A PARALLEL ADAPTIVE MULTIGRID METHOD

G. Zumbusch
Institut für Angewandte Mathematik
Abteilung Wissenschaftliches Rechnen
und Numerische Simulation
Wegelerstraße 6
53115 Bonn

SUMMARY

A parallel version of an adaptive multigrid solver for elliptic partial differential equations is described. It operates on a finite difference discretization on quad-tree and oct-tree meshes, which are obtained by adaptive mesh refinement. A fast parallel load balancing strategy for the parallel multigrid equation solver is proposed that is defined by a space-filling Hilbert curve and is applicable to arbitrary shaped domains. Some numerical experiments demonstrate the parallel efficiency and scalability of the approach.

AN ADAPTIVE MULTIGRID SOLVER

Our goal is to solve an elliptic partial differential equation as fast as possible. We consider a multigrid preconditioner, adaptive grid refinement and their efficient parallelization. We have to develop a parallel multigrid code that is almost identical to the sequential implementation. The computational workload has to be distributed into similar sized partitions and, at the same time, the communication between the processors has to be small. The underlying computer model takes into account the local processor execution time and the communication time. The first term is proportional to the number of operations and the second one depends on the amount of data to be transferred between processors.

The PDE is discretized by finite differences on a 1-irregular quad-tree or oct-tree grid. 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 levels, or one level up or down. The distance to the neighbor is determined by the refinement level they share.

So pure geometric information is sufficient to apply the finite difference operator to some vector. Hence 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.

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 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 are sufficient, one forward and one backward.

A parallel version of multiplicative multigrid usually is based on a partition of all nested grids. The domain Ω is decomposed into several sub-domains Ω_j, which induces partitions of all grids. Each processor holds a fraction of each grid in such a way that these fractions of each grid form a nested sequence. Hence each operation on a specific level is partitioned and mapped to all processors. Furthermore the communication during grid transfer operations is small because of nested sequences on a processor. This means that one has to treat global problems on each level, which are partitioned to all processors. The intra-grid communication has to be small, that is the number of nodes on the boundary of the partition should be small. Furthermore the amount of work on coarse grid levels usually is small and each processor does not compute much. There are several strategies to deal with the coarse grid problem in general, such as to centralize the computation on a master processor, to perform identical computations on all processors or to modify the coarse grid correction step.

A static partition of the domain into strips or squares can be used for uniform grids and has been used for the first parallel multigrid implementations [14, 8], see also the survey [18]. In contrast to the geometry oriented parallelization of multiplicative multigrid methods, the additive multigrid version or additive multilevel preconditioners can be parallelized in a more flexible way. The overall workload has to be partitioned, but we do not have to consider individual levels. Here, also the communication takes place in a single step for all nodes, which are located on the boundary of at least one grid of the nested sequence. The multilevel BPX preconditioner for a uniform grid has been parallelized in [4, 31]. These approaches can easily be generalized to block-structured grids.

THE LOAD-BALANCING PROBLEM

Parallel multigrid methods on a sequence of adaptively refined grids require some grid partitioning algorithms. The grid partition problem can be equivalently formulated as a graph partitioning problem. However, the general graph partitioning problem is NP-hard. Even the problem to find asymptotically optimal partitions for unstructured grids is NP-hard [9]. Several heuristic algorithms have been developed in the area of parallel computing: There are bisection algorithms based on the coordinates of nodes and elements and there are many algorithms based on the graph of the stiffness matrix, such as recursive spectral bisection, and multilevel versions of other heuristics. Some PDE codes also use data diffusion [28] or some specific heuristics [3, 5, 2]. For a survey on grid partitioning methods we refer to [24].

Graph partitioning can be expensive. However, in the framework of adaptive, element-wise refinement, partitions and mappings have to be computed quickly and during run-time. On a shared memory parallel computer, the serial representation of the grid hierarchy in memory and coloring of the elements along with dynamic scheduling of lists of elements can be used. This has been proposed for a code based on triangles and the additive hierarchical bases preconditioner [17].

On a distributed memory computer however, the grid hierarchy has to be partitioned and maintained, which requires a substantial amount of bookkeeping. In [10] a multiplicative multigrid method or a finite volume discretization on a grid with quadrilaterals and hanging nodes is proposed. The elements are partitioned by a hierarchical recursive coordinate bisection. Test indicated that the repartitioning of coarser levels, when new elements were created, did not pay off. Adaptive conforming grids consisting of triangles, which are refined by a bisection strategy, were employed in the parallel multigrid codes of [2, 28, 19]. Here, different repartitioning strategies for refined grids were used.

The key point of any dynamic data partition method is efficiency. We look for a cheap, linear time heuristic for the solution of the partition problem. Furthermore the heuristic should parallelize well. This is why we look for even cheaper partition methods. They are provided by the concept of *space-filling curves.*

First we have to define space-filling curves. A curve is space-filling if and only if the image of the mapping

$$f \;:\; [0,1] =: I \;\mapsto Q := [0,1]^2, \qquad f \text{ continuous and surjective}$$

does have a classical positive d-dimensional measure. One of the oldest and most prominent space-filling curve, the Hilbert curve can be defined geometrically [16], see also [26]. If the interval I can be mapped to Q by a space-filling curve, then this must be true also for the mapping of four quarters of I to the four quadrants of Q, see Figure 1. Iterating this sub-division process, while maintaining the neighborhood relationships between the intervals leads to the Hilbert curve in the limit case. The mapping is defined by the recursive sub-division of the interval I and the square Q.

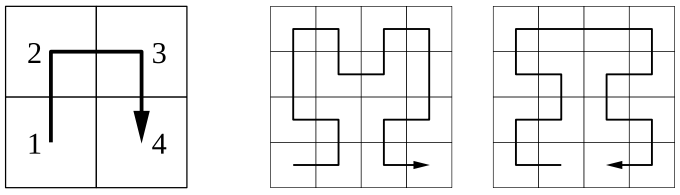

Figure 1: Construction of a Hilbert space-filling curve. The open and the closed curve.

Often some curves as intermediate results of iterative construction procedure are more interesting than the final Hilbert curve itself. The construction begins with a generator, which defines the order in which the four quadrants are visited. The generator is applied in every quadrant and their sub-quadrants. By afine mappings and connections between the loose ends of the pieces of curves, the Hilbert curve is obtained.

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 $0_4.q_1q_2q_3\ldots$.

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

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

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

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 invertible, see [26].

There are basically two different Hilbert curves for the square, modulo symmetries, see Figure 1. An open and a closed space-filling curve can be constructed by the Hilbert curve generator maintaining both neighborhood relations and the sub-division procedure. The resulting curve is indeed continuous and space-filling. Although each curve of the iterative construction is injective and does not cross itself, the resulting Hilbert curve is not injective. This can be demonstrated easily by looking at the point $(1/2, 1/2) \in Q$, which is contained in the image of all four quadrants. Hence several points on I are mapped to this point of Q. In three dimensions, there are 1536 versions of Hilbert curves for the unit cube [1].

How can space-filling curves contribute to the efficient parallelization of the introduced applications? The basic idea is to map the nodes or entities in space to points on one iterate of a space-filling curve. The points can be mapped to the unit interval by the inverse mapping of the space-filling. The points, which now lay on the unit interval, can be sorted. The points can be partitioned and the sets of points can be mapped to processors, which defines a partition of the original problem in space. The partition of the volume $\mathbb{R}^d$ induced by the space-filling curve still gives perfect load-balance. However, the separators usually are larger than the optimal separators. As a technical detail, we consider the partition and mapping of nodes of the grid and we choose a space-filling curve which is aligned to the grids. Hence a sufficiently fine, finite representation of the space-filling curve contains all nodes and covers a larger domain than the domain of interest Ω. Partitioning by space-filling curves has been employed for finite element computations in [21, 20] and has been compared to other heuristics in [23]. The main advantage of space-filling curves in this context is their simplicity.

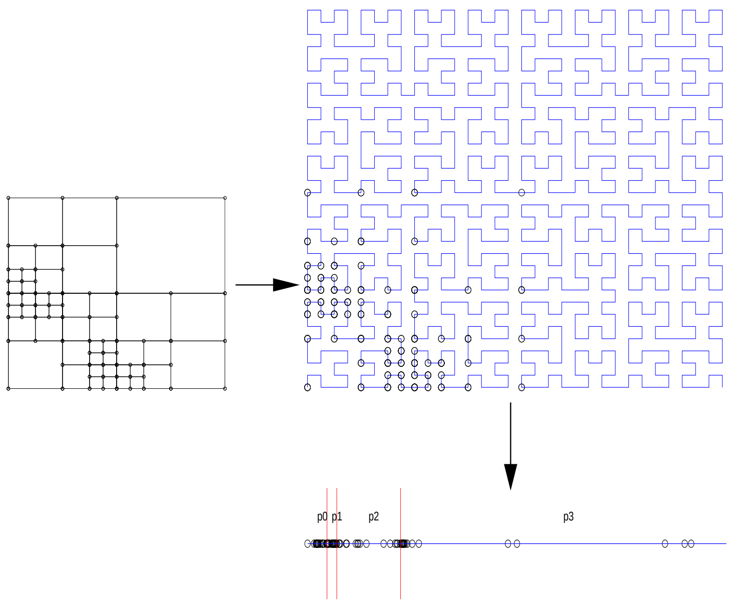

Figure 2: Mapping a 2D adapted grid to a space-filling curve (left) and mapping points on a space filling curve to a parallel processor (right).

We have to compute the separators such that the partitions contains the same number of nodes. This can be accomplished on a list of nodes sorted by their position on the space-filling curve $f^{-1}(x_i)$. The list is cut into equal-sized pieces and each piece is mapped to one processor. Hence the partition of nodes can be done with an ordinary sorting algorithm. Given a set of nodes x_i with their keys $f^{-1}(x_i)$, we first create a sorted list of keys. However, we need to perform the partition algorithm on a parallel computer. Given a set of nodes with their keys, which are distributed over the processors, we look for a parallel sort algorithm, which results in a partition where each partition resides on the appropriate processor. There is no need to gather all keys on a master processor, but a pure scalable parallel algorithm performs better with respect to communication volume, memory usage and scalability. Currently we employ single step radix sort, where the previous separators serve as an initial guess for the sorting. For uniform grid refinement f.e., two steps of local neighbor communication are required only.

KEY-BASED ADDRESSING

Instead of linked lists or trees, we propose to use *hash storage* techniques. First we describe a *key based* addressing scheme. The entity (a node) is stored in an abstract vector, where it can be retrieved by its key. Furthermore it is possible to decide, whether a given key is stored in the table or not, and it is possible to loop over all keys stored in the vector. In order to reduce the amount of storage of the grid, we omit any pointers and use keys instead. For a (hyper-) cube shaped domain $\Omega = [0,1]^d$, we can use the coordinates of a node for addressing purposes. The coordinates (and the keys) of hierarchical son nodes and father nodes can be computed from the node's coordinates easily. The computation of neighbor nodes requires special care, because it is not immediately clear, where to look for the node. Given a one-irregular grid with hanging nodes, for example, a neighbor node can be located in the distance of h or $2h$ from the node with a local step-size h. In the worst case this results in two vector lookup operations, one in distance h along a coordinate direction and, if it was unsuccessful, one lookup in distance $2h$, see [12]. Similar key based addressing schemes can be obtained for other grid refinement procedures and for different domains, see [28, 25].

Key based addressing does simplify the implementation of a sequential, adaptive code. Now, we generalize the concept of key addressing and hash tables to the parallel case. The idea is to store the data in a hash table located on the local processor. However, we use global keys, so a ghost copy of the node may also reside in the hash table of a neighbor processor. Furthermore we base the code on space-filling curve partitions of the previous section. The position of a node on the space-filling curve, along with the known partition, defines the home processor of a node. Given a node on a processor, it is easy to determine to which processor the node belongs to. If a node occurs, which does not belong to the processor, it must be a ghost copy, and it is computable where to find its original.

The next idea is to combine the position on the space-filling curve with the hash key [29, 30, 22]. The computation of the position on the curve can be computed for any given coordinate tuple. It is a unique mapping $[0,1]^d \to [0,1]$ similar to mapping required for hash keys. The position can be used as a key. Furthermore, for the construction of the hash table, we need a hash function. This can be any mapping $[0,1] \to [0,m]$ with a large integer number m, preferably prime. Many cheap functions related to pseudo-random numbers will do here. Modifications of the hash function can improve the cash performance of the code: Space-filling curves introduce locality in the key addressing scheme, which is used for the parallelization of the code. Exploiting the data locality once again on the local processor, one can optimize the usage of secondary disk storage and of the memory hierarchy of cashes, which is difficult otherwise [11].

This framework for the parallelization of adaptive codes originally has been invented for particle methods [29] and has been generalized to programming environments for some grand challenge PDE projects [22]. Multigrid methods have been considered in [12, 13].

NUMERICAL EXPERIMENTS

As test cases for our approach, we consider the Poisson equation and a problem of linear elasticity, the Lamé equation in the displacement approach. We use a finite difference discretization, where the degrees of freedom are associated with the nodes, and the differential operator is defined o the edges connecting the nodes. In a similar fashion the additive multigrid can be defined.

All numbers reported are scaled CPU times measured on a Cray T3E parallel computer with 600MHz Alpha processors. We use the portable message passing MPI programming interface and its intrinsic timing routines.

We consider adaptively refined grids of the unit square and the unit cube. The grids are refined towards the corner $\vec{0}$, see Figure 3. Tables 1 and 2 depict times in the adaptive grid refinement case of the Poisson equation. These times give the wall clock times for the solution of the equation system, on different levels of adaptive grids and on different numbers of processors. We assume a constant number of iterations within a nested iteration.

time	processors					
nodes	1	4	16	64	128	256
1089	5.08	1.27	0.72	0.64	0.84	1.30
1662	5.85	2.01	0.97	0.72	0.86	1.33
2745	10.7	3.26	1.37	0.85	0.94	1.38
4834	20.3	5.84	2.01	1.08	1.08	1.46
8915	39.8	10.9	3.38	1.42	1.26	1.56
16948	78.5	39.7	5.68	2.08	1.66	1.78
32788	157	77.7	10.7	3.34	2.30	2.14
64251			20.7	5.97	3.62	2.80
126810				10.9	6.14	4.12
251468					11.2	6.64
500135					21.2	11.7
996531					41.0	21.8
1988043					80.6	41.4

Table 1: Adaptive refinement example, two-dimensional Poisson problem, timing, Cray T3e-1200.

time	processors					
nodes	1	4	16	64	128	256
35937	291	85.6	29.6	11.2	7.61	5.94
50904	423	129	41.0	14.8	10.1	7.17
89076	405	236	71.2	24.6	14.6	9.98
189581			154	49.7	29.2	17.2
460421				109	61.1	35.6
1201650					142	77.2
3251102					345	188

Table 2: Adaptive refinement example, three-dimensional Poisson problem, timing, Cray T3e-1200.

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 all processor efficiently, the grid has to be fine enough, i.e it has to have more than 10^5-10^6 unknowns.

As a second test case for our approach, we consider the linear elasticity problem. We consider the Lamé equation [6] in the displacement formulation, as a linear three dimen-

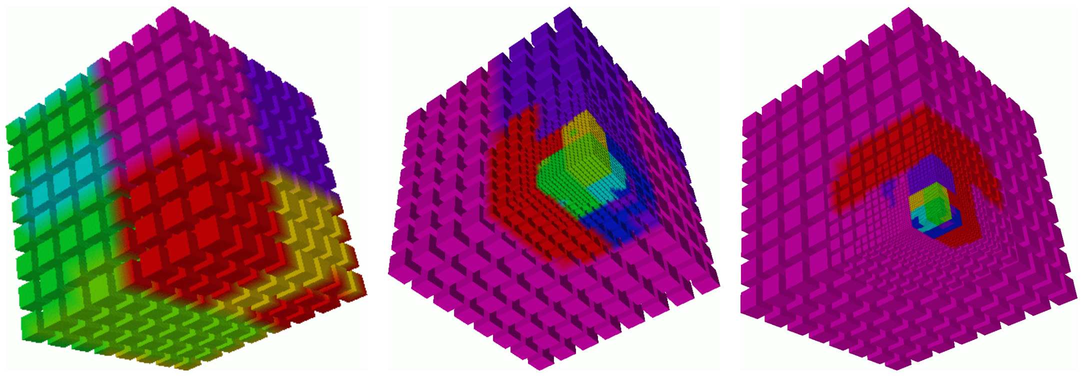

Figure 3: A sequence of adaptively refined grids mapped to 8 processors, partition is color coded.

sional problem. The body under a different load $\vec{f}$ is considered such that singularities occur. Here adaptive refinement is used to resolve the singularities.

time		processors								
nodes	dof	1	4	16	64	128	256	512	768	1024
35937	107811	162	34.1	9.11	2.23	1.23	0.75	0.55	0.56	0.52
109873	329619	435	108	29.6	7.20	3.57	1.87	1.13	0.91	0.80
410546	1231638			114	28.6	14.2	7.02	3.51	2.48	1.94
1857030	5571090				133	67.1	33.3	16.5	11.0	
9619175	28857525					351				

Table 3: Adaptive refinement example, three-dimensional linear Elasticity problem, timing, Cray T3e-1200.

Also for Lamé's equation, principally the same behavior of our approach as for the Poisson problem can be seen. Table 3 show that our method scales well. Note that the parallel efficiency is even higher than for the Poisson problem. This is due to the higher amount of work associated with each node, while the expenses associated with the grid stay the same. In the elasticity case, there are three degrees of freedom located at each node.

GENERAL DOMAINS

The treatment of general domains can be done in several ways. The unit square or cube can be mapped to a curved domain by a differentiable mapping function which leads to a variable coefficient problem, several domains can be joined together in an overlapping or non-overlapping way which leads in a natural way to domain decomposition solvers, or the domain can be embedded into the square/ cube with a special type of discretization.

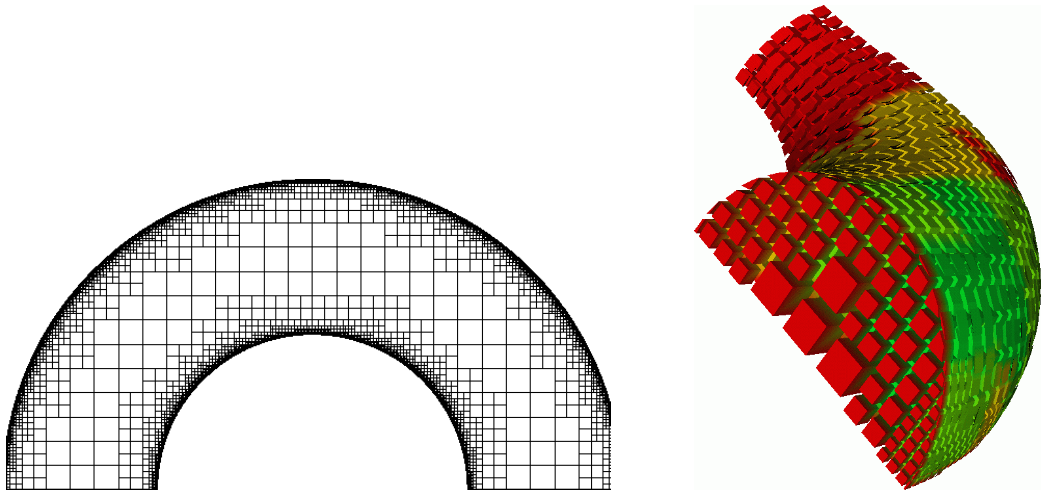
Figure 4: Adaptively refined grid of a 2D and a 3D torus with a Shortley-Weller type of boundary approximation.

We choose the last version: The domain is enclosed by a cube $\Omega \subset [0,1]^d$ and all nodes $x \in [0,1]^d \setminus \Omega$ are considered as boundary nodes. Furthermore, the finite difference stencils are adapted for nodes in the vicinity of the boundary $\partial\Omega$, that is all nodes whose difference stencil crosses the boundary. The step size h is reduced to the actual distance to the boundary, see [27, 15]. Furthermore, adaptive grid refinement is employed in the vicinity of the boundary to increase the resolution of the boundary, as can be seen in Figure 4. We also present the space-filling curve partitions of such grids in Figures 3 and 5. Some difficulties related to this approach are the symmetry of the discrete operator and the treatment of certain types of Neumann and Robin boundary conditions in a matrix-free implementation.

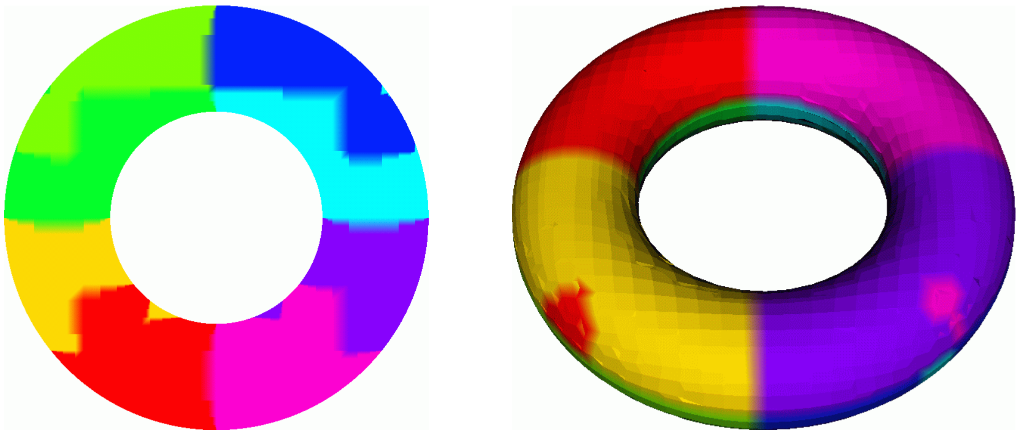
Figure 5: Space-filling curve partitions of grids of a 2D and a 3D torus for eight processors.

CONCLUSION

In this paper we gave a survey of the basic ingredients of an efficient solver for self-adjoint elliptic PDEs, i.e. multilevel solvers, adaptive grid refinement and parallelization. We focused on the interplay between the these ingredients and tried to illustrate how the they can be glued together into an adaptive parallel multilevel method. Here we proposed the application of hash storage techniques for data management and the use of space-filling curves for load balancing in the parallel version of the algorithms and presented a version of a parallel adaptive multilevel method based on these approaches.

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.

ACKNOWLEDGMENT

We want to thank Cray Research Inc. for the computing time on an experimental Cray T3E-1200 system.

REFERENCES

[1] J. Alber and R. Niedermeier. On multi-dimensional Hilbert indexings. In *Proc. of the Fourth Annual International Computing and Combinatorics Conference (CO-COON'98), Taipei 1998*, Lecture Notes in Computer Science. Springer, 1998.

[2] P. Bastian. Load balancing for adaptive multigrid methods. *SIAM J. Sci. Comput.*, 19(4):1303–1321, 1998.

[3] M. J. Berger and S. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, C–36:570–580, 1987.

[4] J. Bey. Analyse und Simulation eines Konjugierte-Gradienten-Verfahrens mit einem Multilevel Präkonditionierer zur Lösung dreidimensionaler elliptischer Randwertprobleme für massiv parallele Rechner. Master's thesis, RWTH Aachen, 1991.

[5] S. H. Bokhari, T. W. Crockett, and D. N. Nicol. Parametric binary dissection. Technical Report 93-39, ICASE, 1993.

[6] D. Braess. *Finite Elemente.* Springer, Berlin, 2nd edition, 1996.

[7] J. H. Bramble, J. E. Pasciak, and J. Xu. Parallel multilevel preconditioners. *Math. Comp.*, 55:1–22, 1990.

[8] A. Brandt. Multigrid solvers on parallel computers. In M. H. Schultz, editor, *Elliptic Problem Solvers*, pages 39–83. Academic Press, New York, 1981.

[9] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Letters*, 42:153–159, 1992.

[10] J. De Keyser and D. Roose. Partitioning and mapping adaptive multigrid hierarchies on distributed memory computers. Technical Report TW 166, Univ. Leuven, Dept. Computer Science, 1992.

[11] C. C. Douglas. Caching in with multigrid algorithms: problems in two dimensions. *Paral. Alg. Appl.*, 9:195–204, 1996.

[12] M. Griebel and G. Zumbusch. Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization. In J. Mandel, C. Farhat, and X.-C. Cai, editors, *Proc. Domain Decomposition Methods 10*, volume 218 of *Contemporary Mathematics*, pages 279–286, Providence, Rhode Island, 1998. AMS.

[13] M. Griebel and G. Zumbusch. Parallel adaptive subspace correction schemes with applications to elasticity. *CMAME*, 1999. submitted.

[14] C. E. Grosch. Poisson solvers on large array computers. In B. L. Buzbee and J. F. Morrison, editors, *Proc. 1978 LANL Workshop on Vector and Parallel Processors*, 1978.

[15] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen.* Teubner, 1986.

[16] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[17] P. Leinen. *Ein schneller adaptiver Löser für elliptische Randwertprobleme auf Seriell- und Parallelrechnern.* PhD thesis, Universität Dortmund, 1990.

[18] O. A. McBryan, P. O. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C. A. Thole, and U. Trottenberg. Multigrid methods on parallel computers – a survey of recent developments. *Impact of Computing in Science and Engineering*, 3:1–75, 1991.

[19] W. F. 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.

[20] J. T. Oden, A. Patra, and Y. Feng. Domain decomposition for adaptive hp finite element methods. In *Proc. Domain Decomposition 7*, volume 180 of *Contemporary Mathematics*, pages 295–301. AMS, 1994.

[21] Chao-Wei Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular and adaptive problems. In *Proceedings of International Conference on Parallel and Distributed Systems*, 1993.

[22] M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawai International Conference on System Sciences*, 1996.

[23] J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. Technical Report CS94-349, UCSD, Dept. Computer Science, 1994.

[24] A. Pothen. Graph partitioning algorithms with applications to scientific computing. In D. E. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, pages 323–368. Kluwer, 1997.

[25] S. Roberts, S. Kalyanasundaram, M. Cardew-Hall, and W. Clarke. A key based parallel adaptive refinement technique for finite element methods. In *Proc. Computational Techniques and Applications: CTAC '97*. World Scientific, 1998.

[26] H. Sagan. *Space-Filling Curves*. Springer, New York, 1994.

[27] G. H. Shortley and R. Weller. Numerical solution of laplace's equation. *J. Appl. Phys.*, 1938.

[28] L. Stals. *Parallel Implementation of Multigrid Methods*. PhD thesis, Australian National Univ., Dept. of Mathematics, 1995.

[29] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Comput. Phys. Comm.*, 87:266–290, 1995.

[30] M. S. Warren and J. K. Salmon. Abstractions and techniques for parallel n-body simulations. In *Parallel Object Oriented Methods and Applications (POOMA) '96*, 1996.

[31] G. Zumbusch. Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme. SFB-Report 342/19/91A, TUM-I9127, TU München, 1991.