Contemporary Mathematics
Volume **218**, 1998

Hash-Storage Techniques for Adaptive Multilevel Solvers and Their Domain Decomposition Parallelization

Michael Griebel and Gerhard Zumbusch

1. Introduction

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

We study a simple model problem, an elliptic scalar differential equation on a two-dimensional domain. A finite difference discretization of the problem leads to a linear equation system, which is solved efficiently by a multigrid method. The underlying grid is adapted in an iterative refinement procedure. Furthermore, we run the code on a parallel computer. In the overall approach we then 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 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 moderate constant if a statistical data distribution is assumed. Hash tables allow to deal with locally adapted data in a simple way. Furthermore, data decomposition techniques based on space-filling curves provide a simple and efficient way to partition the data and to balance the computational load.

We demonstrate the concepts of hash-storage and space-filling curves by a simple example code, using a square shaped two-dimensional domain and finite difference discretization of the Laplacian. The concepts can also be applied to more complicated domains, equations and grids. A finite element discretization on an unstructured tetrahedral grid for example requires more data, more complicated data structures and more lines of code. However, the concepts presented in this article remain attractive even for such a code.

1991 *Mathematics Subject Classification.* Primary 65Y05, 65N55; Secondary 65N06, 65N50.

©1998 American Mathematical Society

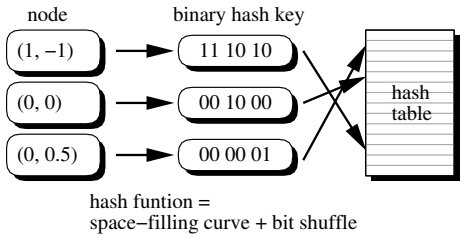

FIGURE 1. Storing nodes in a hash table.

2. Hash Addressing

2.1. Hash-Storage. Looking for a different way to manage adaptive grids than tree data structures, we propose to use hash storage techniques. *Hash tables* are a well established method to store and retrieve large amounts of data, see f.e. [**8**, chap. 6.4]. They are heavily used in database systems, computer language interpreters such as 'Perl' and the Unix 'C shell' and in compilers. We propose to use hash table for numerics.

The idea of hashing is to map each entity of data to a *hash-key* by a hash-function. The hash-key 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 vector of cells (buckets) as illustrated in Figure 1. Since there are many more possible different entities than different hash-keys, the hash function cannot be injective. Algorithms to resolve collisions are needed. Furthermore, some buckets in the hash table may be left empty, because no present entity is mapped to that key. We use space-filling curves as hash functions, see Chapter 3.

In general, access to a specific entry in the hash table can be performed in constant time, which is cheaper than random access in a sorted list or a tree. However, this is only true if the hash function scatters the entries broad enough and there are enough different cells in the hash table.

The hash table code does not need additional storage overhead for logical connectivities like tree-type data structures which are usually used in adaptive finite element codes, citeLeinen. Furthermore, and this is an additional advantage of the hash table methodology, it allows relatively easy coding and parallelization with simple load balancing.

2.2. Finite Difference Discretization. We take a strictly node-based approach. The nodes are stored in a hash table. Each interior node represents one unknown. Neither elements nor edges are stored. We use a one-irregular grid with 'hanging' nodes, see Figure 2, whose values are determined by interpolation. This is equivalent to the property that there is at most one 'hanging' node per edge. The one-irregular condition is a kind of a geometric smoothness condition for the adaptive grid. Additionally we consider only square shaped elements.

The partial differential equation is discretized 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, its neighbors can be only on a limited number of level, or one level up or down. The distance to the neighbor is determined by the level they share.

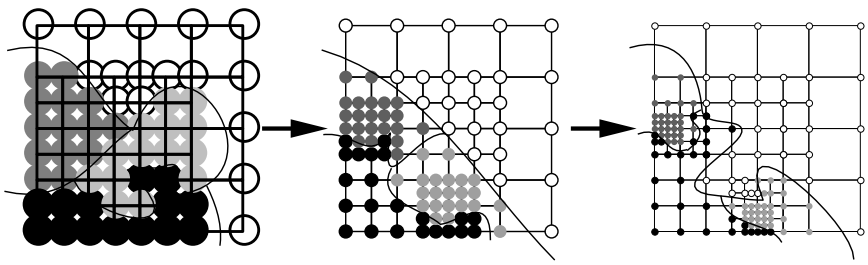

FIGURE 2. A Sequence of adaptively refined grids mapped onto four processors.

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.

2.3. Multilevel Preconditioner. We use an additive version of the multigrid method for the solution of the equation system, i.e. the so called BPX preconditioner [**5**].

$$Bu \;=\; \sum_{\text{level } j} \sum_{i} 4^{-j}(u, \phi_i^j)\phi_i^j$$

This requires an outer Krylov iterative solver. The BPX preconditioner has the advantage of an optimal $\mathcal{O}(1)$ condition number and an implementation of order $\mathcal{O}(n)$, which is optimal, even in the presence of degenerate grids. Furthermore, this additive version of multigrid is also easier to parallelize than multiplicative multigrid versions.

The straightforward implementation is similar to the implementation of a multigrid V-cycle. However, the implementation with optimal order is similar to the hierarchical basis transformation and requires one auxiliary vector. 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 iterating over the nodes in the right order, two ordinary loops over all nodes in the hash table are sufficient, one forward and one backward.

2.4. Adaptive 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 estimated error is converted into a flag field, determining whether grid refinement is required in the neighborhood. Then, 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.

3. Space-Filling Curves

3.1. Space-Filling Curve Enumeration. A domain or data decomposition is needed for the parallelization of the code. Usually domain decomposition and mapping strategies are difficult and expensive. Adaptive grids require such an

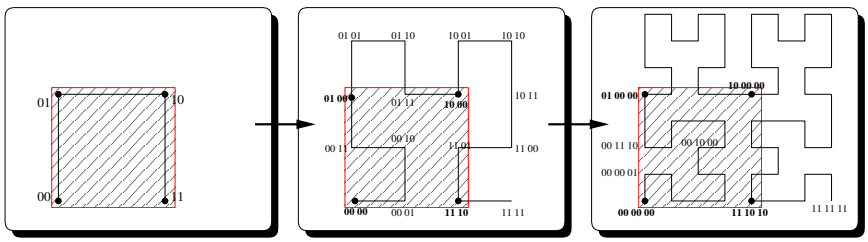

FIGURE 3. Hilbert's space-filling curve at different levels of resolution. It covers the whole domain, say $\Omega = [-1,1]^2$ (drawn shaded). The nodes are numbered binary from 0 to $4^j - 1$. Additionally, the numbers are used for the hash-keys.

expensive decomposition several times. Hence we are interested in cheap decomposition heuristics.

We choose a computational very cheap method based on space-filling curves [**14**]. A space-filling curve is defined recursively by substituting a straight line segment by a certain pattern of lines, similar to fractals. This recursion is applied infinitely times and results in a curve, which fills the whole domain. The curve defines a (continuous) mapping from an interval to the whole domain via a scaled arc-length. Space-filling curves are often used for theoretical purposes, e.g. for complexity bounds. Furthermore, they have been employed in combinatorial optimization [**1**], in computer graphics [**16**], in operating systems and routing, and in parallel computing as a tool for domain partitioning [**4, 12**].

We use space-filling curves as a way to enumerate and order nodes in the computational domain. One can think of such a space-filling curve as passing all nodes of a given grid, e.g. an adaptive grid. Because of the boundary nodes, we choose a curve which covers a larger domain than the computational domain, see Figure 3. We assign the scaled arc length of the curve to each node of the grid, called index. The indices imply a total order relation on the nodes. The space-filling curve is never constructed explicitly, but it is used for the computation of the indices. The indices are used for the construction of hash-keys.

3.2. Space-Filling Curve Partition. Given an ordered list of nodes induced by a space-filling curve, we construct a static partition of the grid points and data in the following way: We cut the list into p equally sized intervals and map them according to this order to processors with increasing numbers. The partition is defined by its $p-1$ cuts.

The computational load is balanced exactly, see [**18, 13**]. The volume of communication depends on the boundaries of the partitions.

4. Parallel Code

For the parallelization of the sequential code, all its components such as the solution of the linear system, the estimation of errors and the creation of 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 nodes, a step which was not present in the sequential version or for uniform refinement [**7**]. We consider a distributed memory, MIMD, message passing paradigm. This makes

the parallelization more involved than it would be on a shared memory computer as in [**9, 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 [**19, 17, 2, 15, 10**]. However, the parallelization of an adaptive code based on hash tables, which we consider here, will turn out to be much easier.

4.1. Partition in Parallel. Using the space filling curve, the partitioning problem reduces to a sorting problem. This requires a parallel sort algorithm with distributed input and output. We employ a one-stage *radix sort* algorithm, see [**8**, chap. 5.2.5]. Here we can make use of the assumption that the previous data decomposition still guarantees good load-balancing for the parallel sort.

The result is a new partition of the grid. In total, the space-filling curve load balancing is very cheap, because most of the data has been sorted in a previous step. It 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 the node to a processor and additionally for addressing the node in the local hash table of the processor. In case that a copy of a node (a ghost node) is required on another processor, the index is also used for addressing the copy in the hash table of this processor. Comparing the index of a node to the $p-1$ partition cut values, it is easy to determine the processor the node originally belongs to.

4.2. Finite Differences in Parallel. The parallel iterative solution consists of several components. The Krylov iterative solver requires matrix multiplications, scalar products and the application of the BPX-preconditioner in our case. The scalar product can be implemented as ordinary data reduction operations [**11**] offered by any message passing library. Any modification of a node's value is performed by the processor who owns the node, "owner computes". This implies a rule of how the computational work is partitioned to the processors.

The matrix multiplication requires the update of auxiliary (ghost) values located at the boundary of the partition, see [**6**]. 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 necessary.

4.3. Multilevel Preconditioner in Parallel. The communication pattern of the BPX-preconditioner is more dense than pattern for the matrix multiplication: The local restriction operations can be performed in parallel without any communication. The resulting values have to be reduced [**11**] and distributed.

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 between all-to-all and a pure local pattern.

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

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

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.

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 [**19**]. 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 the additive multigrid has an advantage for computers with high communication latency, while for computers with low latency the number of communication steps is less important.

5. Experiments

We consider the two dimensional Poisson equation $-\Delta u = 0$ on $\Omega = [-1,1]^2$ with Dirichlet boundary conditions $u = 0$ on $\partial\Omega \setminus [-1,0)^2$ and $u = 1$ on the remainder of $\partial\Omega$. We run our adaptive multilevel finite difference code to solve it. The solution possesses 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 MPICH on a fast ethernet network.

5.1. 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 using different numbers of processors.

We observe a scaling of a factor of 4 from one level to the next finer level which corresponds to the factor of 4 increase in the amount of unknowns on that level. The computing times decay and a scale-up can be seen. However, the 8 processor perform efficiently only for sufficiently large problems, i.e. for problems with more than 8 levels.

5.2. 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 numbers 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. This is due to an increase of the amount of nodes by a factor of 4, because the grid has been adapted already towards the singularities on previous levels. Increasing the number of processors speeds up the computation accordingly, at least for two and four processors. In order to use seven processor efficiently, the grid has to be fine enough, i.e it has to have more than 8 levels.

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

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 3. Ratio sorting nodes (partitioning and mapping) to solving the equation system (multilevel), level 8, 1 to 8 processors.

sort time / solve time	processors			
	1	2	4	7/8
uniform	0	0.0028	0.0079	0.0141
adaptive	0	0.0066	0.0149	0.2367

5.3. 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 using different numbers of 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 because of the overall amount of data, even if most of the nodes stay on the processor. Other load balancing strategies can be quite expensive for adaptive refinement procedures, see [**2**].

6. 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. 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. J. Bartholdi and L. K. Platzman, *Heuristics based on space-filling curves for combinatorial optimization problems.*, Management Science **34** (1988), 291–305.
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*, Tech. Report 93-39, ICASE, 1993.
5. J. H. Bramble, J. E. Pasciak, and J. Xu, *Parallel multilevel preconditioners*, Math. Comp. **55** (1990), 1–22.
6. G. C. Fox, *Matrix operations on the homogeneous machine.*, Tech. Report C3P-005, Caltech, 1982.
7. M. Griebel, *Parallel domain-oriented multilevel methods*, SIAM Journal on Scientific Computing **16** (1995), no. 5, 1105–1125.
8. D. E. Knuth, *The art of computer programming*, vol. 3, Addison-Wesley, 1973.
9. P. Leinen, *Ein schneller adaptiver Löser für elliptische Randwertprobleme auf Seriell– und Parallelrechnern*, Ph.D. thesis, Universität Dortmund, 1990.
10. 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.
11. MPI Forum, *MPI: A message-passing interface standard*, University of Tennessee, Knoxville, Tennessee, 1.1 ed., 1995.
12. J. T. Oden, A. Patra, and Y. Feng, *Domain decomposition for adaptive hp finite element methods*, Proceedings of Domain Decomposition 7, Contemporary Mathematics, vol. 180, AMS, 1994, pp. 295–301.
13. M. Parashar and J.C. Browne, *On partitioning dynamic adaptive grid hierarchies*, Proceedings of the 29th Annual Hawaii International Conference on System Sciences, 1996.
14. H. Sagan, *Space-filling curves*, Springer, 1994.
15. L. Stals, *Parallel multigrid on unstructured grids using adaptive finite element methods*, Ph.D. thesis, Department of Mathematics, Australian National University, 1995.
16. D. Voorhies, *Space-filling curves and a measure of coherence*, Graphics Gems II (J. Arvo, ed.), Academic Press, 1994, pp. 26–30.
17. C. H. Walshaw and M. Berzins, *Dynamic load-balancing for PDE solvers on adaptive unstructured meshes*, Tech. Report Computer Studies 92.32, University of Leeds, 1992.
18. M. Warren and J. Salmon, *A portable parallel particle program*, Comput. Phys. Comm. **87** (1995), 266–290.
19. G. W. Zumbusch, *Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme*, SFB-Report 342/19/91A, TUM-I9127, TU München, 1991.

Institut für Angewandte Mathematik, University Bonn, Wegelerstr. 6, 53115 Bonn, Germany

E-mail address: griebel@iam.uni-bonn.de

Institut für Angewandte Mathematik, University Bonn, Wegelerstr. 6, 53115 Bonn, Germany

E-mail address: zumbusch@iam.uni-bonn.de