# A Cache-Aware Algorithm for PDEs on Hierarchical Data Structures

Frank Günther, Miriam Mehl, Markus Pögl, and Christoph Zenger

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

**Abstract.** A big challenge in implementing up to date simulation software for various applications is to bring together highly efficient mathematical methods on the one hand side and an efficient usage of modern computer archtitectures on the other hand. We concentrate on the solution of PDEs and demonstrate how to overcome the hereby occuring quandary between cache-efficiency and modern multilevel methods on adaptive grids. Our algorithm is based on stacks, the simplest possible and thus very cache-efficient data structures.

## 1 Introduction

In most implementations, competitive numerical algorithms for solving partial differential equations cause a non-negligible overhead in data access and, thus, can not exploit the high performance of processors in a satisfying way. This is mainly caused by tree structures used to store hierarchical data needed for methods like multi-grid and adaptive grid refinement.

We use space-filling curves as an ordering mechanism for our grid cells and – based on this order – to replace the tree structure by data structures which are processed linearly. For this, we restrict to grids associated with space-trees (allowing local refinemt) and (in a certain sense) local difference stencils. In fact, the only kind of data structures used in our implementation is a fixed number of stacks. As stacks can be considered as the most simple data structures used in Computer Science allowing only the two basic operations push and pop[1], data access becomes very fast – even faster than the common access of non-hierarchical data stored in matrices – and, in particular, cache misses are reduced considerably. Even 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 and doesn't worsen the cache efficiency.

In literature, space-filling curves are a well-known device to construct efficient grid partitionings for data parallel implementations of the numerical solution of partial differential equations [13–16, 19, 23, 24]. 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 like e.g. matrix

[1] push puts data on top of a pile and pop takes data from the top of a pile.

J. Dongarra, K. Madsen, and J. Waśniewski (Eds.): PARA 2004, LNCS 3732, pp. 874–882, 2005.

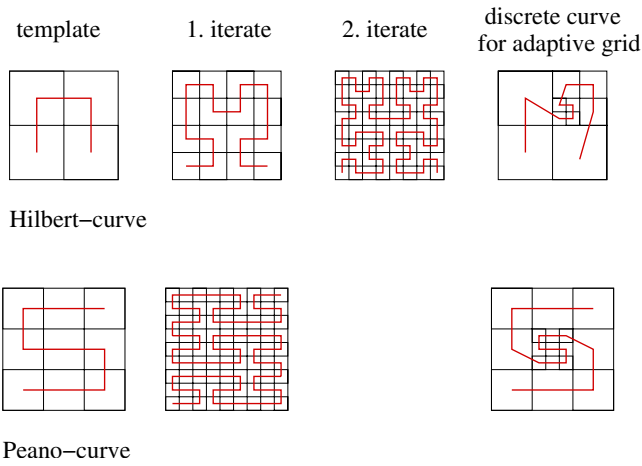


**Fig. 1.** Generating templates, first iterates and discrete curve on an adaptively refined grid for two-dimensional Hilbert- and Peano-curves

transposition [5] or matrix multiplication [6]. We enhance this effect by constructing stacks for which we can even *completely avoid* 'jumps' in the adresses instead of only reducing their probability or frequency.

## 2 Space-Filling Curves and Stacks

As we look for an odering mechanism for the cells of a multilevel adaptive rectangular grid based on a space tree, we restrict our attention to a certain class of space-filling curves[2], namely 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. See figure 1 for some iterates of the two-dimensional Hilbert- and Peano-curves, which are prominent representatives of this class of curves. As can be seen, the iterate we use in a particular part of our domain depends on the local resolution.

Now, these iterates of the space-filling curves associated to the given grid – also called discrete space-filling curves – and not the space-filling curve itself[3] define the processing order of grid cells. To apply an operator-matrix to the vector of unknowns, we process the cells strictly in this order and perform all computations in a cell-oriented way, which means that in each cell, we evaluate only those parts of the corresponding operator that can be computed solely with the help of the cell's own information and, in particular, without adressing any information of neighbouring cells. This method is standard for finite element methods (see e.g. [4]), but can be generalized for 'local'

[2] For general information on space-filling curves see [20].

[3] The space-filling curve itself can be interpreted as the limit of the recursive refinement procedure and is – in contrast to the iterates – *not* injective.

discrete operators, which means that for the evaluation in one grid point, only direct neighbours are needed[4].

To get more concrete, we look at a space-tree with function values located at the vertices of cells in the following. To apply the common five-point-stencil for the Laplacian operator, for example, we have to decompose the operator into four parts associated to the four neighbouring cells of a gridpoint[5]:

$$\Delta_h u_h|_{i,j} =$$

$$\frac{u_{i-1,j}+u_{i,j-1}-2u_{i,j}}{2h^2} + \frac{u_{i+1,j}+u_{i,j-1}-2u_{i,j}}{2h^2} + \frac{u_{i+1,j}+u_{i,j+1}-2u_{i,j}}{2h^2} + \frac{u_{i-1,j}+u_{i,j+1}-2u_{i,j}}{2h^2}.$$

$$\begin{array}{ccc}
\cdot - \cdot - \cdot \\
| \quad | \quad | \\
\frac{1}{2} - -1 - \cdot \\
| \quad | \quad | \\
\cdot - \frac{1}{2} - \cdot
\end{array}
\qquad
\begin{array}{ccc}
\cdot - \cdot - \cdot \\
| \quad | \quad | \\
\cdot - -1 - \frac{1}{2} \\
| \quad | \quad | \\
\cdot - \frac{1}{2} - \cdot
\end{array}
\qquad
\begin{array}{ccc}
\cdot - \frac{1}{2} - \cdot \\
| \quad | \quad | \\
\cdot - -1 - \frac{1}{2} \\
| \quad | \quad | \\
\cdot - \cdot - \cdot
\end{array}
\qquad
\begin{array}{ccc}
\cdot - \frac{1}{2} - \cdot \\
| \quad | \quad | \\
\frac{1}{2} - -1 - \cdot \\
| \quad | \quad | \\
\cdot - \cdot - \cdot
\end{array}$$

Of course, in each cell, we evaluate the cell-parts of the operator values for all four vertices all at once. Thus, we have to construct stacks such that all data associated to the respective cell-vertices lie on top of the stacks when we enter the cell during our run along the discrete space-filling curve. We will start with a simple regular two-dimensional grid illustrated in figure 2 and nodal data. If we follow the Peano-curve, we see that during our run through the lower part of the domain, data points on the middle line marked by 1 to 9 are processed linearly from the left to the right, during the subsequent run through the upper part vice versa from the right to the left. Analogously, all other grid points can be organized on lines which are processed linearly forward and, afterwards, backward. For the Peano-curve and a regular grid, this can be shown to hold for arbitrarily fine grids, as well.

As these linearly processed lines work with only two possibilities of data access, namely push (write data at the next position in the line) and pop (read data from the actual position of the line), they correspond directly to our data stacks. We can even integrate several lines in one stack, such that it is sufficient to have two stacks: $1D_{black}$ for the points on the right-hand-side of the curve and $1D_{red}$ for the points at the left-hand-side of the curve. The behavior of a point concerning read and write operations on stacks can be predicted in a deterministic[6] way. We only have to classify the points as 'inner points' or 'boundary points' and respect the (local) progression of the curve.

[4] In addition, in the case of adaptively refined grid, the cell-oriented view makes the storage of specialized operators at the boundary of local refinements redundant as a single cell computes its contribution to the overall operators without taking into consideration the refinement depth of surrounding cells.

[5] If we want to minimze the number of operations, this equal decomposition is of course not the optimal choice, at least not for equidistant grids and constant coefficients.

[6] In this context, deterministic means depending only on locally available informations like the local direction of the Peano-curve.

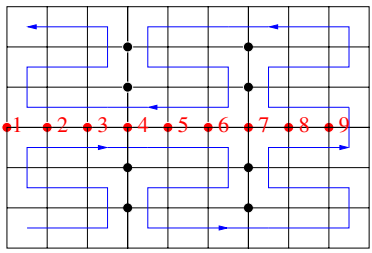


**Fig. 2.** Example in 2D using the Peano-curve

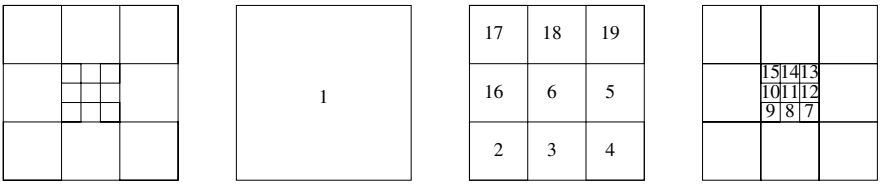


**Fig. 3.** Example for the order of hierarchical cells defined by the discrete Peano-curve

We use the Peano-curve instead of the Hilbert-curve since in 3D the Peano-curve guarantees the inversion of the processing order of grid points along a plane in the domain if we switch from the domain on one side of the plane to the domain on the other side. This property is essential for our stack concept and we could not find any Hilbert-curve fulfilling it. There are good reasons to assume that this is not possible at all.

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 shows only when we look at adaptively refined grids and – in the general case – hierarchical data in connection with generating systems [11]. This leads to more than one degree of freedom per grid point and function. Even in this case, the space-filling curve defines a linear order of cells respecting all grid levels: the cells are visited in a a top-down depth-first process reflecting the recursive definition of the curve itself (see figure 3 for an example).

In our stack-context, we have to assure that even then predictable and linear data access to and from stacks is possible. In particular, we have to assure that grid points of different levels lie on the stacks in the correct order and do not "block" each other. We could show that it is sufficient to use four colors representing four different stacks of the same type and a second type of stack, called $0D$ stack. Thus, we end up with a still fixed – and in particular independent of the grid resolution – number of eight stacks (for details see [10]).

To be able to process data on a domain several times – as needed for each iterative solver – without loss of efficiency, we write all points to a so called 2D- or plain-stack 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 very efficiently. We apply this repeatedly and, thus, change the processing direction of grid cells after each iteration.

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!)[7] because of the fact, that using linear stacks is a good idea in conjunction with modern processors prefetching techniques. An additional gain of using our concept is the minimal storage cost for administrational data. We can do completely without any pointer to neighbours and/or father- and son-cells. Instead, we only have to store one bit for refinement informations and one bit for geometrical information (in- or outside the computational domain) per cell.

## 3 Results

To point out the potential of our algorithm, we show some first results for simple examples with the focus on the efficiency 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.

### 3.1 Two-Dimensional Poisson Equation

As a first test, we solve the two-dimensional Poisson equation on the unit-square and on a two-dimensional disc with homogeneous Dirichlet boundary conditions:

$$\triangle u(\mathbf{x}) = -2\pi^2 \sin(\pi x)\sin(\pi y), \quad \forall\, \mathbf{x} = (x,y)^T \in \Omega \tag{3.1}$$

$$u(\mathbf{x}) = \sin(\pi x)\cdot\sin(\pi y) \quad \forall\, \mathbf{x} \in \partial\Omega \tag{3.2}$$

The exact solution of this problem is given by $u(\mathbf{x}) = \sin(\pi x)\cdot\sin(\pi y)$. To discretize the Laplacian operator, we use the common Finite Element stencil arising from the use of bilinear basis functions. The resulting system of linear equations was solved by an additive multi-grid method with bilinear interpolation and full-weighting as restriction operator. As criterion 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.

On the unit square, we used regular grids with growing resolution. On the disc, we used different adaptive grids gained by local coarsening strategies starting from an initial grid with $729 \times 729$ cells. To get a sufficient accuracy near the boundary, we did not allow any coarsening of boundary cells. Tables 1 and 2 show performance values obtained on a dual Intel XEON 2.4 GHz with 4 GB of RAM.

Note that the number of iterations until convergence is in both cases independent from the resolution, which one would have expected for a multi-grid method. The analysis of cache misses and cache hits on the level 2 cache gives a good hint on the high

[7] Algorithms wich are cache-aware by concept without detailed knowledge of the cache parameters are also called cache-oblivious [8, 9, 18].

**Table 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

| resolution | iterations | L2 cache misses per it. | rate |
|---|---|---|---|
| $243 \times 243$ | 39 | 134,432 | 1.09 |
| $729 \times 729$ | 39 | 1,246,949 | 1.12 |
| $2187 \times 2187$ | 39 | 11,278,058 | 1.12 |

**Table 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

| variables | % of full grid | L2-hitrate | iter | cost per variable |
|---|---|---|---|---|
| 66220 | 15.094% | 99.28% | 287 | 0.002088 |
| 101146 | 23.055% | 99.26% | 287 | 0.002031 |
| 156316 | 35.630% | 99.24% | 286 | 0.001967 |
| 351486 | 80.116% | 99.11% | 280 | 0.001822 |
| 438719 | 100.00% | 99.16% | 281 | 0.002030 |

efficiency of our method. But for a more realistic judgement of the efficieny, the very good relation between the number of actual cache-misses and the minimal number of necessary cache-misses caused by the unavoidable first loading of data to the cache is more significant:

With the size $s$ of a stack element, the number $n$ of degrees of freedom and the size $cl$ of a 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 four times in our algorithm as well as in most FEM-algorithms. Thus, this minimum guess is assumed to be even too low. 'Rate' in table 1 is defined as $\frac{cm_{real}}{cm_{min}}$ where $cm_{real}$ are the L2 cache misses per iteration simulated with `calltree` [25]. As this rate is nearly one in our algorithm, we can conclude that we produce hardly no needless cache misses at all. In addition, we get a measured L2-hit-rate of at least 99,13%, using hardware performance counters [26] on an Intel XEON, which is a very high value for 'real' algorithms beyond simple programs for testing performance of a given architecture.

In table 2, the column 'costs per variable' shows the amount of time needed per variable and per iteration. These values are nearly constant. Thus, we see that we have a linear dependency between the number of variables and the computing time, no matter if 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, adaptivity doesn't any additional costs.

### 3.2 Three-Dimensional Poisson Equation

In the previous sections, we only considered the two-dimensional case. But our concepts can be generalized in a very natural way to three or even more dimensions. Here, we

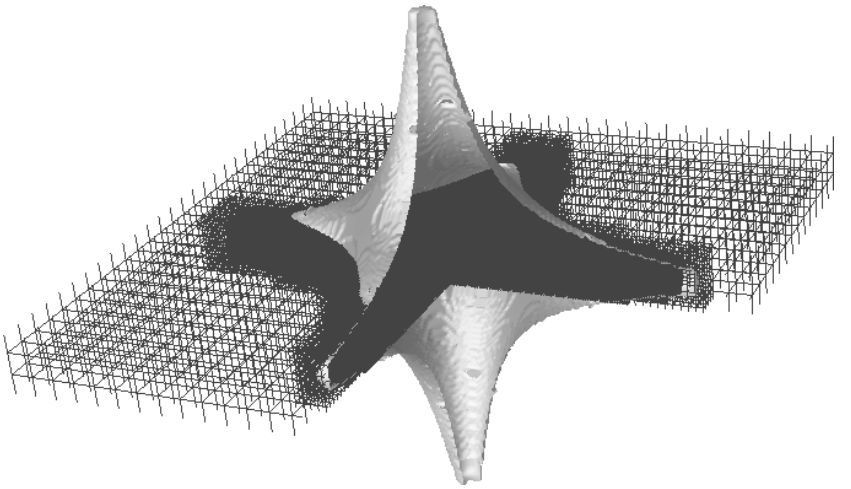

**Fig. 4.** Representation of the star-shaped domain with the help of adaptive grids gained by a geometry oriented coarsening of a $243 \times 243 \times 243$ grid

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 [17].

The first – and obvious – change is that we need 3D in- and output stacks and 2D, 1D and 0D stacks during our run through the space tree instead of 1D and 0D stacks only. In addition, we use 8 colors for the 0D, 12 colors for the 1D stacks, and 6 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". This reduces the number of different cases dramatically and can even be generalized to arbitrary dimensions, too.

We solve the three-dimensional Poisson equation

$$\triangle u((\mathbf{x})) = 1 \tag{3.3}$$

on a star-shaped domain (see figure 4) 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 $|r_{max}| \leq 10^{-5}$.

Table 3 shows performance values obtained on a dual Intel XEON 2.4 GHz with 4 GB of RAM for adaptively refined grids. As can be seen, all important performance properties like multi-grid performance and cache efficiency carry over from the 2D case. The rate between the number of real cache misses and the minimal number of expected cache misses was measured for a different example (poisson equation in a sphere) and stays also in the range of 1.10. Thus, even in 3D, the essential part of the cache misses is really necessary and cannot be avoided by any algorithm.

**Table 3.** 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

| max. resolution | variables | storage req. | L2-hit-rate | iter | cost per variable |
|---|---|---|---|---|---|
| $81 \times 81 \times 81$ | 18,752 | 0.7MB | 99.99% | 96 | 0.0998 |
| $243 \times 243 \times 243$ | 508,528 | 9MB | 99.99% | 103 | 0.0459 |
| $729 \times 729 \times 729$ | 13,775,328 | 230MB | 99.98% | 103 | 0.0448 |

## 4 Conclusion

In this paper we presented a method combining important mathematical features for the numerical solution of partial differential equations – adaptivity, multi-grid, geometrical flexibility – with a very cache-efficient way of data organization tailored to modern computer architectures and suitable for general discretized systems of partial differential equations (yet to be implemented). The parallelization of the method follows step by step the approach of Zumbusch [23] who has already used successfully space-filling curves for the parallelization of PDE-solvers. The generalization to systems of PDEs is also straightforward. With an experimental version of our programs we already solved the non-stationaray Navier-Stokes equations in two dimensions, but the work is still in progress and shall be published in the near future. The same holds for detailed performance results for various computer architectures.

## 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. R.A. Brualdi and B.L. Shader. On sign-nonsingular matrices and the conversion of the permanent into the determinant. In P. Gritzmann and B. Sturmfels, editors. Applied Geometry and Discrete Mathematics, *The Victor Klee Festschrift*, pages 117-134, Providence, RI, 1991. American Mathematical Society.
3. F.A. Bornemann. An adaptive multilevel approach to parabolic equations III: 2D error estimation and multilevel preconditioning. *IMPACT Computational Science and Engeneering*, 4:1-45, 1992.
4. Braess. Finite Elements. Theory, Fast Solvers and Applications in Solid Mechanics. Cambridge University Press, 2001.
5. S. Chatterjee and S. Sen. Chache-Efficient Matrix Transposition. In *Proceedings of HPCA-6*, pages 195–205, Toulouse, France, January 2000.
6. 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*, pages 222-231, Saint-Malo, France, 1999.
7. W. Clarke. Key-based parallel adaptive refinement for FEM. Bachelor thesis, Australian National Univ., Dept. of Engineering, 1996.
8. 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, to appear.

9. 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.
10. F. Günther. Eine cache-optimale Implementierung der Finite-Elemente-Methode. Doctoral thesis, Institut für Informatik, TU München, 2004.
11. M. Griebel. Multilevelverfahren als Iterationsmethoden über Erzeugendensystemen. Habilitationsschrift, TU München, 1993.
12. M. Griebel, S. Knapek, G. Zumbusch, and A. Caglar. Numerische Simulation in der Moleküldynamik. *Numerik, Algorithmen, Parallelisierung, Anwendungen*, Springer, Berlin, Heidelberg, 2004.
13. 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.
14. M. Griebel and G. Zumbusch. Hash based adaptive parallel multilevel methods with space-filling curves. In H. Rollnik and D. Wolf, editors, *NIC Series*, 9:479-492, Germany, 2002. Forschungszentrum Jülich.
15. J.T. Oden, A. Para, and Y. Feng. Domain decomposition for adaptive *hp* finite element methods. In D.E. Keyes and J. Xu, editors, Domain decomposition methods in scientific and engineering computing, *proceedings of the 7th int. conf. on domain decomposition*, vol. 180 of Contemp. Math., pages 203-214, 1994, Pennsylvania State Universitiy.
16. A.K. Patra, J. Long, A. Laszloff. Efficient Parallel Adaptive Finite Element Methods Using Self-Scheduling Data and Computations. HiPC, pages 359-363, 1999.
17. M. Pögl. Entwicklung eines cache-optimalen 3D Finite-Element-Verfahrens für große Probleme. Doctoral thesis, Institut für Informatik, TU München, 2004.
18. H. Prokop. Cache-Oblivious Algorithms. Master Thesis, Massachusetts Institute of Technology, 1999.
19. S. Roberts, S. Klyanasundaram, M. Cardew-Hall, and W. Clarke. A key based parallel adaptive refinement technique for finite element methods. In B.J. Noye, M.D. Teubner, and A.W. Gill, editors, *Proc. Computational Techniques and Applications: CTAC '97*, pages 577-584, World Scientific, Singapore, 1998.
20. H. Sagan. Space-Filling Curves. Springer-Verlag, New York, 1994.
21. 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, pages 520-526, 1983.
22. L. Velho, J. de Miranda Gomes. Digital Halftoning with Space-Filling Curves. *Computer Graphics*, 25:81-90, 1991.
23. G.W. Zumbusch. Adaptive Parallel Multilevel Methods for Partial Differential Equations. Habilitationsschrift, Universität Bonn, 2001.
24. 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.
25. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior, *Proceedings of the 2004 International Conference on Computational Science, Krakow, Poland, June 2004*, vol. 3038, Lecture Notes in Computer Science (LNCS), Springer.
26. `http://user.it.uu.se/ mikpe/linux/perfctr/`