A CACHE-AWARE ALGORITHM FOR PDES ON HIERARCHICAL DATA STRUCTURES BASED ON SPACE-FILLING CURVES

FRANK GÜNTHER, MIRIAM MEHL, MARKUS PÖGL, CHRISTOPH ZENGER *

Abstract. Competitive numerical algorithms for solving partial differential equations have to work with the most efficient numerical methods like multi-grid and adaptive grid refinement and thus with hierarchical data structures. Unfortunately, in most implementations, hierarchical data – typically stored in trees – cause a non-negligible overhead in data access. To overcome this quandary – numerical efficiency versus efficient implementation – our algorithm uses space-filling curves to build up data structures which are processed linearly. In fact, the only kind of data structure used in our implementation are stacks. Thus, data access becomes very fast – even faster than the common access to non-hierarchical data stored in matrices – and, in particular, cache misses are reduced considerably. Furthermore, the implementation of multi-grid cycles and/or higher order discretizations as well as the parallelization of the whole algorithm becomes very easy and straightforward on these data structures.

Key words. partial differential equations, space-tree, space-filling curves, cache-awareness

AMS subject classifications. 35J05, 25K99, 35R99, 65Y20, 68P05, 68V20

1. Introduction. Most of the computing time for the numerical solution of partial differential equations is usually spent for the multiplication of a sparse matrix – representing the discrete operator – with a vector – representing the approximate solution. The amount of computing time and also the amount of memory heavily depend on the data structure used to represent the matrix and the solution vector. Many clever approaches are known, but it is usually impossible to access the memory in the course of the computation in such a way that the addresses do not "jump". As modern computer architectures use one or more cache-levels for the access of memory, jumps in the address space may cause cache misses leading to a sometimes dramatic slow down of the computation. This fact is actually one of the most serious bottlenecks in high performance computing.

It seems to be very difficult to avoid these jumps in general. Therefore, we restrict our attention to a more special situation which is on the other hand general enough to be useful in many applications. The grid in our context is restricted to grids associated with space-trees [2], but we allow local (adaptive) refinement because adaptivity is crucial for the efficient solution of many problems. Moreover, space-trees allow a simple implementation of modern multilevel methods (additive or multiplicative) using appropriate hierarchical bases or generating systems ([13]) as we shall see in the sequel. Hierarchical bases or generating systems also allow the simple handling of hanging nodes appearing in case of local refinement. Complicated geometries can also be described easily by space-trees if some care is taken of a reasonably accurate discretization near the boundary.

As a main result of this paper, we show that in this restricted context we can avoid random access in the memory almost completely by using only a fixed number of stacks independent of the number of nodes in the grid. Stacks can be considered as the most simple data structures used in Computer Science. The two basic operations allowed on stacks are push and pop, where push puts data on top of a pile and pop takes data from the top of a pile. It is immediately clear that subsequent accesses to

*Institut für Informatik, TU München, Boltzmannstraße 3, 85748 Garching, Germany ({guenthef,mehl,poegl,zenger}@in.tum.de).

memory can move only from one memory location to the previous or next location and, thus, stacks can be implemented very efficiently on modern computer architectures. As a consequence, the organization of our algorithms has to be done in such a way that the data needed in the sequence of operations are always on top of one of the stacks, avoiding transports from one stack to another to get access to elements deeper in the stack.

The present algorithm described in this paper is based on a standard finite element discretization but the principle behind can be adapted to other discretization schemes as well. The leaves of the space-tree describe the finite elements and the accumulation process for the operator matrix – which is not explicitly stored but instead directly applied to the approximate solution – runs in a sequence described by a space-filling curve, the so-called Peano-curve.

Space-filling curves are a well-known device to design efficient algorithms in computer graphics (see e.g. [28, 30]). In the context of numerical simulations based on space-trees, space-filling curves are already an established tool for some key-based addressing of grid elements and/or nodes and, in particular, data parallel implementations (see e.g. [32, 33, 15, 16, 21, 22, 26, 18, 23, 22]). The grid partitioning algorithm defined by space-filling curves can be shown to have linear complexity and to give quasi-optimal partitions with respect to cut-sizes ([32]). In addition, the partitioning is well suited for multilevel grids ([32, 33, 15, 16, 21, 26, 18, 23]).

It is also known that – due to locality properties of the curves – reordering grid cells according to the numbering induced by a space-filling curve improves cache-efficiency (see e.g. [1]). Similar benefits of reordering data along space-filling curves can also be observed for other applications such as matrix transposition ([8]) or matrix multiplication ([9]). We go one step further and – in addition to the reordering of cells – construct stacks for which we do not need any addressing and/or hashing as we always know that all data needed lie on top of one of the stacks and can thus be accessed in an even more cache-efficient way.

In contrast to other approaches to cache-optimzations for PDE solvers that work with some hardware-oriented strategies like specialized data padding [**?**], the efficiency of our algorithm doesn't depend on the particular setting of cache parameters like cache-line length, associativity etc. In literature, such wich are cache-aware by concept without detailed knowledge of the cache parameters are also called cache-oblivious [7, 12, 25]. The general need for cache-optimization on the software side can easily be deducted from the impossibility of the implementation of an optimal replacement strategy for the cache-lines on the hardware side [29].

Section 2 gives some basic background information on the operator evaluation during a run along a space-filling curve, in Section 3, the essence of our algorithm, the construction of stacks with the help of the Peano-curve is described. Section 4 describes the realization of some numerical algorithms like multi-grid and evaluation of the operator matrix on adaptively refined grids. Finally, in Section 5, we give some results – processing times, number of cache misses etc. – for some test examples. Section 6 shortly describes the extension to the three-dimensional case including numerical results as well.

2. Operator Accumulation along Space-Filling Curves. In the mathematical definition, a space-filling curve is a surjective continuous mapping of the unit interval $[0;1]$ to a compact d-dimensional domain Ω with positive measure. In our context, Ω is always the unit square or the unit cube in 3D, respectively. As we look at multilevel adaptive rectangular grids, we restrict to recursively defined, self-similar

space-filling curves with rectangular recursive decomposition of the domain. These curves are given by a simple generating template and a recursive refinement procedure which describes the (rotated or mirrored) application of the generating template in sub-cells of the domain to be covered ([27]). Prominent representatives of this class of space-filling curves are the Hilbert-curve and the Peano-curve. See figure 2.1 for some iterates of the two-dimensional curves.

In fact, as our grids have finite resolution, the iterates – so called discrete space-filling curves – are what we need (instead of the continuous space-filling curves). If we work with adaptively refined grids, the iterate we use in a particular part of our domain depends on the local resolution (see also figure 2.1 for some examples).

In our algorithm, this discrete space-filling curve defines the processing order of grid cells – corresponding to the leaves of our space-tree in a single-level context or all nodes of the space-tree in a multilevel context. The application of the operator-matrix to the vector of data is done in a strictly cell-oriented way. For this, we decompose the discrete operator into parts per cell which accumulate to the result of the operator evaluation after one run over all grid cells. This method is standard for finite element methods (see e.g. [6]), but can be generalized to 'local' discrete operators, which means that for the evaluation in one grid point, only direct neighbors are needed. In some cases – if the operator can be composed of a small number of 'local' operators – this restriction can even be weakend. To illustrate the cell-oriented operator decomposition, we look at the one-dimensional three-point stencil

$$\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$

This simple stencil can be decomposed in a cell-part

$$\begin{bmatrix} 1 & -1 \end{bmatrix}$$

for the grid cell at the left-hand side of the respective point and

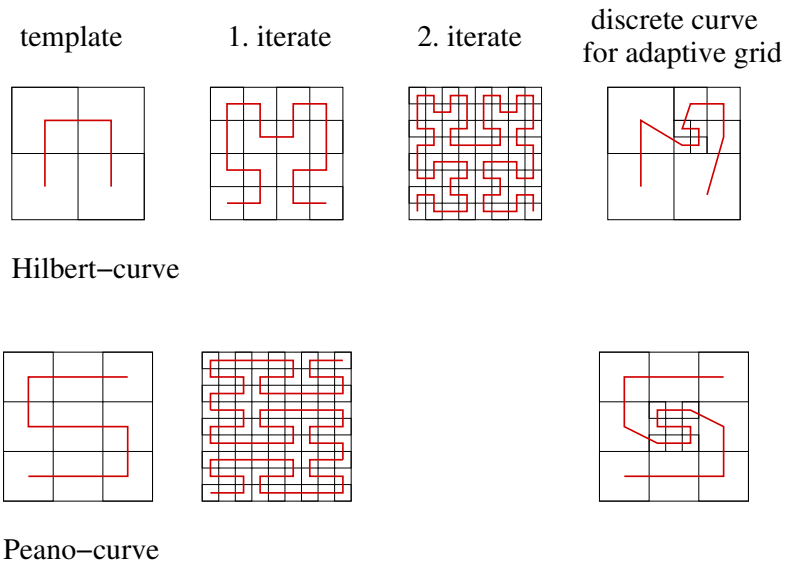

FIG. 2.1. *Generating templates, first iterates and discrete curve on an adaptively refined grid for two-dimensional Hilbert- and Peano-curves*

$$\begin{bmatrix} -1 & 1 \end{bmatrix}$$

for the grid cell at the right-hand-side.

Thus, if we look at a space-tree with function values located at the vertices of cells, we have to construct our stacks such that all data from the respective cell-vertices lie on top of the stacks when we enter the cell during our run along the discrete space-filling curve.

3. Construction of Stacks.

3.1. Regular Grids with Nodal Data. To show the interaction of discrete space-filling curves and data stacks, we will start with a simple regular two-dimensional grid and nodal data. Figure 3.1 shows a grid where the processing order of grid cells is given by a discrete Peano-curve.

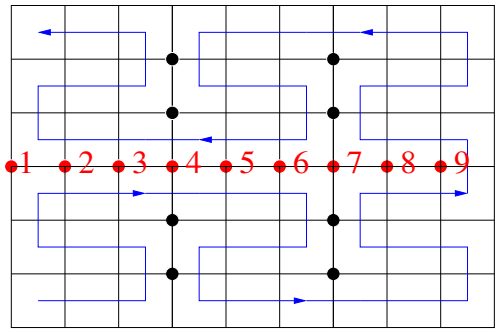

FIG. 3.1. *Example in 2D using the Peano-curve*

Let's have a closer look at data points on the middle line marked by 1 to 9. In the lower part of the domain, these data are processed linearly from 1 to 9, in the upper part vice versa from 9 to 1. For the Peano curve and a regular grid, this linear forward and backward processing of the middle line can be shown for arbitrarily fine grids, as well. Analogously, all other grid points can be organized on lines which are processed linearly forward and backward, if we use the Peano-curve to define the processing order of cells. Therefore, we organize all data within the well-known concept of stacks where we only have two possibilities of data access:

- push(data,number): write data on top of stack number
- pop(number): read uppermost element of stack number

Here, data is a structure where all physical information of a grid point is held – for example the velocity vector and the type of boundary conditions.

A closer look at the 2D example in figure 3.2 shows that it is sufficient to have two stacks: $1D_{black}$ for the points on the right-hand-side of the curve and $1D_{grey}$ for the points at the left-hand-side of the curve. These stacks represent the evolving lines of points along the curve as described above. The behavior of a point concerning read and write operations on stacks can be predicted in a locally deterministic way. We only have to classify the points as 'inner points' or 'boundary points' and respect the progression of the curve.

If we have a closer look at the example for the Peano-curve, we can already extract two important properties our space-filling curves have to fulfil: First, the processing

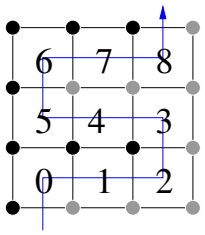

FIG. 3.2. *Stacks within usage of the Peano-curve*

order of grid points on a line has to be inverted if – on our way along the Peano-curve – we switch from the domain on the right-hand-side of the line to the domain on the left-hand-side. This is quite easy to fulfill in the two-dimensional case, but in the three-dimensional case, the same has to hold for grid points on planes (parallel to the coordinate planes). Second, a refinement of the grid must not destroy the order of existing points, but only insert new points between existing points. We could not find any Hilbert-curve in 3D fulfilling these two properties and there are good reasons to assume that this is not possible at all. Thus, we restrict our attention to Peano-curves in the following. For the Peano-curve even generalizations to four or more dimensions are complicated but straightforward.

It is a disadvantage of the Peano curve that in the refinement process each side of a cube is partitioned into three equal parts leading to ternary trees instead of the standard binary trees used for quad-trees or octrees. This implies that the number of grid-points grows like 3^d instead of 2^d in d-dimensional space from level to level. This implies that for a three-dimensional problem the number of grid-points is multiplied by a factor of 27 if we add another refinement level.

An efficient algorithm deducted from the concepts described above passes the grid cell-by-cell, pushes/pops data to/from stacks deterministically and automatically and will be cache-aware by concept (not by optimization!) because of the fact that using linear stacks is a good idea in conjunction with modern processors prefetching techniques.

3.2. Extension to Adaptive Grids and Hierarchical Data. Up to this point, we have restricted our considerations to regular grids to explain the general idea of our algorithm, but the whole potential of our approach gets obvious only when we look at adaptively refined grids and – in the general case – hierarchical data in connection with generating systems ([13]). This leads to more than one degree of freedom per grid point and function on coarse grid levels.

Before we can define our stacks for this case, we need an algorithm which now recursively visits the cells of different grid levels in a top-down depth-first process. Since the discrete space-filling curve is defined by a recursion itself, this order can be deducted directly from the space-filling curve of the respective level. Figure 3.3 shows an example for such an order. The first cell visited is the coarsest cell, containing the whole domain. Next, the cells of the first refinement level are processed according to the first iterate of the Peano-curve. As soon as we reach a cell which is further refined (in this example the middle cell), we process all 'sons' of this cell according to the respective part of the next iterate of the Peano curve before we return to the next cell of the first refinement level.

In our stack context, we have to assure that even then predictable and linear data access to and from stacks is possible. As points are visited on different grid levels

now, we have to assure that grid points of coarser grid levels lie above those of finer levels in our stacks. Therefore, we have to change two basic properties of the concept described above: First, we have to look at a cell consisting of 3×3 finer cells and thus containing 16 grid points instead of single cells with only 4 grid points. Second, we have to introduce four 'colors' instead of two (in figure **??**, the four colors are marked by crosses, circles, triangles, and squares) and a second type of stacks, so called $0D$-stacks, in addition to the $1D$-stacks. The $0D$-stacks can be interpreted as an intermediate storage for vertex data to avoid the hiding of informations associated to different levels wheras the $1D$-stacks represent certain grid lines or parts of them, respectively. Again, we have a small and fixed number of stacks which is – in particular – independent of the problem size: four $1D$-stacks and four $0D$-stacks transporting hierarchical data over grid levels.

The construction of these stacks can be understood best if we look at an example again. Figure 3.4 shows two levels of a grid. Points 1 and 2 exist on both levels – marked by 1a, 2a for the coarse level and 1b, 2b on the fine level. In a first step, we only look at hierarchical data with respect to a hierarchical basis but not a generating system yet. We process our grid cells in the recursive top-down order described above. If we used two 'colors' or two stacks, respectively, as with the nodal basis, the following problem would occur: Points 3 and 4 would have the same color as point 2a. Therefore, cells 3 and 8 on the fine level would store points 3 and 4 on their stack and, due to recursivity, cell 1 on the coarse level would store point $2a$ *after-wards* on top of the same stack (point $2a$ is needed again in coarse cell 4). So cell 9 and cell 14, respectively, on the fine level would not be able to pop points 3 and 4 from the stack, because the uppermost point on this stack would be $2a$. If we introduce four 'colors' representing four different stacks (horizontal/vertical lines at the right-hand side of the Peano-curve, horizontal/vertical lines at the left-hand side of the Peano-curve) of the same type as shown in figure 3.4, this will not happen.

If we now allow a generating system, which means that one grid point contains data on *all* grid levels, the next problem arises: Cell 20 on the fine level would push point $1b$ to the 'triangle'-stack and it would be the uppermost element on this stack, when coarse cell 3 is finished. Thus, coarse cell 4 could not pop point $2a$ from the stack. To avoid level-dependent coloring of points, we decided to overcome this problem by using two stacks of each 'color', we call them $1D$- or line-stacks and $0D$- or point-stacks. Together with a strict order of pushes and pops of points the problem described above is eliminated. For example, if coarse cell 1 stores point $2a$ in the 'ttriangle'-$1D$-stack and fine cell 20 pushes point $1b$ to the 'triangle'-$0D$-stack, everything is fine. This consiederation results in an algorithms where every point in the grid is written to the $0D$-stack of the correct 'color' when needed by a cell for the first time, poped from there and written to the $1D$-stack when needed by the second cell, moved again to the $0D$-stack when needed by the third cell, and, finally, popped from there

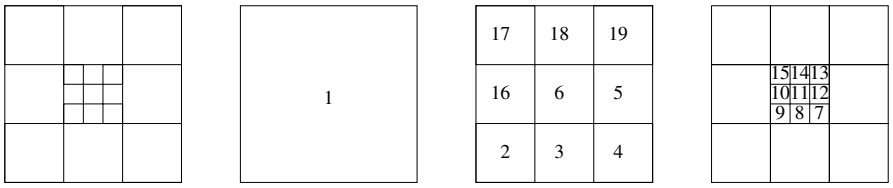

FIG. 3.3. *Example for the order of hierarchical cells defined by the discrete Peano-curve*

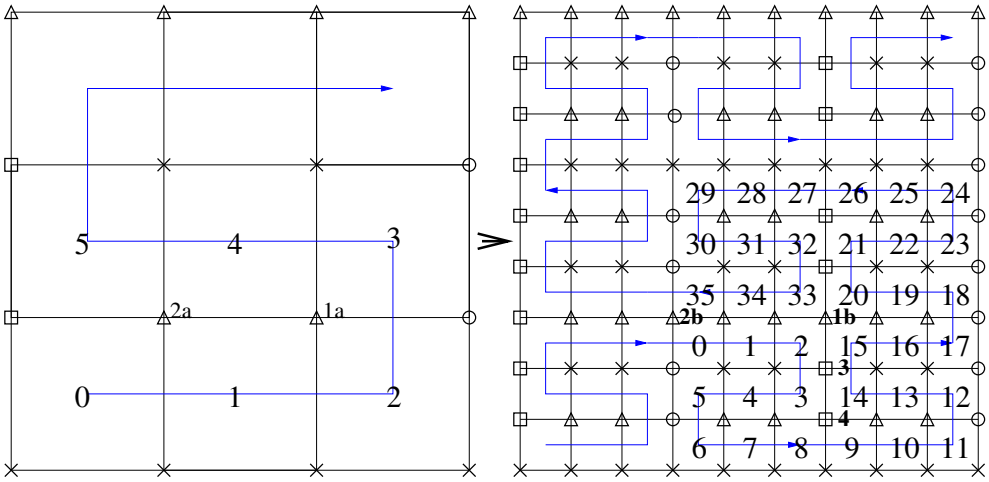

FIG. 3.4. *Recursive definition of stack-points using four colors/eight stacks*

and written to the output after being used by the fourth neighbouring cell. Since recursively processed coarser grid points of a local area always lie above the finer ones, it is guaranteed that they can be popped from stacks before the fine grid points – which fits the top-down structure of the algorithm.

An extension to adaptively refined grids is very easy, now: a local refinement of a cell can be handled nearly the same as a refinement in a regular grid. We only have to provide a set of rules for the 'hanging nodes' at the boundary of a local refinement since they will be visited by two (fine) cells only instead of four. The stack-concept is not disturbed by this since finer grid points are placed between the coarser grid points on our stacks.

3.3. Iterations on Data. In the previous we have only talked about the intermediate storage of data within one solver iteration. Thus, we have to enhance our stack system by some input and output stacks to store all data between iterations. For this, we use so called $2D$- or plane-stack stacks (as they include all data of the computational domain) which contain vertex data in the order of their first usage during one iteration. To store the output of the iteration, we write all points to a $2D$-stack again as soon as they are 'ready'.

It can easily be seen, that, if we process the grid cells in opposite direction in the next iteration, the order of data within this $2D$-stack enables us to pop all grid points from this $2D$-stack as soon as they are 'touched' for the first time. Such, we can use the $2D$-stack as input stack again very efficiently. We apply this repeatedly and, thus, change the processing direction of grid cells after each iteration.

With the ideas described above, we can now define numerical algorithms suited to adaptively refined grids, hierarchical data and generating systems in the following section.

4. Numerical Algorithms. As mentioned above, our data structures are very well suited for highly efficient numerical methods such as multi-grid, adaptive grid refinement and higher order discretizations. In this section, we will describe the concrete algorithmic realization of these methods.

We assume that we have to solve a system of linear equations

$$L_h u_h = f_h, \tag{4.1}$$

where h indicates the (locally defined) width of the square/cubic grid cells and the discrete operator L_h results from some 'local' (see above) discretization of a differential operator.

4.1. Relaxation Methods. Before we describe the implementation of a multi-grid method, we will shortly mention the realization of relaxation methods with the help of our stacks and the cell-oriented evaluation of operators: The only thing special in comparison to standard implementations is that we do not compute the residual components $r_h^{(i)} = f_h^{(i)} - (L_h u_h)^{(i)}$ in one step per point, but accumulate them while visiting neighboring cells (as described above when we explained the cell-oriented evaluation of operators). For this, we need an appropriate decomposition of the right-hand side into cell-parts, which can easily be achieved for example by dividing f_h into 4 equal parts, which are assigned to the four neighboring cells. As soon as a grid point is visited the last time, we have computed the whole residual and the respective value is updated according to the iteration scheme used.

4.2. Multi-grid. As, in our stack, we store data of all refinement levels of the grid, the algorithm works on a generating system (in the context of finite elements) instead of a basis only. A multi-grid cycle corresponds to a single run over all data. Thus, multi-grid does not worsen the performance – in terms of runtime efficiency – of our program. In the following, we will shortly describe the basic ingredients of our additive multi-grid algorithms, which turned out to perform well for adaptively refined grids in [5].

In contrast to relaxation methods, the multi-grid method works on the whole set of hierarchical data, not only on the finest level. As described in the previous section, data are processed in a top-down, depth-first order (see also Figure 3.3 for an example of the processing order of cells). Thus, we can not finish one step (smoothing, interpolation, or restriction) at all grid points of a level before we proceed to the next level. The cycle function is called recursively for all (nine in the case of two-dimensional Peano-curves) sub-cells of a coarse cell and, thus, whenever we enter a cell, the whole work for one multi-grid cycle within this cell has to be finished before we enter the next cell of the same level. Therefore, the natural choice in terms of our algorithm is an additive multigrid method where smoothing is done simultanuously on all levels[1]. Descending the space-tree[2] within the top-down depth-first run over the grid along the Peano-curve, we interpolate coarse grid values to the finer grids in order to achieve nodal values on the finest grid. At the finest level we compute the residual and apply the smoother. Ascending the space-tree, we restrict the residual to the coarse levels and apply the smoother on the coarse levels.

Remark 1: We have to take care that residuals are correctly transported from fine to coarse grid points: since we restrict the residual each time we visit a fine grid point (and *not* only when the whole residual is computed), we may only restrict the cell-part of the respective residual and not the accumulated residual. This complication requires some additional local variables for the cell-parts of the residual.

[1] The implementation of a mutlplicative multi-grid method is possible but a little more technical

[2] In contrast to 'descending' in the common context of multi-grid methods, descending in our context means proceeding from the coarsest level to the finest.

Remark 2: As we use a *hierarchical* generating system for u_h instead of nodal values, coarse grid corrections are simply done by smoothing the coarse-grid coefficients of the function representations. The transport to fine grid values is done implicitly during dehierarchization of u_h.

4.3. Adaptively Refined Grids. If we use the processing order of cells and the construction of stacks described in the previous section, adaptively refined grids can also be handled in a very natural way without any further complications like for example special difference stencils at borders between differently refined areas.

Like for regular grids, we compute all operators in a cell-oriented way. Thus, the stencils used look the same for each cell (up to a potential scaling with the cell width h). Thus, in a first step, we end up with contributions to operators at all vertices of cells at the locally finest level. In particular, hanging nodes also have to carry function values and contributions to the operators. To eliminate hanging nodes from the system of equations (hanging nodes may not be considered as degrees of freedom!) we have to compute interpolated function values at hanging nodes before the operator evaluation and distribute the respective cell-part of the operators to the neighbouring coarse grid points by some suitable restriction. With this restriction, we automatically get correct operator values at all coarse grid points. From the algorithmic point of view, this interpolation and restriction works completely analogously to the multi-grid interpolation and restriction.

5. Examples. To point out the potential of our algorithm, we show some simple examples and achieved results and make some remarks on the efficiency of our program concerning storage requirements, processing time and cache behavior. Note that up to now we work with an experimental code which is not optimized at all yet. Thus, absolute values like computing time will be improved further and our focus here is only on the cache behavior and the qualitative dependencies between the number of unknowns and the performance values like computing times.

5.1. Poisson Equation on the Unit Square. As a first test, we solve the two-dimensional Poisson equation on the unit-square with homogeneous Dirichlet boundary conditions:

$$\triangle u(\mathbf{x}) = -2\pi^2 \sin(\pi x)\sin(\pi y), \quad \forall\, \mathbf{x} = (x,y)^T \in \Omega =]0,1[\times]0,1[\tag{5.1}$$

$$u(\mathbf{x}) = 0 \quad \forall\, \mathbf{x} \in \partial\Omega \tag{5.2}$$

The exact solution of this problem is given by $u(\mathbf{x}) = \sin(\pi x) \cdot \sin(\pi y)$. To discretize the Laplace operator, we use the common Finite Element stencil

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The resulting system of linear equations was solved by an additive multi-grid method with bilinear interpolation and full-weighting as restriction operator. As criteria for termination of the iteration loop we took a value of $r_{max} = 10^{-5}$, where r_{max} is the maximum (in magnitude) of all corrections over all levels. Table 5.1 shows performance values obtained on a dual Intel XEON 2.4 GHz with 4 GB of RAM for regular grids with growing resolution.

Note that the number of iterations until convergence is independent of the resolution, which one would have expected for a multi-grid method. method, which can be shown by the analysis of cache misses and cache hits on the level 2 cache.

resolution	variables over all levels	iterations	processing time	CPU time	L2 cache misses per iteration	#real misses / #minimal misses
27×27	744	39	0.083s	0.070s	156	0.11
81×81	7,144	39	0.779s	0.760s	11,819	0.88
243×243	65,708	39	5,819s	5,800s	134,432	1.09
729×729	595,692	39	52,240s	52,130s	1,246,949	1.12
2187×2187	5,374,288	39	473,699s	472,410s	11,278,058	1.12

TABLE 5.1
Performance values for the two-dimensional Poisson equation on the unit square solved on a dual Intel XEON 2.4 GHz with 4 GB of RAM

Table 5.1 shows very high L2-hit-rates of at least 99,13% measured on an Intel XEON, which is a very high value for 'real' algorithms beyond simple programs for testing performance of a given architecture. Even more significant for the efficiency of our algorithm is the result of a comparison of the minimal number of necessary cache-misses and actually measured cache-misses: With the size s of the C struct of stack elements, the number n of degrees of freedom and the size cl of an L2 cache line on the used architecture we can guess a minimum $cm_{min} = \frac{n \cdot s}{cl}$ of cache misses per iteration, which has to occur if we read each grid point once per iteration producing $\frac{s}{cl}$ cache misses per point. In fact, grid points are typically used several times (in our case once by each of the four neighbouring cells) in our algorithm as well as in most FEM-algorithms. Thus, this minimum guess is assumed to be even too low. The entries of the last column in Table 5.1 are defined as $\frac{cm_{real}}{cm_{min}}$ where cm_{real} are the L2 cache misses per iteration simulated with `calltree`. As this rate is nearly one in our algorithm, we produce hardly more cache misses as if we used every grid point only once per iteration.

5.2. Poisson Equation on a Disk. The examples above show a very high performance on the L2-Cache for full grids, but the major advantage of our method is that this holds also for adaptive grids and on more complicated geometries. To show this at least for a simple example, we consider the two-dimensional Poisson equation with inhomogeneous Dirichlet boundary conditions on a disk with radius one:

$$\triangle u(\mathbf{x}) = -2\pi^2 \sin(\pi x) \sin(\pi y), \quad \forall\, \mathbf{x} = (x, y)^T \text{ with } \|\mathbf{x}\| < 1 \tag{5.3}$$

$$u(\mathbf{x}) = \sin(\pi x) \cdot \sin(\pi y) \quad \forall\, \mathbf{x} \in \partial\Omega \tag{5.4}$$

Figure 5.1 shows the representation of the disk with the help of an adaptive grid gained by a geometry-oriented coarsening of an initial grid with 81×81 cells.

For our numerical tests, we used different adaptive grids gained by local coarsening strategies starting from an initial grid with 729×729 cells. To get a sufficient accuracy near the boundary, we did not allow any coarsening of boundary cells.

The stopping criterion for the iterative solver was again $|r_{max}| < 10^{-5}$. In Table 5.2, the column 'costs per variable' shows the amount of time needed per variable and per iteration. These values are nearly constant or even better for some adaptive

variables	% of full grid	computing time	L2-hitrate	iter	ms per iteration	ms per variable per iteration
66220	15.094%	39.68s	99.28%	287	138.26	0.002088
85289	19.440%	50.05s	99.28%	287	174.39	0.002045
101146	23.055%	58.95s	99.26%	287	205.40	0.002031
117672	26.822%	67.14s	99.25%	285	235.58	0.002002
156316	35.630%	87.94s	99.24%	286	307.48	0.001967
175578	40.021%	97.10s	99.20%	286	339.51	0.001934
351486	80.116%	179.36s	99.11%	280	640.57	0.001822
438719	100.00%	250.23s	99.16%	281	890.50	0.002030

TABLE 5.2
performance values for the two-dimensional Poisson equation on a disk solved on a dual Intel XEON 2.4 GHz with 4 GB of RAM

grids than for the full grid. In Figure 5.2 you see the value 'ms per iteration' plotted against the number of variables with crosses. The solid line is the corresponding line from the origin through the full grid point $(438719, 768.934)$. We see that we have a linear dependency between the number of variables/grid points and the computing time, no matter whether we have a regular full grid or an adaptively refined grid. This is a remarkable result as we can conclude from this correlation that in our method variables on coarser grids do not have higher costs than the ones on the finest level.

6. Extension to 3D. In the previous sections, we only considered the two-dimensional case. But, as mentioned in the introduction, our concepts can be generalized in a very natural way to three or even more dimensions. In this section, we want to give a few preliminary results on the 3D case. The basic concepts are the same as in the two-dimensional case, but to achieve an efficient implementation, we introduced some changes and/or enhancements in the concrete realization.

The first – and obvious – change is that we need 3D in- and output stacks and $2D$-, $1D$- and $0D$-stacks (corresponding to faces, edges and vertices of a cube) during

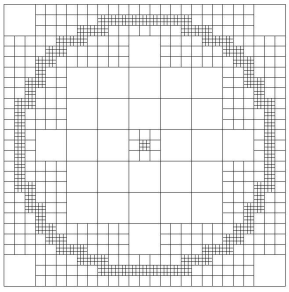

FIG. 5.1. *Representation of the disk with the help of an adaptive grid gained by a geometry oriented coarsening of a* 81×81 *grid*

our run through the space tree instead of $1D$- and $0D$-stacks only. In addition, we use twelve colors for the $0D$- and $1D$-stacks and six colors for the $2D$-stacks.

Another interesting aspect in 3D is that we replace the direct refinement of a cell by the introduction of 27 sub-cells by a dimension recursive refinement: A cell is cut into three "plates" in a first step, each of these plates is cut into three "bars", and, finally, each bar into three "cubes" (see Figure 6.1). This reduces the number of different cases dramatically and can even be generalized to arbitrary dimensions, too [17]. Detailed despcriptions of the 3D implementations including dynamical adaptivity, higher order discritizations and full multi-grid methods are given in [24, 20, 11].

As can be seen from the following example, the performance of our algorithm carries over from 2D to 3D. The L2-hit-rates even get better by an order of magnitude. Only the absolute computing time per variable and per iteration becomes worse. Both phenomena can easily be explained for the most part by the more complicated discretization stencil with more non-zero entries.

We solve the three-dimensional Poisson equation

$$\triangle u((\mathbf{x})) = 1 \tag{6.1}$$

on a star-shaped domain, a sphere (see Figure 6.2) and the unit cube with homogeneous boundary conditions. The Laplace operator is discretized by the common Finite Difference stencil and – analoguously to the 2D case – we use an additive multi-grid method with trilinear interpolation and full-weighting as restriction operator. The termination criterion was $|r_{max}| \leq 10^{-5}$. Tables 6.1 and 6.2 show performance values obtained on a dual Intel XEON 2.4 GHz with 4 GB of RAM for adaptive grids (see also Figure 6.2 for a part of the – adaptively coarsened – $243 \times 243 \times 243$ grids). For the star-shaped domain, the grid-coarsening was restricted to the domain part outside the computational domain. Table 6.3 shows another important advantage of our algorithm, so we could handle a huge number of degrees of freedom with the

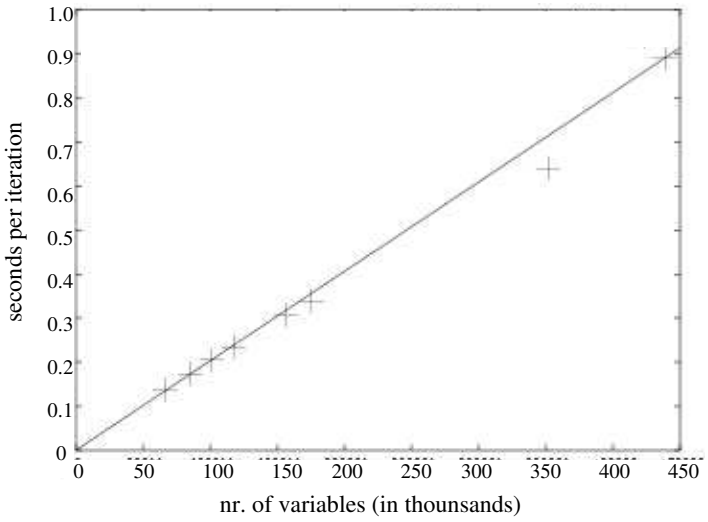

FIG. 5.2. *Linear dependency: number of variables – computing time*

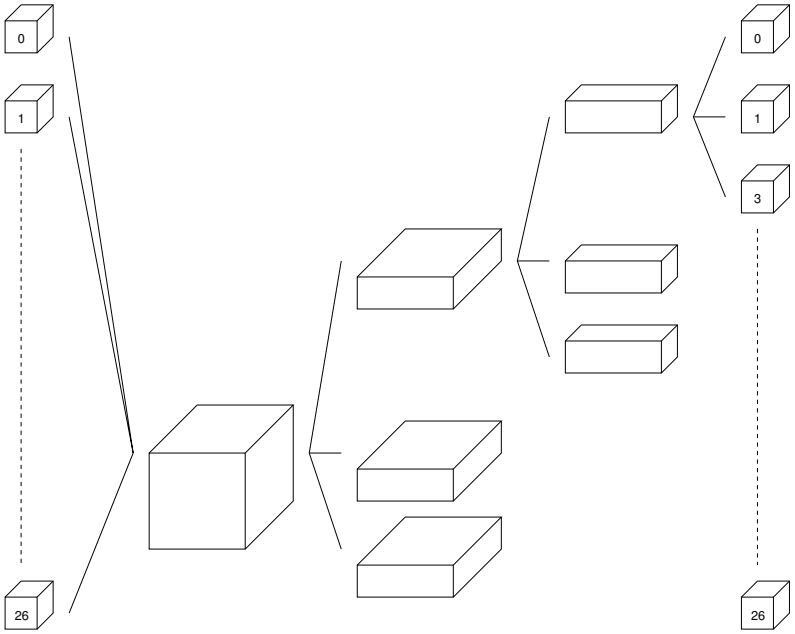

FIG. 6.1. *Dimension recursive refinement of a cell*

resolution of the full grid	variables	storage requirements	computing time	L2-hit-rate	iter	ms per variable per iteration
$81 \times 81 \times 81$	18,752	0.7MB	179.73s	99.99%	96	0.0998
$243 \times 243 \times 243$	508,528	18MB	5,228.3s	99.99%	103	0.0998
$243 \times 243 \times 243$	508,528 (adp.)	9MB	2,405.86s	99.99%	103	0.0459
$729 \times 729 \times 729$	13,775,328 (adp.)	230MB	63,502.14s	99.98%	103	0.0448

TABLE 6.1

Performance values for the three-dimensional Poisson equation on a star-shaped domain solved on a dual Intel XEON 2.4 GHz with 4 GB of RAM

same performance (e.g. over 400,000,000 variables) and with a very small amount of memory.

7. Conclusion. In this paper we presented a method combining several features for the solution of partial differential equations:

- Adaptivity
- Efficient multilevel algorithm
- Geometrical flexibility
- Suitability for modern computer architectures (cache awareness and parallelisability combined with load balancing)
- Suitability for general discretized systems of partial differential equations

The present implementations include the first three of these aspects and part of the fourth aspect. But the implementation of the other aspects is not difficult. The

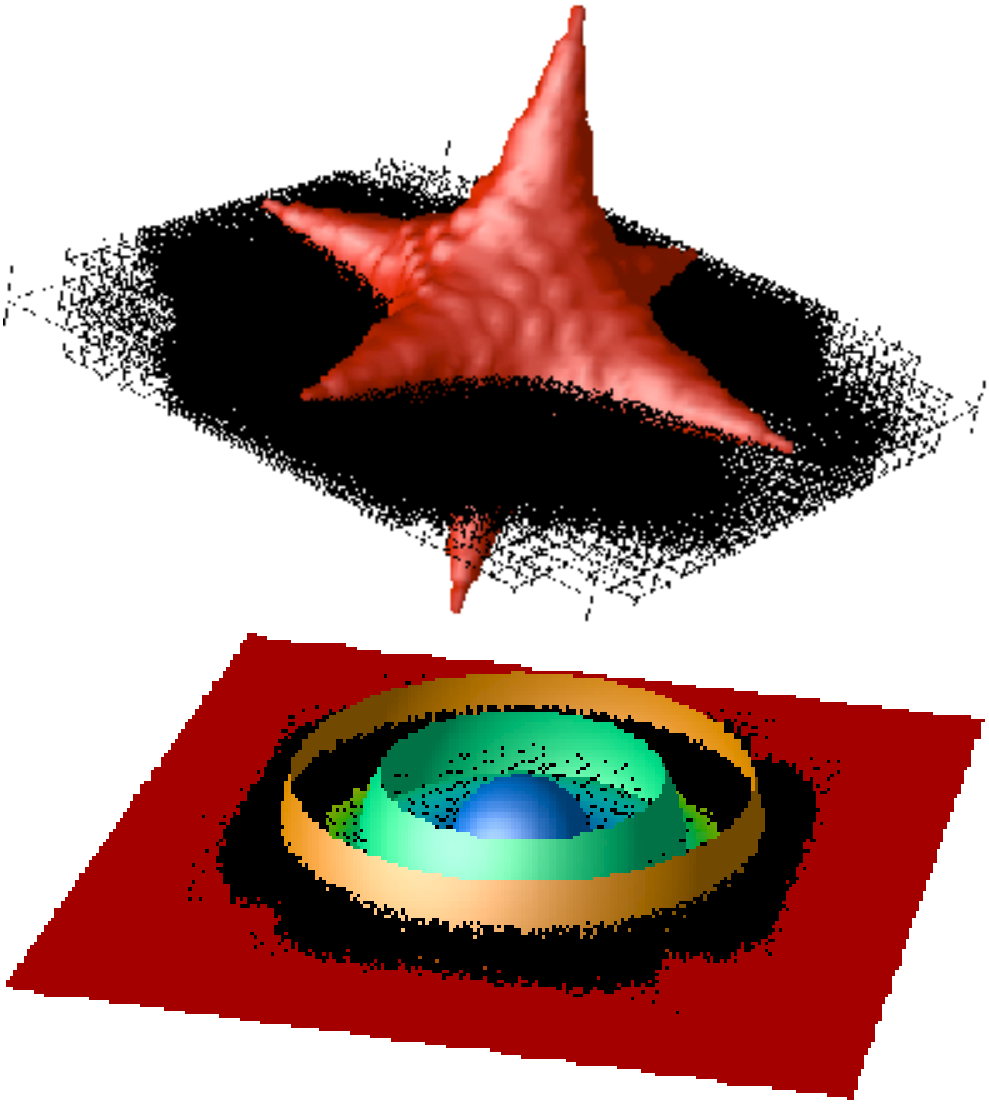

FIG. 6.2. *Representation of the star-shaped domain and a sphere with the help of adaptive grids gained by a geometry oriented coarsening of a* $243 \times 243 \times 243$ *grid*

parallelization of the method follows step by step the approach of Zumbusch ([32]) who has already successfully used space-filling curves for the parallelization of PDE-solvers. The generalization to systems of PDEs is also straightforward. [31] implemented a version solving the 2D Navier-Stokes equations. A thesis on problems with non-constant coefficients will be published within the next months.

As all codes used are still in an experimental state in particular without any run-time optimization, we will publish further detailed performance results in subsequent papers.

As a further remark we want to mention that space-filling curves can also be used e. g. for the computation of the product of a full matrix with a full vector or a full matrix with a full matrix where we also obtain a reduction of cache misses [3].

resolution of the full grid	variables	storage requirements	computing time	L2-hit-rate	iter	ms per variable per iteration
$81 \times 81 \times 81$	72,576	1MB	167.34s	99.99%	88	0.0262
$243 \times 243 \times 243$	1,969,872	24MB	4,949.87s	99.98%	96	0.0262
$243 \times 243 \times 243$	900,024 (adp.)	6MB	773.21s	99.97%	97	0.0089
$729 \times 729 \times 729$	24,231,440 (adp.)	150MB	20,127.97s	99.94%	100	0.0083

TABLE 6.2
Performance values for the three-dimensional Poisson equation on a sphere solved on a dual Intel XEON 2.4 GHz with 4 GB of RAM

resolution of the full grid	variables	storage requirements	computing time	L2-hit-rate	iter	ms per variable per iteration
$81 \times 81 \times 81$	530,096	3MB	98.09s	99.96%	42	0.0044
$243 \times 243 \times 243$	14,702,584	76MB	2,641.86s	99.91%	42	0.0043
$729 \times 729 \times 729$	400,530,936	2GB	72,307.26s	99.90%	42	0.0043

TABLE 6.3
Performance values for the three-dimensional Poisson equation on the unit cube solved on a dual Intel XEON 2.4 GHz with 4 GB of RAM

REFERENCES

[1] M. J. AFTOSMIS, M. J. BERGER, AND G. ADOMAVIVIUS, *A Parallel Multilevel Method for adaptively Refined Cartesian Grids with Embedded Boundaries*, AIAA Paper, 2000.

[2] M. BADER, H.-J. BUNGARTZ, A. FRANK, AND R. MUNDANI, *Space tree structures for pde solution*, in P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, eds., Proceedings of ICCS 2002, part 3, volume 2331 of LNCS. Springer, 2002.

[3] M. BADER AND C. ZENGER, *Cache oblivious matrix multiplication using an element ordering based on the Peano curve*, submitted to Linear Algebra and its Applications (Elsevier).

[4] R. A. BRUALDI AND B. L. SHADER, *On sign-nonsingular matrices and the conversion of the permanent into the determinant*, in Applied Geometry and Discrete Mathematics, The Victor Klee Festschrift, P. Gritzmann and B. Sturmfels, eds., American Mathematical Society, Providence, RI, 1991, pp. 117–134.

[5] F. A. BORNEMANN, *An adaptive multilevel approach to parabolic equations III: 2D error estimation and multilevel preconditioning*, IMPACT Comput. Sci. Eng. 4, 1992, pp. 1-45.

[6] BRAESS, *Finite Elements. Theory, Fast Solvers and Applications in Solid Mechanics*, Cambridge University Press, 2001.

[7] E. D. DEMAINE, *Cache-Oblivious Algorithms and Data Structures*, in Lecture Notes from the EEF Summer School on Massive Data Sets, Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27-July 1, 2002.

[8] S. CHATTERJEE, S. SEN, *Cache-Efficient Matrix Transposition*, in Proceedings of HPCA-6, pages 195–205, Toulouse, France, Jan. 2000.

[9] S. CHATTERJEE, A. R. LEBECK, P. K. PATNALA, AND M. THOTTETHODI, *Recursive array layouts and fast parallel matrix multiplication*, in proceedings of Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, Saint-Malo, France, 1999, pp. 222-231.

[10] W. CLARKE, *Key-based parallel adaptive refinement for FEM*, bachelor thesis, Australian National Univ., Dept. of Engineering, 1996.

[11] N. Dieminger, Kriterien fr die Selbstadaption cache-effizienter Mehrgitteralgorithmen, diploma thesis, Institut für Informatik, TU München, 2005.
[12] M. Frigo, C. E. Leierson, H. Prokop, and S. Ramchandran, *Cache-oblivious algorithms*, in Proceedings of the 40th Annual Sympoisium on Foundations of Computer Science, pages 285-297, New York, October 1999.
[13] M. Griebel, Multilevelverfahren als Iterationsmethoden über Erzeugendensystemen, Habilitationsschrift, TU München, 1993.
[14] M. Griebel, S. Knapek, G. Zumbusch, and A. Caglar, *Numerische Simulation in der Moleküldynamik. Numerik, Algorithmen, Parallelisierung, Anwendungen.* Springer, Berlin, Heidelberg, 2004.
[15] M. Griebel and G. W. Zumbusch, *Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves*, Parallel Computing, 25:827-843, 1999.
[16] M. Griebel and G. Zumbusch, *Hash based adaptive parallel multilevel methods with space-filling curves*, in Horst Rollnik and Dietrich Wolf, editors, NIC Symposium 2001, volume 9 of NIC Series, ISBN 3-00-009055-X, pages 479-492, Germany, 2002. Forschungszentrum Jülich.
[17] J. Hartmann, Entwicklung eines cache-optimalen Finite-Element-Verfahrens zur Lösung d-dimensionaler Probleme, diploma thesis, Institut für Informatik, TU München, 2005.
[18] M. Griebel and G. Zumbusch, *Hash-storage techniques for adaptive multilevel solvers and their parallelization*, in Domain decomposition methods 10. The 10th int. conf., Boulder, J. Mandel, C. Farhat, and X.-C. Cai, eds., vol. 218 of Contemp. Math., AMS, Providence, Rhode Island, 1996, pp. 271-278.
[19] M. Kowarschik, C. Wei, *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, in Proceedings of the GI-Dagstuhl Forschungsseminar: Algorithms for Memory Hierarchies, Lecture Notes in Computer Science (LNCS), Vol. 2625, Springer, 2003.
[20] A. Krahnke, Adaptive Verfahren höherer Ordnung auf cache-optimalen Datenstrukturen für dreidimensionale Probleme, doctoral thesis, Institut für Informatik, TU München.
[21] J. T. Oden, A. Para, and Y. Feng, *Domain decomposition for adaptive hp finite element methods*, in Domain decomposition methods in scientific and engineering computing. Proc. of the 7th int. conf. on domain decomposition, Pennsylvania State Universitiy, D. E. Keyes and J. Xu, eds., vol. 180 of Contemp. Math., AMS, Providence, Rhode Island, 1994, pp. 203-214.
[22] A. K. Patra, J. Long, A. Laszloff, *Efficient Parallel Adaptive Finite Element Methods Using Self-Scheduling Data and Computations*, HiPC, 1999, pp. 359-363
[23] M. Parashar and J. C. Browne, *On partitioning dynamic adaptive grid hierarchies*, in Proc. of the 29th Annual Hawaii Int.Conf. on System Sciences, 1996, pp. 604-613.
[24] M. Pögl, Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens fr große Probleme, doctoral thesis, Institut für Informatik, TU München, 2004.
[25] H. Prokop, *Cache-Oblivious Algorithms*, Master Thesis, Massachusetts Institute of Technology, 1999.
[26] S. Roberts, S. Klyanasundaram, 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, B. J. Noye, M. D. Teubner, and A. W. Gill, eds. World Scientific, Singapore, 1998, p. 577-584.
[27] H. Sagan, *Space-Filling Curves*, Springer-Verlag, New York, 1994.
[28] R. J. Stevens, A. F. Lehar, and F. H. Preston, *Manipulation and Presentation of Multidimensional Image Data Using the Peano Scan*, IEEE Trans. Pattern An. and Machine Intelligence, Vol PAMI-5, 1983, pp. 520-526.
[29] O. Temam, *Investigating Optimal Local Memory Performance*, in Proc. ACM int. Conference on Architectural Support for Programming Languages and Operating Systems, San Diego, California, USA, 1998.
[30] L. Velho, J. de Miranda Gomes, *Digital Halftoning with Space-Filling Curves*, Computer Graphics, Vol. 25, 1991, pp. 81-90.
[31] T. Weinzierl, Eine cache-optimale Implementierung eines Navier-Stokes Lösers unter besonderer Berücksichtigung physikalischer Erhaltungssätze, diploma thesis, Institut für Informatik, TU München, 2005.
[32] G. Zumbusch, *Adaptive Parallel Multilevel Methods for Partial Differential Equations*, Habilitationsschrift, Universität Bonn, 2001.
[33] G. W. Zumbusch, *On the quality of space-filling curve induced partitions*, Z. Angew. Math. Mech., 81:25-28, 2001. Suppl. 1, also as report SFB 256, University Bonn, no. 674, 2000.