Dynamic loadbalancing in a lightweight adaptive parallel multigrid PDE solver.

Gerhard W. Zumbusch*

Abstract

A parallel version of an adaptive multigrid solver for partial differential equations is considered. The main emphasis is put on the load balancing algorithm to distribute the adaptive grids at runtime. The background and some applications of space-filling curves are discussed, which are later on used as the basic principle of the load-balancing heuristic. A tight integration of space-filling curves as a memory addressing scheme into the numerical algorithm is proposed. Some experiments on a cluster of PCs demonstrates the parallel efficiency and scalability of the approach.

1 An adaptive multigrid solver

Our goal is to solve a partial differential equation as fast as possible. We consider a multigrid solver, 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. 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.

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.

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 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.

We use an additive version of the multigrid method for the solution of the equation system, i.e. the so called BPX preconditioner [10]. This requires an outer Krylov iterative

*Department for Applied Mathematics, University Bonn, Germany.

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 The load-balancing problem

Parallel multigrid methods on a sequence of unstructured 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 [12]. 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 [29] or some specific heuristics [6, 9, 4]. For a survey on grid partitioning methods we refer to [25].

Graph partitioning can be expensive. However, in the framework of adaptive, element-wise refinement, partitions and mappings have to be computed quickly and during runtime. 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 [18].

On a distributed memory computer, the grid hierarchy has to be partitioned and maintained, which requires a substantial amount of bookkeeping. In [13] 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 [4, 29, 19]. Here, different repartitioning strategies for refined grids were used.

The parallelization of an adaptive code usually is non-trivial and requires a substantial amount of code for the parallelization only. Hierarchies of refined grids, where neighbor elements may reside on different processors, have to be managed [8]. That is appropriate ghost nodes and elements have to be created and updated, when the parallel algorithm performs a communication operation. This happens both in the numerical part, where an equation system is set up and solved, and in the non-numerical part, where grids are refined and partitioned, see also [4, 17].

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. Here, parallel graph coarsening is popular. It results in a coarser

graph on which then a more expensive heuristic on a single processor can be employed. However, graph coarsening is at least a linear time algorithm itself and lowers the quality of the heuristic further. This is why we look for even cheaper partition methods. They are provided by the concept of *space-filling curves.*

3 Space filling curves

First we have to define curves. The term curve shall denote the image of a continuous mapping of the unit interval to the $\mathbb{R}^d$. A curve is space-filling if and only if the image of the mapping does have a classical positive d-dimensional measure. For reasons of simplicity we restrict our attention to a simple domain, namely the unit square. We are interested in a mapping

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

One of the oldest and most prominent space-filling curve, the Hilbert curve can be defined geometrically [16], see also [27]. 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.

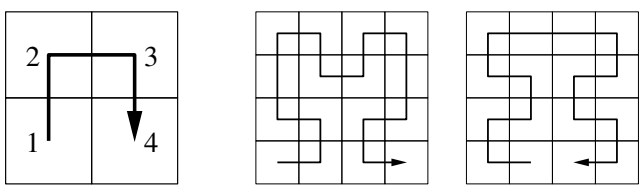

FIG. 1. *Construction of a Hilbert space-filling curve. The open and the closed curve.*

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 affine mappings and connections between the loose ends of the pieces of curves, the Hilbert curve is obtained.

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].

The Hilbert curve may be the most prominent curve. However it is not the oldest one. The discovery of 'topological monsters', as they were called initially started when Cantor in 1878 proved the existence of a bijective mapping between two smooth, arbitrary manifolds of arbitrary, but finite dimension. A year later Netto proved that such mappings are non-continuous. This means that they are not curves in our previous definition. However, without the property of an injective mapping, Peano was construct a space-filling curve

in 1890, see Figure 2. Later on several other curves were constructed by Hilbert, Moore, Lebesgue and Sierpinski and many others.

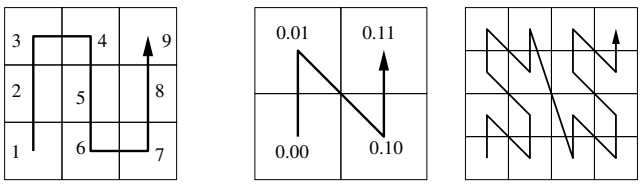

FIG. 2. *The construction of a Peano (left) Lebesgue curve (middle and right).*

There are two basic differences of the Lebesgue space-filling curve compared to the previous curves: The curve is differentiable almost everywhere, while the previous mentioned curves are not differentiable anywhere. The Hilbert and Peano curves are self-similar, a feature they share with Fractals: Sub-intervals of the unit interval I are mapped to curves of similar structure than the original curve. However, the Lebesgue curve is not self-similar. It can be defined on the Cantor set C. This set is defined by the remainder of the interval I, after successively one third $(1/3, 2/3]$ has been removed. Any element $x \in C$ of the Cantor set can be represented as a number in base-3 expansion, $0_3.t_1t_2t_3\ldots$ where all digits t_i are zeros and twos $t_i \in \{0, 2\}$, because the one's have been removed by the construction of the Cantor set. The mapping of the Lebesgue curve is defined by

$$f(0_3.(2d_1)(2d_2)(2d_3)\ldots) \;:=\; \begin{pmatrix} 0_2.d_1d_3d_5\ldots \\ 0_2.d_2d_4d_6\ldots \end{pmatrix} \text{ with binary digits } d_i \in \{0,1\}\ .$$

The function f defined on the Cantor set C is extended to the unit interval I by linear interpolation. The generator of the Lebesgue curve looks like in Figure 2. Although constructed by digit shuffling, the curve is continuous and a space-filling curve. The order of the quadrants imposed by the generator of the curve can be found in the depth-first traversal of oct-trees and related algorithms.

4 Applications of space filling curves

Space-filling curves had been created for purely mathematical purposes. However, nowadays there is a number of applications of space-filling curves. Basically, multi-dimensional data is mapped to a one-dimensional sequence. This mapping is useful for load-balancing of parallel simulations on a computer, for data locality of memory or disc accesses inside a computer in geographic information systems [2, 11], for finding shortest paths in optimization [3, 7] and ordering data in computer graphics [23, 32, 30] and in other applications [5].

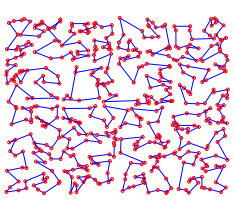

FIG. 3. *A traveling salesman tour defined by a closed Hilbert space-filling curve.*

Particle or n-body problems are defined by the interaction of n entities by some interaction forces. This model describes different phenomena like the movement of planets or dust under gravitational forces in astrophysics and the dynamics of atoms or groups of atoms in molecular dynamics. The number of particles of interest easily reaches the range of $10^6 - 10^9$. The model leads to a system of n ordinary differential equations. The right hand side of each equation consists of the $n-1$ forces of particles, which interact with the particle under consideration. Usual model forces decay with the distance of the particles, which can be exploited by efficient approximation algorithms like the fast-multipole and the Barnes-Hut algorithm. However, there is still a global coupling of the particles, which cannot be neglected. Furthermore, particle can be distributed randomly and can form clusters. Hence a parallel particle simulation code requires efficient load-balancing strategies for the enormous amount of data. The particle move in space due to the acceleration imposed by the interaction-forces. This means, that a re-balancing or a dynamic load-balancing is needed for parallel computing, which can be done by space-filling curves [31, 28, 22].

A slightly different situation can be found in adaptive discretizations of partial differential equations. Now a grid consisting of n nodes and elements or volumes has to be distributed to a parallel computer. The nodes and elements can be found at arbitrary positions (completely unstructured grids) or at fixed positions, which are a priori known (structured grids) or at least computable by grid-refinement rules from a coarse grid (adapted grids). The degrees of freedom are coupled locally, usually between neighboring nodes. However, algorithms for the solution of the resulting equation systems couple all degrees of freedom together, which imposes difficulties on the parallelization of the solver. For the solution of stationary problems, no nodes are moved in general. However, due to adaptive grid refinement, new nodes are created during the computation. This requires dynamic load-balancing, which can also be done by space-filling curves [20, 22, 15, 26].

5 A parallel solver based on space filling curves

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.

There are different ways to implement the first step of our partitioning procedure, which depend on the type of space-filling curve in use. We have to map a node or some other geometric entity onto an iterate of a space-filling curve in space. We look for a point on the curve in the vicinity and compute its position on the space-filling curve.

This can be done easily for Lebesgue-type of curves: We round the coordinates of the node (x, y) (inside the unit square) to a finite binary representation of k digits. This results in a point which is on the kth iterate of Lebesgue curve and all further iterates

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} 0_2.x_1x_2x_3\ldots x_k \\ 0_2.y_1y_2y_3\ldots y_k \end{pmatrix} \mapsto 0_2.x_1y_1x_2y_2\ldots x_ky_k\ .$$

This procedure can be equivalently described as: Compute the address of the quad-tree cell of the kth level where the node resides in.

Other type of space-filling curves require slightly more involved procedures to find a point on the curve next to the node and do the inverse mapping. However, the procedure

related to the Hilbert curve can be described as a post-processing step of the Lebesgue procedure above. In a single loop from 1 to k the tuples (x_i, y_i) can be transformed into tuples $(\hat{x}_i, \hat{y}_i)$ of the corresponding Hilbert curve, which is a similar process as the recursive construction of the Hilbert curve. The same is true for space-filling curves in $\mathbb{R}^d$.

The space-filling discussed so far define a continuous mapping of a line segment $[0,1] \subset \mathbb{R}$ to a polygonal area $[0,1]^2 \subset \mathbb{R}^2$. Such curves can also be used for the inverse mapping from a domain $\Omega \subset \mathbb{R}^d$ to an interval $[0,1] \subset \mathbb{R}$. This means that we can map geometric entities in $\mathbb{R}^d$ to the one dimensional interval. Entities, which are neighbors on the interval, are also neighbors in the volume $\mathbb{R}^d$. Unfortunately the reverse cannot be true and neighbors in the volume may be separated through the mapping. However, we know how to solve a one-dimensional partition problem: We cut the interval into equal sized pieces, which gives perfect load-balance and small separators of size one. The partition of the volume $\mathbb{R}^d$ induced by the space-filling curve, see Figure 4, 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.

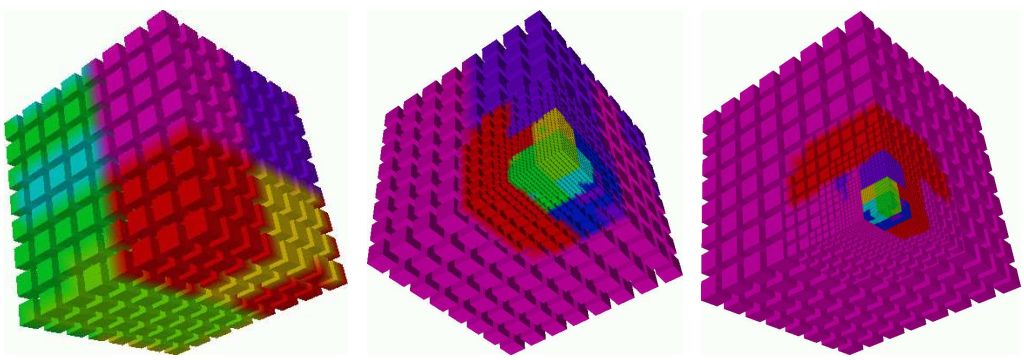

FIG. 4. *A sequence of adaptively refined grids mapped to 8 processors, partition is color coded.*

Partitioning by space-filling curves has been employed for finite element computations in [21, 20] and has been compared to other heuristics in [24]. The main advantage of space-filling curves in this context is their simplicity: If we want to bisect a set S of points $x_i \in \Omega$ on a space-filling curve, we can do this by a single number s and the inverse space-filling curve mapping f^{-1}

$$S = \{x_i | f^{-1}(x_i) \leq s\} \cup \{x_i | f^{-1}(x_i) > s\} .$$

Each point is either left or right of the reference s on the space-filling curve. We we take s as the median of $f^{-1}(S)$, both subsets are of the same size. In the same way we can partition the set of points S into p different subsets, if we partition $f^{-1}(S) \subset [0,1]$ by $p-1$ separators s_i like $\{x_i | s_j < f^{-1}(x_i) \leq s_{j+1}\}$. There is almost no bookkeeping necessary, because the partition is deterministic and can be computed on the fly from the separators. Each sub-set is assigned to one processor. Only the $p-1$ separators have to be stored on each processor.

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.

The remaining question is about the quality of this simple heuristic because it is cheap and scalable. The workload is partitioned evenly. Hence the communication volume cannot be optimal for $d > 1$. However, we can formulate

Theorem (Separator sizes for Hilbert curve partitionings). *Suppose that we partition a uniform grid which covers the cube $[0,1]^d$ by a Hilbert curve, then for any subdomain Ω_i with n nodes obtained this way*

the number of boundary edges s is $s \approx n^{\frac{d-1}{d}}$ with constants dependent only on d and the discretization stencil.

Hence the separator sizes for uniform grids obtained by space-filling curve partitionings are optimal up to a constant, if we neglect effects of the boundary $\partial\Omega$. This indicates, that the partitioning algorithm performs well in this case. In more general cases of adaptive refined grids, numerical experiments indicate that the graph separators of space-filling curve partitions are of sufficient quality, see [24].

6 Experiments

Instead of linked lists or trees, we propose to use *hash storage* techniques. First we describe a *key based* addressing scheme. An implementation of a key based scheme with hash tables is described later. Each entity of the grid is assigned to a unique key, which is an integer number. The entity 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 of hierarchical son nodes and father nodes can be computed from the node's coordinates easily. Hence, the keys of the nodes are available and the nodes can be looked up in the vector, if they exist. Nodes on the finest refinement level can be determined by the fact, that they do not have son nodes. 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 [15]. Similar key based addressing schemes can be obtained for other grid refinement procedures and for different domains, see [29, 26]. For example the element of a general triangulation or tetrahedrization τ_0 of a polygonal domain Ω can be enumerated. Along with a numbering scheme based on local coordinates in each element, a general key addressing scheme can be established. Also quad- or octree-tree techniques

can be used, see [31].

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 [31, 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 [14].

As a test case for our approach, we consider 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 our parallel computing cluster 'Parnass2'. It consists of dual processor Pentium II 400MHz boards with at least 256 Mbytes of main memory, interconnected by a Myrinet network in a fat-tree configuration. The MPI message passing protocol implementation Mpich-PM showed a bandwidth between each two boards of 850 Mbit/s, see also [33]. Furthermore we compare the execution time of the load-balancing step with the execution time of the linear solver and compute their ratio $\alpha := t_{\text{balancing}}/t_{\text{solving}}$.

time		processors						
nodes	dof	1	2	4	8	16	32	64
450	1350	1.44	0.99	0.80	1.35	0.50	0.39	1.05
1155	3465	4.14	2.48	1.71	1.32	1.00	0.70	2.74
4412	13236	19.0	10.3	6.09	5.23	3.07	1.89	1.21
18890	56670	98.6	50.3	28.1	20.6	11.6	6.35	3.70
93021	279063	582	294	157	102	54.8	28.2	15.1
506620	1519860				556	306	155	78.1
3178218	9534654							494

TABLE 1

Adaptive refinement example, elasticity 3D, timing, 1 to 64 processors.

In the single processor case, no load balancing is needed, so the partitioning and mapping time and their ratio α is zero. Otherwise, the nodes have to be partitioned and

ratio α	processors						
nodes	1	2	4	8	16	32	64
18890	0	3.5e-4	3.4e-4	3.9e-4	6.0e-4	1.15e-3	2.73e-3
93021	0	3.6e-4	3.5e-4	4.0e-4	4.9e-4	6.9e-4	1.43e-3
506620	0	—	—	4.2e-4	4.6e-4	6.3e-4	9.1e-4

TABLE 2

Ratio α of execution times of partitioning and mapping nodes to solving the equation system.

mapped by a parallel (partial) sort algorithm. The relative cost of partitioning nodes α is of the order 1e-3. Hence our load balancing is also very cheap in this case. In the adaptive grid case, dynamic load balancing generally is required. Note that in all cases load balancing is much cheaper than solving the equation system. However, higher number of processors make the mapping and partitioning relatively slower. Mapping data for adaptive refinement requires the movement of a large 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 [4, 29].

References

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

[2] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer, *Space filling curves and their use in the design of geometric data structures*, Theoretical Computer Science, 181 (1997), pp. 3–15. also as Proc. Second Latin American Symposium on Theoretical Informatics, LATIN '95, Valparaiso, Chile, Springer, Lecture Notes in Computer Science 911, p. 36ff.

[3] J. Bartholdi and L. K. Platzman, *Heuristics based on space-filling curves for combinatorial optimization problems.*, Management Science, 34 (1988), pp. 291–305.

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

[5] I. Beichl and F. Sullivan, *Interleave in peace, or interleave in pieces*, IEEE Computational Science & Engineering, 5 (1998), pp. 92–96.

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

[7] D. Bertsimas and M. Grigni, *On the spacefilling curve heuristic for the Euclidean traveling salesman problem*, tech. rep., Massachusetts Institute of Technology, Cambridge, MA, 1988.

[8] K. Birken and C. Helf, *A dynamic data model for parallel adaptive PDE solvers*, in Proceedings of HPCN Europe 1995, B. Hertzberger and G. Serazzi, eds., vol. 919 of Lecture Notes in Computer Science, Milan, Italy, 1995, Springer.

[9] S. H. Bokhari, T. W. Crockett, and D. N. Nicol, *Parametric binary dissection*, Tech. Rep. 93-39, ICASE, 1993.

[10] J. H. Bramble, J. E. Pasciak, and J. Xu, *Parallel multilevel preconditioners*, Math. Comp., 55 (1990), pp. 1–22.

[11] E. Bugnion, T. Roos, R. Wattenhofer, and P. Widmayer, *Space filling curves versus random walks*, in Proc. Algorithmic Foundations of Geographic Information Systems, vol. 1340 of Lecture Notes in Computer Science, Springer, 1997.

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

[13] J. De Keyser and D. Roose, *Partitioning and mapping adaptive multigrid hierarchies on distributed memory computers*, Tech. Rep. TW 166, Univ. Leuven, Dept. Computer Science, 1992.

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

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

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

[17] M. T. Jones and P. E. Plassmann, *Parallel algorithms for adaptive mesh refinement*, SIAM J. Sientific Computing, 18 (1997), pp. 686–708.

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

[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, vol. 180 of Contemporary Mathematics, AMS, 1994, pp. 295–301.

[21] C.-W. 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] A. Pérez, S. Kamata, and E. Kawaguchi, *Peano scanning of arbitrary size images*, in Proc. Int. Conf. Pattern Recognition, 1992, pp. 565–568.

[24] J. R. Pilkington and S. B. Baden, *Partitioning with spacefilling curves*, Tech. Rep. CS94-349, UCSD, Dept. Computer Science, 1994.

[25] A. Pothen, *Graph partitioning algorithms with applications to scientific computing*, in Parallel Numerical Algorithms, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., Kluwer, 1997, pp. 323–368.

[26] 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. to appear.

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

[28] J. K. Salmon, M. S. Warren, and G. Winckelmans, *Fast parallel tree codes for gravitational and fluid dynamical n-body problems*, International Journal of Supercomputer Applications, 8.

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

[30] D. Voorhies, *Space-filling curves and a measure of coherence*, in Graphics Gems II, J. Arvo, ed., Academic Press, 1994, pp. 26–30.

[31] M. Warren and J. Salmon, *A portable parallel particle program*, Comput. Phys. Comm., 87 (1995), pp. 266–290.

[32] R. E. Webber and Y. Zhang, *Space diffusion: An improved parallel halfoning technique using space-filling curves*, in Proc. ACM Comput. Graphics Ann. Conf. Series, 1993, p. 305ff.

[33] G. Zumbusch, *Parnass2: A cluster of PCs.* http://wwwwissrech.iam.uni-bonn.de/ research/projects/parnass2/, 1998.