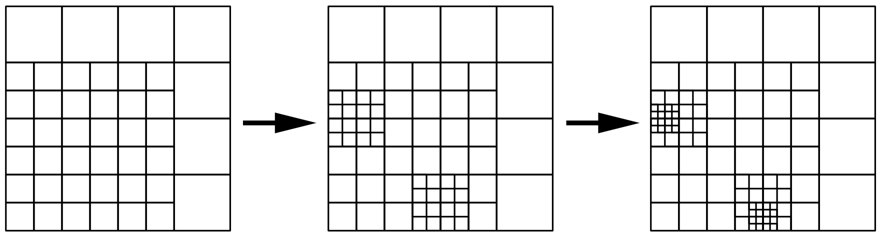
Figure 4. A Sequence of adaptively refined grids.

3.2. Multigrid Preconditioning

We use an additive version of the multigrid method for the solution of the equation system, i.e. the so called BPX preconditioner [5]. This requires an outer Krylov iterative solver. This approach has the advantage of an optimal $\mathcal{O}(1)$ condition number and there are optimal order $\mathcal{O}(n)$ implementations available, even in the presence of degenerate grids. The implementation is similar to the hierarchical basis transformation, requires one auxiliary vector, but enjoys a much better condition number than the hierarchical basis method. Two loops over all nodes are necessary, one for the restriction operation and one for the prolongation operation. They can be both implemented as a tree traversal. However, by ordering the nodes in the right way, two ordinary loops over all nodes in the hash table are sufficient, one forward and one backward. Furthermore, the additive version of multigrid is also easier to parallelize than multiplicative multigrid versions.

3.3. Error Estimation and Grid Refinement

In order to create adaptive grids, we have to locate areas, where to refine the grid. Applying an error estimator or error indicator gives an error function defined on the grid. With some threshold value, the error functions is converted into a flag field, determining whether grid refinement is required in the neighborhood. Large error values result in refinement. In the next step, new nodes are created. Finally a geometric grid has to be constructed, which fulfills the additionally imposed geometric constraints, e.g. one-irregularity.

The main question is the error indicator to be applied. There are lots of different suggestions. We took a type of gradient based criterion for each node. The implementation is very similar to the implementation of a matrix multiply. The threshold value is determined by a mixture of mean value of the local errors and some constraints on the minimum number of nodes to create. The grid creation requires a tree traversal to check the one-irregularity condition.

4. DISTRIBUTING ADAPTIVE GRIDS

Porting the adaptive multigrid code to a parallel computer requires a grid partitioning strategy. The computational load and therefore the grid has to be decomposed into several partitions, each one mapped to one processor. This partitioning has to be done at run time since there is no a priori knowledge where the grid is going to be refined and where load is created. Thus we extend our adaptive cycle algorithm by an additional load balancing and partitioning step right after new nodes are created.

A good solution to the partitioning problem is a key point of the efficiency of adaptive parallel codes. The computational load has to be equidistributed over the processors. The volume of data to be transferred between processors during computation and during mapping the partitions to processors should be low. Furthermore the load balancing and mapping process should be cheaper than the actual computation. Often the last demand is violated, so the balancing is applied less often, taking into account some load imbalance and a harder mapping problem, since more load has to distributed in each mapping step.

4.1. Space-Filling Curves

For the solution of the partition and mapping problem, we choose a computational very cheap method based on space-filling curves [13,4]. We use space-filling curves as a way to enumerate and order nodes in the computational domain. Such a curve can be recursively defined by substituting a straight line segment by a certain pattern of lines. Doing this infinitely times covers the whole domain. Doing this recursion only a finite number of times results in a coarser version of the curve. One can think of such a curve as passing the nodes of a given grid, e.g. an adaptive grid. In order to reach all points, the curve has to fine enough, it has to be aligned, and it has to cover the whole domain. Because of the boundary nodes, it usually covers even a larger domain and is clipped, see figure 5. We assign the arc length of the curve to each node of the grid and define a total order relation on the nodes. The curve defines a (continuous) mapping from an interval to the whole domain.

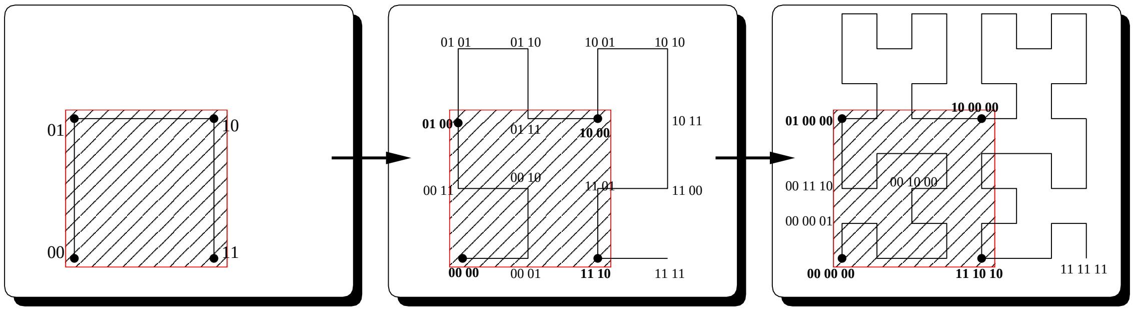

Figure 5. Hilbert's space-filling curve at different levels of resolution. It covers the whole domain, say $\Omega = [-1,1]^2$ (drawn shaded). The nodes are numbered binary from 0 to $4^j - 1$.

4.2. Data Distribution

Given an order of the nodes induced by a space-filling curve, it is simple to construct a partitioning of the set of nodes and to map these partitions to processors: We cut the linear list of nodes into p equally sized intervals and map them according to this order to processors with increasing numbers. The computational load is balanced well, see [16,10].

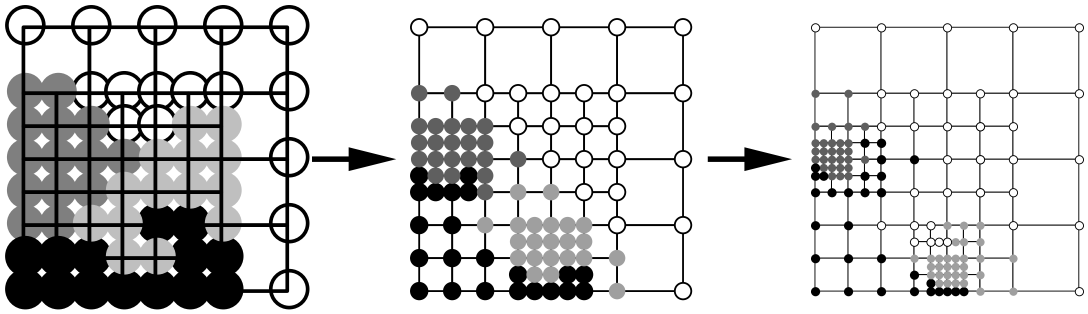

Figure 6. A Sequence of adaptively refined grids mapped onto four processors.

5. THE PARALLEL ADAPTIVE CODE

For the parallelization of the sequential code, all its components such as solving a linear system, estimating errors and creating nodes have to be done in parallel. Additionally the data has to be distributed to the processors. This is done in a load balancing and mapping step right after creating new node, a step which was not present in the sequential version. The parallel algorithm is depicted in figure 7. We consider a distributed memory, MIMD, message passing paradigm. This is much harder than parallelizing such a code on a shared memory computer as considered in [7,3]. Parallelizing a tree based code is quite complicated and time consuming. Here, algorithms must be implemented on sub-trees. Furthermore, algorithms for moving and for joining sub-trees must be implemented. Finally all this must be done in a consistent and transparent way, as indicated in [17,15, 2,14,8]. However, the parallelization of an adaptive code based on hash tables, which we consider here, will turn out to be much easier.

```
    solve linear system in parallel
    estimate local errors in parallel
    flag nodes for refinement in parallel
    create new nodes and
    enforce geometric conditions in parallel
    redistribute nodes,
    load balancing and mapping
until global error $\leq \epsilon$
```

Figure 7. The parallel adaptive algorithm.

5.1. Parallel Load Balancing

Using the space filling curve, the partitioning problem reduces to a sorting problem. Running this in parallel requires a parallel sort algorithm with distributed input and output. We employ a one-stage *radix sort* algorithm [6, chap. 5.2.5]. In order to do this we assume that the previous data distribution still guarantees good load-balancing for the parallel sort.

The result is a new partition of the grid. The load is partitioned exactly, while the volume of communication depends on the boundaries of the partitions. These boundaries may sometimes be kinky, depending on the data, and are certainly not optimal, but are of reasonable size. In total, the load balancing is very cheap, parallelizes very well and thus can be applied in each step of the computation.

The index of a node induced by the space-filling curve is used for assigning it to a processor and for addressing in the local hash table of the processor, see figure 8. In case that a copy of a node is required on another processor, the index is also used for addressing it in the hash table of this processor. Examining the index, it is easy to determine the processor, the node belongs to.

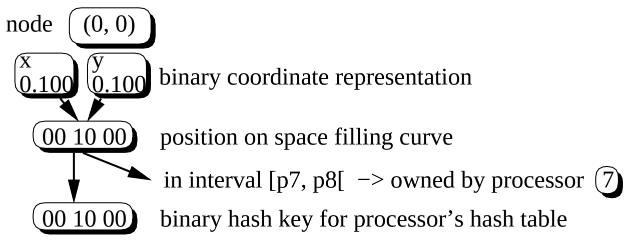
node
(0, 0)
x
0.100
y
0.100
binary coordinate representation
00 10 00
position on space filling curve
in interval [p7, p8[−> owned by processor
7
00 10 00
binary hash key for processor's hash table

Figure 8. Computing the owner processor and the hash key of a node.

5.2. Parallel Krylov Iteration

The parallel iterative solution consists of several components. The Krylov iterative solver requires matrix multiplications, scalar products and the application of the preconditioner which is the additive multigrid method in our case. The scalar product can be implemented as ordinary data *reduction*[1] operations offered by any message passing

[1]The reduction operation "+" is defined as a global sum, one number from each processor, distributed to all processors. A similar operation computes the global maximum.

library. Using the "owner computes"[2] paradigm, the matrix multiplication, requires the update of auxiliary (ghost) values located at the boundary of the partition. The variables of ghost nodes in this region are filled with actual values. Then, the local matrix multiplication can take place without any further communication, and only one local nearest neighbor communication is sufficient.

5.3. Parallel Multigrid Preconditioning

The communication pattern of the additive multigrid is more expensive than for the matrix multiplication: The local restriction operations can be performed in parallel without any communication. The resulting values have to be *reduced* and distributed. Each node sums up the values of all it's distributed copies. This can be implemented by two consecutive communication steps, fetching and distributing the values. Now the restricted values are present on all nodes and ghost nodes. Finally, the reverse process of prolongation can take place as local operations again. Thus the result is valid on all nodes without the ghost nodes.

The local restriction and prolongation operations are organized as ordinary restriction and prolongation, just restricted to the local nodes and ghost nodes on a processor. They can be implemented either as tree traversals or as a forward and a backward loops on properly ordered nodes, i.e. on the hash table. The ghost nodes are determined as set of ghost nodes of grids on all levels. Hence the communication takes place between nearest neighbors, where neighbors at all grid levels have to be considered. In this sense the communication pattern is a local one only. One fetch and one distribute step are necessary to exchange all data.

Compared to multiplicative multigrid methods, where communication on each level takes place separately in the smoothing process, the hierarchical nearest neighbor communication is a great advantage [17]. However, the total volume of data to be communicated in the additive and the multiplicative multigrid method are of the same order (depending on the number of smoothing steps). This means that for high communication latency computers the additive multigrid has an advantage, while for low latency computers the number of communication steps is less important.

5.4. Adaptive Refinement

The estimation of errors in parallel is structurally very similar to the iterative solution of the equation system, so a nearest neighbor communication pattern applies. Flagging nodes for refinement and creating nodes can also be done easily with some nearest neighbor communication. A global threshold for flagging nodes requires a global reduction operations.

The tricky part of adaptive refinement is the enforcement of a one-irregular grid condition, see section 3.1. This means fulfilling some geometric constraint on the grid. A refinement in one area theoretically may spread out to the whole domain. Instead of weakening the constraint in the presence of partition boundaries, we change the parallel programming paradigm here: We employ some parallel tree traversal, breadth first, which leads to a data driven execution model. The tree traversal fills lists of requests for

[2]Each variable belongs to one processor. Any write operation to the variable is done by that processor, along with all computations, which lead to the value to be written: "Owner computes"

neighbor processors, which are blocked and transferred. The algorithm terminates when all lists have been consumed, implemented by some token ring mechanism.

6. NUMERICAL EXPERIMENTS

We consider the two dimensional model problem

$$\begin{aligned} -\Delta u &= 0 \quad \text{on } \Omega = [-1,1]^2 \\ u &= 0 \quad \text{on } \partial\Omega \setminus [-1,0[^2 \\ u &= 1 \quad \text{on rest of } \partial\Omega \end{aligned}$$

and run our adaptive multilevel finite difference code to solve it. The solution develops two singularities located at the jumps in the boundary data $(-1,0)$ and $(0,-1)$. All numbers reported are scaled CPU times measured on a cluster of SGI O2 workstations, running the MPI message passing protocol on a fast ethernet network.

We choose the simple Poisson problem, because the ratio of computational work to communication is low compared to more complicated equations like the Navier-Stokes equations. In this sense the Poisson problem is a very hard test problem for parallelization.

6.1. Uniform Example

In the first test we consider regular grids (uniform refinement). Table 1 shows wall clock times for the solution of the equation system on a regular grid of different levels. The program ran on different numbers of processors.

time		processors			
		1	2	4	8
levels	6	0.0580	0.0326	0.0198	0.0473
	7	0.2345	0.1238	0.0665	0.1861
	8	1.0000	0.4914	0.2519	0.2350
	9			1.1297	0.6282

Table 1
Uniform refinement example, timing, levels 6 to 9, 1 to 8 processors.

We observe a scaling of a factor of 4 from one level to the next finer level which corresponds to 4 times more unknowns on that level. The computing times decay, if we increase the number of processor from one to two and four. However, the 8 processor perform efficiently only for sufficiently large problems, i.e. problems with more than 8 levels. We observe perfect scalability of the algorithm.

6.2. Adaptive Example

In the next test we consider adaptive refined grids. The grids are refined towards the two singularities. Table 2 depicts times in the adaptive case. These times give the wall clock times for the solution of the equation system again, now on different levels of adaptive grids and on different numbers of processors.

We obtain a scaling of about a factor 4 from one level to the next finer level, which means 4 times more unknowns on the next finer level, an almost uniform refinement of a grid already adapted on previous levels. Increasing the number of processors speeds up the computation accordingly. Again we observe scalability of the algorithm. In order to

time		processors			
		1	2	4	7
levels	7	0.0578	0.0321	0.0187	0.0229
	8	0.2291	0.1197	0.0645	0.0572
	9	1.0000	0.5039	0.2554	0.1711

Table 2
Adaptive refinement example, timing, levels 7 to 9, 1 to 7 processors.

use seven processor efficiently, the grid has to be fine enough, i.e it has to have more than 8 levels.

6.3. Load Balancing

Now we compare the time for solving the equation system with the time required for sorting the nodes and mapping them to processors. The ratio indicates how expensive the load balancing and mapping task is in comparison to the rest of the code. We give the values in table 3 for the previous uniform and adaptive refinement examples for different numbers of processors.

$\frac{\text{sort time}}{\text{solve time}}$	processors			
	1	2	4	7/8
uniform	0	0.0028	0.0079	0.0141
adaptive	0	0.0066	0.0149	0.2367

Table 3
Ratio sorting nodes to solving the equation system, level 8, 1 to 8 processors.

In the single processor case, no load balancing is needed, so the sort time to solve time ratio is zero. In the uniform grid case the numbers stay below two percent. In the adaptive grid case, load balancing generally is more expensive. But note that load balancing still is much cheaper than solving the equation systems. However, higher number of processors make the mapping relatively slower.

In the case of uniform refinement, for a refined grid, there are only few nodes located at processor boundaries which may have to be moved during the mapping. Hence our load balancing is very cheap in this case. Mapping data for adaptive refinement requires the movement of a large amount of data, even if most of the nodes stay on the processor.

All load balancing methods also have to face the problem of large data to move, especially for larger processor numbers, see [2]. Moving data can make up a substantial part of the overall computing time, even if incremental load balancing methods are employed. Hence it is very important to have a cheap method at hand, such as our space filling curve procedure.

7. CONCLUSION

We have introduced hash storage techniques for the solution of partial differential equations by a parallel adaptive multigrid method. Hash tables lead to a substantial reduction of memory requirements to store sequences of adaptive grids compared to standard tree based implementations. Furthermore, the implementation of an adaptive code based on hash tables proved to be simpler than the tree counterpart. Both properties, low amount of memory and especially the simple programming, carried over to the parallelization of the code. Here space filling curves were used for data partitioning and at the same time for providing a proper hash function.

The results of our numerical experiments showed that load balancing based on space filling curves is indeed cheap compared to other more complex graph based heuristics. Hence we can in fact afford to use it in each grid refinement step. Thus our algorithm operates on load balanced data at any time. This is in contrary to other procedures, which have to be used often in connection with more expensive load balancing mechanisms, where load imbalance is accumulated for several steps.

REFERENCES

1. R. E. Bank and T. F. Dupont. An optimal order process for solving elliptic finite element equations. *Math. Comp.*, 36:967–975, 1981.
2. P. Bastian. *Parallele Adaptive Mehrgitterverfahren.* B. G. Teubner, Stuttgart, 1996.
3. K. Birken. Ein Parallelisierungskonzept für adaptive, numerische Berechnungen. Master's thesis, Universität Erlangen-Nürnberg, 1993.
4. S. H. Bokhari, T. W. Crockett, and D. N. Nicol. Parametric binary dissection. Technical Report 93-39, ICASE, 1993.
5. J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.
6. D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.
7. P. Leinen. *Ein schneller adaptiver Löser für elliptische Randwertprobleme auf Seriell– und Parallelrechnern.* PhD thesis, Universität Dortmund, 1990.
8. W. Mitchell. A parallel multigrid method using the full domain partition. *Electronic Transactions on Numerical Analysis*, 97. Special issue for proceedings of the 8th Copper Mountain Conference on Multigrid Methods.
9. W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Software*, 15:326–347, 1989.
10. M. Parashar and J.C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 1996.
11. P. J. Plauger, A. Stepanov, M. Lee, and D. Musser. *The Standard Template Library.* Prentice-Hall, 1996.
12. M. C. Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *Internat. J. Numer. Methods Engrg.*, 20:745–756, 1984.
13. H. Sagan. *Space-Filling Curves.* Springer, 1994.
14. L. Stals. *Parallel Multigrid On Unstructured Grids Using Adaptive Finite Element Methods.* PhD thesis, Dept. of Mathematics, Australian National University, 1995.
15. C. H. Walshaw and M. Berzins. Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. Technical Report Computer Studies 92.32, University of Leeds, 1992.
16. M. Warren and J. Salmon. A portable parallel particle program. *Comput. Phys. Comm.*, 87:266–290, 1995.
17. G. W. Zumbusch. Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme. SFB-Report 342/19/91A, TUM-I9127, TU München, 1991.