Parallel Multigrid in an Adaptive PDE Solver based on Hashing and Space-Filling Curves

Michael Griebel and Gerhard Zumbusch[a]

[a]Institut für Angewandte Mathematik, SFB 256, 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 program. The code requires substantial less amount of memory than implementations based on tree type data structures and is easier to program 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 parallelization of the code substantially and introduces a cheap way to solve the load balancing and mapping problem. We report on the main features of the method and give the results of numerical experiments with the new parallel solver on a cluster of 64 Pentium II/400MHz connected by a Myrinet in a fat tree topology.

1. INTRODUCTION

We consider a partial differential equation, e.g. an elliptic scalar differential equation on a two-dimensional domain. For reasons of efficiency, we use an optimal order solution algorithm: The equation system is discretized and then solved numerically by a multigrid method. The solution procedure can be further accelerated by means of adaptive refinement 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. Now, a way to further 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.

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. Finally, they are easy to program 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. To this end, we use an approach based on space-filling curves. This results in a fast and effective solution of the load balancing and data migration problem.

The remainder of this paper is organized as follows: In section 2 we discuss data structures for adaptive PDE solvers. Here, we suggest to use hash tables instead of the usually employed tree type data structures. Then, in section 3 we discuss the main features of the sequential adaptive multilevel solver. Section 4 deals with the partitioning and distribution of adaptive grids with space-filling curves and section 5 gives the main features of our new parallelized adaptive multilevel solver. In section 6 we present the results of numerical experiments on a parallel cluster computer with up to 64 processors. It is shown that our approach works nicely also for problems with severe singularities which result in locally refined meshes. Here, the work overhead for load balancing and data distribution remains only a small fraction of the overall work load.

2. DATA STRUCTURES FOR ADAPTIVE PDE SOLVERS

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: We start with a very coarse grid and discretize and solve the problem there. Then, an error estimator gives information where a finer grid is locally needed to better approximate the solution. Here, we add new nodes accordingly and discretize and solve the problem on the resulting grid. This procedure is repeated until a final error tolerance is matched. Then, we have obtained our solution on a fine adapted grid and stop. 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 grid refinement steps leads in a natural way to a sequence of nested grids. We can use the discrete 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. Thus, 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 [2,12,15]. Different trees represent the hierarchies of nodes, edges and elements, see also Figure 2. The components of the element-tree, the node-tree and the edge-tree from the root down to a specific level collectively represent the grid on that level. A refinement step for the actual finest grid adds a new level of leaves to the tree. Here, a single multigrid iteration step requires actually a multiple tree traversal where some floating point operations on the nodes are to be performed.

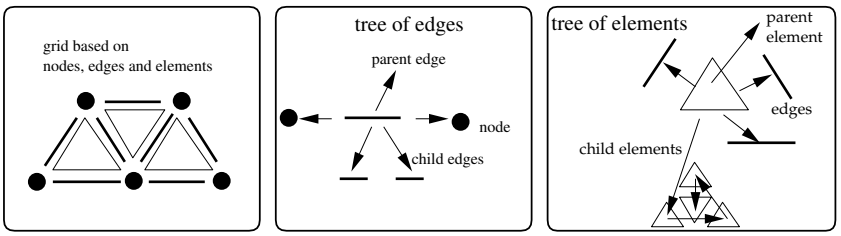

Figure 2. Storage of an adaptively refined unstructured grid.

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 [10,3]). In three dimensions, even more memory per node is needed (faces instead of edges, etc.). Consequently, more memory is required for the administration of the numerical data than for the data itself; the storage overhead is substantial.

Now, if we perform numerical operations on one grid (matrix vector multiplication, scalar products, residual computation), mostly a complete tree traversal is required. 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

To overcome the above mentioned problems we need a different way to manage adaptive grids. To this end, we propose to use hash storage techniques instead of trees. *Hash tables*

are a well established method to store and retrieve large amounts of data [9, chap. 6.4]. A hash table is implemented as a linear array of cells (buckets). The idea is to map each entity of data t to a *hash-key* $g(t)$ which is used as an address in the hash table. Now, the entity is stored and can be retrieved at that address in the hash table. The mapping is computed by a (deterministic) hash function g. Since there are many more possible different entities than different hash keys, the hash function g is not injective. Algorithms to resolve collisions are needed, see [9]. 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. 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, access to data using hash tables is cheaper in a statistical sense than random access in a sorted list or a tree. The basic principle of the hash table approach is illustrated in Figure 3.

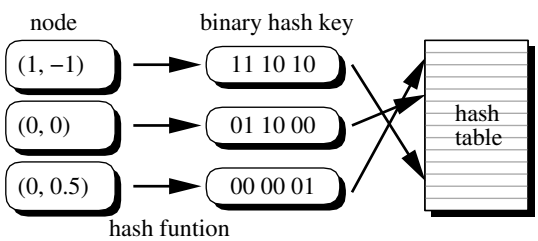

Figure 3. Storing nodes in a hash table.

We presently use the hash table implementation of an extended version of the C++ standard template library (STL) [14] by Silicon Graphics, which provides automatic resizing. There, the number of cells (always a prime) is kept roughly proportional to the number of entries and we only have to bother with a well suited hash function g. For example, for integer values the standard mapping

$$g(t) = t \bmod(\#\text{cells})$$

can be used.

An iteration over the contents of a hash table is faster than a traversal through the trees of elements or nodes. STL iterators simplify the coding even further. In a general statistical setting the cost of random access on some data in a hash table is constant, compared to logarithmic in a tree structure [9]. In total an amount of 74 bytes (double precision) of memory per unknown is used in our implementation, which is less than that needed in the above mentioned tree based codes. This is due to the lack of pointers in the hash table approach.

3. THE ADAPTIVE CODE

We describe the design of a simple PDE code to outline our concepts for the parallelization of an adaptive multigrid method. The code uses a simple two dimensional finite difference discretization of the Poisson equation and an adaptively refined grid with hanging nodes which covers the unit square. An additive multigrid solver and an adaptive grid refinement procedure are implemented. We will discuss the parallelization of the code later.

3.1. Adaptive Finite Differences

We follow a strictly node-based approach. The nodes are stored in a hash table. Each node represents one unknown. Neither elements nor edges are stored. Additionally we consider only square shaped elements. We use a one-irregular grid with 'hanging' nodes, see Figure 4, who's values are determined by interpolation. The geometric one-irregular condition is equivalent to the property that there are at most two different distances to neighbor nodes in x- and y-direction, respectively.

Figure 4. A one-irregular grid with 'hanging' nodes (left) and a grid which contains several mistakes (right) indicated by the circles.

A scalar elliptic partial differential equation is discretized by finite differences. We set up the operator as a set of difference stencils connecting one node with its neighbor nodes in the grid which can be easily determined as follows: In analogy to a standard five-point finite difference stencil, neighbor nodes are located in four directions now either at a distance h or $2h$ with a local mesh size h. No other locations are possible due to the one-irregular grid property. Other types of stencils with more nodes can be treated similarly. Thus, pure geometric information is sufficient to apply the finite difference operator to some vector.

We avoid to store the stiffness matrix or any related information. For the iterative solution of the equation system, we have to implement a matrix vector multiplication. In other words, the differential operator is applied to a given vector. For this purpose, just a loop is required over all nodes which are present in the hash table.

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 [7]. This requires an outer Krylov iterative

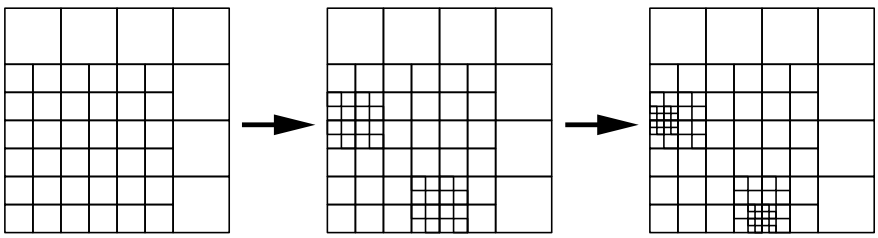

Figure 5. A sequence of adaptively refined grids.

solver. This approach results in an optimal $\mathcal{O}(1)$ condition number. Additionally, there are optimal order $\mathcal{O}(n)$ implementations available, even in the presence of degenerate grids [6]. The implementation is similar to the hierarchical basis transformation [10]. It also requires only one auxiliary vector, but it enjoys a much better condition number than the hierarchical basis method. Two loops over all nodes are necessary, one for the so-called restriction operation and one for the so-called prolongation operation which are needed in the BPX method. They can be both implemented as a loop through the nodes level by level. However, by traversing the nodes in the right way, two ordinary loops over all nodes in the hash table are sufficient, one forward and one backward. Furthermore, this additive version of multigrid is also easier parallelizable than multiplicative multigrid versions. For further details on the method, see [3].

3.3. Error Estimation and Grid Refinement

In order to create adaptive grids, we have to locate areas where the grid is to be refined. Here an error estimator or error indicator gives an error function defined on the grid. For some given threshold value, the error function is converted into a flag field, which determines whether grid refinement is required in the neighborhood of each node of the grid. In the next step, new nodes are created locally where the error flag field indicates large errors. Finally a geometric grid has to be constructed, which fulfills the additionally imposed geometric constraint (one-irregularity).

Now the question is which error indicator should 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 multiplication. The threshold value is determined by a mixture of the mean value of the local errors and some constraints on the minimum number of nodes to be created. The grid creation requires a tree traversal to check the one-irregularity condition.

4. PARTITION AND DISTRIBUTION OF ADAPTIVE GRIDS

A parallel computer version of the proposed adaptive multigrid code requires a grid partitioning strategy. The computational load and therefore the grid has to be decomposed into several partitions and each one must be 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 have to 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 for the efficiency of adaptive parallel codes. The computational load has to be equidistributed over the processors. Here, data has to be transferred between processors during computation and during the mapping of the partitions to processors. Of course, the volume of this data should be low. Furthermore the load balancing and mapping process should be cheaper than the actual computation. Often the last demand is violated. Then the load balancing step is applied less often. Consequently some load imbalance and a harder mapping problem results, since more load has to distributed in the rarer mapping steps.

In the following we suggest to use a partitioning strategy which is based on the space-filling curve approach. It is extremely cheap, simple to implement and fulfills the above mentioned complexity demand.

4.1. Space-Filling Curves

In order to parallelize a PDE solving code, we have to partition the computational work. We actually partition the set of nodes and assign the partitions to different processors. All computations related to nodes are partitioned analogously and are executed by the processor which owns the respective node. To be precise, the write operation for a node's value is performed by the owning processor, while the read operation of the value is sometimes also executed by neighboring processors by actually reading a copy of the value (owner computes)[1].

For the solution of the partition and mapping problem, we choose a computational very cheap method based on space-filling curves [5,16]. 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. If we apply the recursion scheme only a finite number of times we end up with a coarser version of the curve. Such a coarse curve passes the nodes of a given grid, e.g. of an adaptive grid, if it is fine enough. It is aligned to the grid and covers the whole domain. Because of the boundary nodes, a space-filling curve usually covers even a larger domain and it has to be clipped, see Figure 6. We assign the arc length of the curve to each node of the grid, which implies a total order relation on the nodes. Furthermore, the curve defines a (continuous) mapping from the interval $[0,1]$ to the whole domain Ω.

Throughout this paper, we consider Hilbert's space filling curve. Algorithmically, the Hilbert curve mapping of a point $t \in [0,1]$ can be expressed as follows. We assume that the number t is given in quaternary representation as $0_4.q_1q_2q_3\ldots$.

$$s(0_4.q_1q_2q_3q_4\ldots) \;=\; \mathcal{H}_{q_1} \circ \mathcal{H}_{q_2} \circ \mathcal{H}_{q_3} \circ \mathcal{H}_{q_4} \ldots \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

[1]Each variable belongs to one processor. Any write operation to the variable is performed by that processor along with all computations which lead to the value to be written: Thus "owner computes".

with affine mappings $\mathcal{H}_0$, $\mathcal{H}_1$, $\mathcal{H}_2$, $\mathcal{H}_3$ defined as

$$\begin{aligned}
\mathcal{H}_0\binom{x}{y} &= \begin{pmatrix} 0 & 1/2 \\ 1/2 & 0 \end{pmatrix}\binom{x}{y} \\
\mathcal{H}_1\binom{x}{y} &= \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}\binom{x}{y} + \binom{0}{1/2} \\
\mathcal{H}_2\binom{x}{y} &= \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}\binom{x}{y} + \binom{1/2}{1/2} \\
\mathcal{H}_3\binom{x}{y} &= \begin{pmatrix} 0 & -1/2 \\ -1/2 & 0 \end{pmatrix}\binom{x}{y} + \binom{1}{1/2}
\end{aligned}$$

The related mapping of the discrete Hilbert curve can be obtained by truncation. Given the number $t = 0_4.q_1q_2\ldots q_n$, the corresponding position on the Hilbert curve of fineness 4^n can be computed by

$$s_n(0_4.q_1q_2\ldots q_n) \;=\; \mathcal{H}_{q_1} \circ \mathcal{H}_{q_2} \circ \ldots \circ \mathcal{H}_{q_n}\begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The discrete Hilbert curve mapping s_n can easily be inverted. We will use $s_n^{-1} : \Omega \to [0,1]$ for data distribution. Note that the continuous Hilbert curve s would require further technical modifications in order to be invertble, see [16].

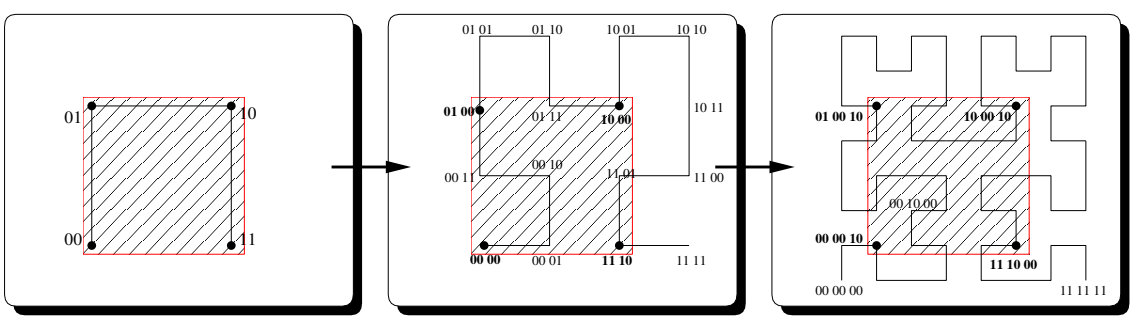

Figure 6. Hilbert's space-filling curve at different levels of resolution. It covers the whole domain, say $\Omega = [-1,1]^2$ (drawn shaded). The nodes are numbered from 0 to $4^j - 1$, in binary representation.

4.2. Data Distribution

Let the nodes be ordered increasingly by the space-filling curve, there is a simple method to construct a partition of nodes and a mapping to the 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. Then the computational load is well balanced, see [13,19].

Here, space-filling curves define a linear ordering on the set of nodes. Consequently, the nodes can be partitioned and distributed simply by a (parallel) sort algorithm. Note that the space-filling curve mapping preserves locality properties of the nodes, see [8].

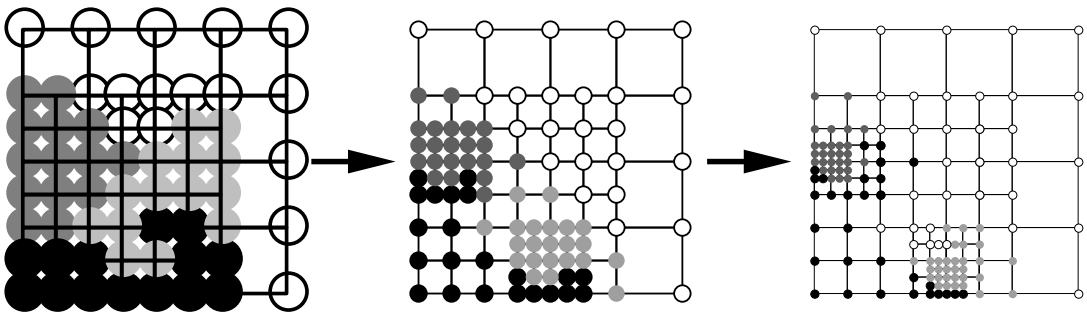

Figure 7. A Sequence of adaptively refined grids mapped onto four processors. The gray half-tones express the processor each node belongs to.

5. THE PARALLEL ADAPTIVE CODE

For the parallelization of the sequential code, all its components such as the solution of the linear system, the estimation of the resulting errors and the creation of new nodes have to be performed in parallel. Additionally the data have to be distributed to the processors. This is done in a load balancing and mapping step right after the new nodes were created. The resulting parallel algorithm is depicted in Figure 8. We consider a distributed memory MIMD parallel computer which we will program using the message passing paradigm. It is much harder to write code for a distributed memory computer with message passing than for a shared memory computer with loop-level parallelism as considered in [4,10], but usually the resulting program is more efficient for large numbers of processors. The parallelization of 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 [3,11,17,18,20]. In contrast to that, the parallelization of an adaptive code based on hash tables, which we consider here, turns out to be much easier.

5.1. Parallel Load Balancing

Since we use the space-filling curve approach, the partitioning problem reduces to a sorting problem. For the distributed memory computer implementation, a parallel sort algorithm with distributed input and output is required. Here, we employ a one-stage *radix sort* algorithm [9, chap. 5.2.5], which is similar to a one-stage *bucket sort*. Moreover we assume that the previous data distribution still guarantees good load-balancing for the parallel sort. During the adaptive solution of a PDE, the load balancing is used only incrementally. Hence the old partition is close to the new one. The set of nodes is already pre-sorted and serves as a good initial guess for the sort algorithm.

The result is a new partition of the grid, which is stored two-fold. First of all, the nodes have been transferred to the processor which owns them. Second, the partition itself, described by the p sub-intervals of the unit interval $[0, 1]$ is stored on each processor.

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 8. The parallel adaptive algorithm.

Given these $p-1$ numbers, it is possible to compute the owner of each node analytically and on all processors. Hence it is possible to find out, which processor actually takes care for a specific node.

The load is partitioned exactly, while the volume of communication depends on the boundaries of the partitions. These boundaries may sometimes be kinky 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 the assignment of the node to a processor. It is also used as the address of the node in the local hash table of the assigned processor, see Figure 9. 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. Thus, just by inspection of the index, it is easy to determine the processor the node belongs to.

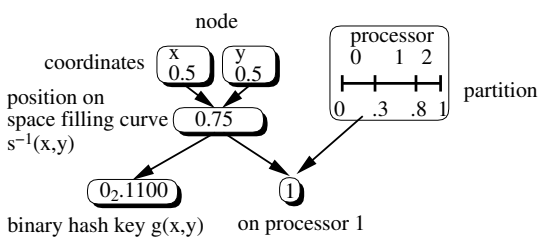

Figure 9. For a given node (x, y), the owner processor is determined by the space-filling curve and the partition. The hash key $g(x, y)$ of the node may also be generated by the space-filling curve for reasons of data locality. However, many other heuristic hash functions g will also do the job.

5.2. Parallel Krylov Iteration

The parallel iterative Krylov solver consists of several components. It requires matrix multiplications, scalar products and the application of the preconditioner, i.e. the additive multigrid method in our case. Here, the scalar product can be implemented by means of ordinary data *reduction*[2] operations offered by any message passing library. Furthermore, using the "owner computes" paradigm, the parallel 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 one local nearest neighbor communication is sufficient.

5.3. Parallel Multigrid Preconditioning

The communication pattern of the additive multigrid is more expensive than that of the matrix multiplication: First, 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 global communication steps in which values are collected and distributed again. Now the restricted values are present on all nodes and ghost nodes. Finally, the adjoint process of prolongation can take place as local operations again. Thus the result is valid on all nodes without the ghost nodes.

The local multigrid restriction and prolongation operations are organized as ordinary restriction and prolongation, just applied to the local nodes and ghost nodes on a processor. They can be implemented as a forward and a backward loops level by level over all nodes in the hash table. The ghost nodes of the multigrid algorithm are determined as union of the ghost nodes of the difference stencils 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 only local. 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 [3,20]. 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). Altogether, the additive multigrid is more efficient for parallel computers with higher communication latencies, whereas the number of communication steps of the multiplicative multigrid is less important for low latency computers.

5.4. Adaptive Refinement

The parallel estimation of errors is structurally very similar to a parallel matrix vector multiply with a nearest neighbor communication pattern. Nearest neighbor communication is also only required from the algorithms which flag nodes for refinement according to a global threshold parameter and consequently create new nodes. The threshold parameter itself can be computed in parallel with a global reduction operation.

[2] A reduction operation is defined as a global operation. Each processor contributes one argument. The final result, e.g. the global sum or global maximum of all values, is distributed to all processors.

The tricky part of parallel adaptive refinement is the enforcement of the one-irregular grid condition, see section 3.1. Additional nodes must be created such that the refined grid fulfills the geometric constraints. However, a local grid refinement in one area theoretically may spread out to the whole domain and may result in a global refinement. Hence the parallel grid refinement step may degenerate to an expensive procedure with a global communication pattern [3].

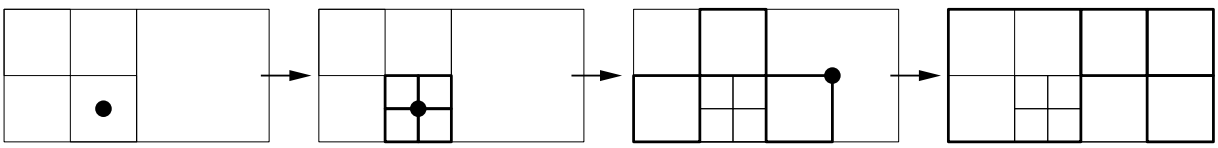

Figure 10. Stages of a square-based one-irregular grid closure algorithm. Squares under consideration are drawn bold.

However, in standard cases we found the following procedure to terminate quickly after few cycles. A new node implies the presence of some adjacent squares and some other, coarser squares, see Figure 10. If the vertices of these squares are present in the current grid, the grid does fulfill the one-irregularity condition locally. If the vertices are not present, they have to be created. One cycle of our procedure starts with a list of nodes to be created. The appropriate squares are checked and the missing vertices are inserted into a new list of nodes to be created. The procedure iterates as long as the list of nodes is not empty. It can easily be parallelized with nearest neighbor communication for the exchange of lists and with a global reduction operation for the termination criterion.

6. NUMERICAL EXPERIMENTS

We consider the two dimensional Poisson equation as a model problem

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

and run our adaptive multilevel finite difference code to solve it. The solution possesses a steep gradient at the lower left corner of the domain Ω. All numbers reported here are CPU times for the solution of the equation systems measured on a cluster computer named 'Parnass2' [1] at our department. It consists of 64 Pentium II processors/400 MHz arranged as 32 dual processor computers, each equipped with 256 MB main memory. The connection network is build from Myrinet switches and components in form of a fat-tree topology. It allows for 850 Mbit/sec point to point transfer rates measured under MPI between different computing nodes and 1.1 Gbit/sec between processes sharing the same

[3] Another remedy proposed by some authors [3] is to weaken the geometric refinement constraints in the neighborhood of the partition boundaries.

computing node. The network has a bisection bandwidth of 41 Gbit/sec which guarantees a blocking free delivery of all messages.

In our experiments, we choose the simple Poisson equation as model problem. Here, the ratio of computational work to communication is low compared to more complicated equations like for example the Navier-Stokes equations. Thus, the simple Poisson problem is a very hard test problem when it comes to 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 sequence of regular grids of different levels without adaptive refinement. We use a nested iteration ('full' multigrid). The solution obtained on level l is used as an initial guess for the iteration on the next finer level $l+1$. Together with the bounded condition number of the multigrid preconditioner, this results in a constant number of iterations for a solution up to discretization error.

time		processors						
level	nodes	1	2	4	8	16	32	64
5	1089	3.30	2.18	1.32	0.94	0.69	0.52	0.43
6	4225	16.0	8.52	4.70	2.88	1.74	1.16	0.79
7	16641	66.9	33.6	17.6	9.87	5.40	3.17	1.88
8	66049	283	141	70.4	36.2	19.0	10.4	5.64
9	263169	1160	583	291	147	73.8	37.5	19.4
10	1050625			1162	584	294	147	74.0
11	4198401					1162	585	292

Table 1
Uniform refinement example, timing, levels 5 to 11, 1 to 64 processors, Parnass2.

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. If we increase the number of processor, the computing times decay. However, the 64 processor perform most efficiently for sufficiently large problems. We observe perfect scalability of the algorithm.

6.2. Adaptive Example

In the next test we consider adaptively refined grids. The grids are refined towards the corner $(0,0)$, see Figure 11. 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 assume a constant number of iterations within a nested iteration.

We obtain a scaling of about a factor 2 from one level to the next finer level, which means 2 times more unknowns on the next finer level. Increasing the number of processors speeds up the computation accordingly. Again we observe scalability of the algorithm. In order to use 64 processor efficiently, the grid has to be fine enough, i.e it has to have

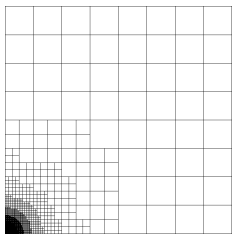

Figure 11. An adaptively refined grid.

time	processors						
nodes	1	2	4	8	16	32	64
384	1.27	0.85	0.69	0.51	0.42	0.37	0.35
682	2.38	1.48	1.04	0.75	0.57	0.45	0.41
1243	4.54	2.81	1.81	1.21	0.83	0.60	0.51
2320	8.75	4.92	3.13	1.95	1.25	0.86	0.62
4391	17.0	9.30	5.19	3.26	1.89	1.25	0.85
8460	33.5	17.8	10.1	5.57	3.27	1.92	1.26
16469	66.9	34.4	18.1	10.1	5.50	3.21	1.99
32291	133	67.7	35.4	19.3	10.3	5.50	3.27
63736	263	134	68.5	36.6	19.2	10.5	5.66
126271	529	272	139	70.4	36.7	19.1	10.3
250911		560	278	143	71.8	36.8	19.1

Table 2
Adaptive refinement example 2D, timing, 1 to 64 processors, Parnass2.

more than 10^5 unknowns.

6.3. Load Balancing

Now we compare the time for solution of the equation system with the time required for load balancing, i.e. the process of sorting the nodes and mapping them to processors. The ratio α 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.

In the single processor case, no load balancing is needed. Thus the ratio of the sort time versus the solve time 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 we note that load balancing still is much cheaper than the solution of the equation systems. However, higher number of processors make the mapping relatively slower.

In the case of uniform refinement, there are only few nodes located at processor bound-

α		processors						
	nodes	1	2	4	8	16	32	64
uniform 2D	66049	0	9.7e-4	1.1e-3	1.3e-3	1.3e-3	3.2e-3	4.9e-3
adaptive 2D	63736	0	6.8e-4	7.9e-4	9.1e-4	1.1e-3	2.9e-3	3.5e-3

Table 3
Ratio sorting nodes to solving the equation system, 1 to 64 processors, Parnass2.

aries which may have to be moved during the mapping. Hence our load balancing technique is very cheap in this case. The mapping of data for adaptively refined grids requires the movement of a larger amount of data, even if most of the nodes stay on their processors, see also [3].

However, all load balancing methods have to face the problem of a larger amount of data to be moved, especially for larger processor numbers. 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 with an adaptive multigrid method. Hash tables lead to a substantial reduction of memory requirements for the storage of the hierarchy of adaptive refined grids compared to standard tree based implementations. Furthermore, the implementation of an adaptive code based on hash tables proved to be simpler than its tree counterpart. Both properties, i.e. low amount of memory and the simple implementation carried over to the parallelization of the code. Here space-filling curves were used for data partitioning and for providing a proper hash function at the same time. Altogether we obtain a parallel adaptive multilevel solver for elliptic PDEs which is simple to implement.

The results of our numerical experiments showed that load balancing based on space-filling curves is indeed cheap compared to other more complicated 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 contrast to other procedures that used fewer load balancing steps and operate on accumulated load imbalances, often in connection with more expensive load balancing mechanisms.

Note finally that load balancing for adaptive multilevel solvers with space filling curves can be obtained (after slight modifications) in an analogous way and with analogous results also for the case of general unstructured grids. This is actual work in progress.

REFERENCES

1. Parnass2: A cluster of PCs. http://wwwwissrech.iam.uni-bonn.de/ research/projects/parnass2/, 1998.

2. R. E. Bank and T. F. Dupont. An optimal order process for solving elliptic finite element equations. *Math. Comp.*, 36:967–975, 1981.
3. P. Bastian. *Parallele Adaptive Mehrgitterverfahren.* B. G. Teubner, Stuttgart, 1996.
4. K. Birken. Ein Parallelisierungskonzept für adaptive, numerische Berechnungen. Master's thesis, Universität Erlangen-Nürnberg, 1993.
5. S. H. Bokhari, T. W. Crockett, and D. N. Nicol. Parametric binary dissection. Technical Report 93-39, ICASE, 1993.
6. F. A. Bornemann. An adaptive multilevel approach to parabolic equations III. *IMPACT Comput. Sci. Engrg.*, 4:1–45, 1992.
7. J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.
8. M. Griebel and G. Zumbusch. Parallel adaptive subspace correction schemes with applications to elasticity. *CMAME*, 1999. submitted.
9. D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.
10. P. Leinen. *Ein schneller adaptiver Löser für elliptische Randwertprobleme auf Seriell– und Parallelrechnern.* PhD thesis, Universität Dortmund, 1990.
11. 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.
12. W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Software*, 15:326–347, 1989.
13. 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.
14. P. J. Plauger, A. Stepanov, M. Lee, and D. Musser. *The Standard Template Library.* Prentice-Hall, 1996.
15. M. C. Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *Internat. J. Numer. Methods Engrg.*, 20:745–756, 1984.
16. H. Sagan. *Space-Filling Curves.* Springer, 1994.
17. L. Stals. *Parallel Multigrid On Unstructured Grids Using Adaptive Finite Element Methods.* PhD thesis, Department of Mathematics, Australian National University, 1995.
18. 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.
19. M. Warren and J. Salmon. A portable parallel particle program. *Comput. Phys. Comm.*, 87:266–290, 1995.
20. G. W. Zumbusch. Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme. SFB-Report 342/19/91A, TUM-I9127, TU München, 1991.