A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves[⋆]

Hans-Joachim Bungartz, Miriam Mehl, and Tobias Weinzierl

Technical University Munich, 85748 Garching, Germany
{bungartz, mehl, weinzier}@in.tum.de
http://www5.in.tum.de

Abstract. In this paper, we present a parallel multigrid PDE solver working on adaptive hierarchical cartesian grids. The presentation is restricted to the linear elliptic operator of second order, but extensions are possible and have already been realised as prototypes. Within the solver the handling of the vertices and the degrees of freedom associated to them is implemented solely using stacks and iterates of a Peano space–filling curve. Thus, due to the structuredness of the grid, two administrative bits per vertex are sufficient to store both geometry and grid refinement information. The implementation and parallel extension, using a space–filling curve to obtain a load balanced domain decomposition, will be formalised. In view of the fact that we are using a multigrid solver of linear complexity $\mathcal{O}(n)$, it has to be ensured that communication cost and, hence, the parallel algorithm's overall complexity do not exceed this linear behaviour.

1 Introduction

An important issue of a finite element code is to implement it in an efficient way. We want to examine four different aspects of efficiency: First of all the numerical efficiency covering all mathematical aspects, from modelling and discretization up to the solver. Second, there is the process integration efficiency, representing classical front– and back–end application integration tasks, such as adding a geometry input or embedding a flow solver into a fluid–structure interaction application. Furthermore, we distinguish between the implementation efficiency, regarding everything influencing the actual execution speed of a given program on a given platform, and parallel efficiency. The latter three often suggest the usage of cartesian grids, since then several implementation tasks are simplified. However, cartesian grids are not competitive for any real world application if they do not support adaptivity. On the other hand, with adaptivity the development of a well–suited traversal order, appropriate data structures, and a data access scheme is not a trivial task anymore.

In fact, many multigrid — i.e. numerically efficient — codes suffer from an inefficient implementation, integration, and parallelisation. We want to address

[⋆] This work has partially been funded by DFG's research unit FOR493 and the DFG project HA 1517/25-1/2.

this problem and, in the following, will derive a traversal and data management algorithm working on adaptive cartesian grids alike [7,12]. This algorithm then is parallelised using a domain decomposition approach based on [6]. Although the results are presented for a three–dimensional Poisson problem on an a priori refined grid only, we are able to solve any d–dimensional problem that can be discretised by a 3^d–point stencil. This is an important subtask of many more complex problems (the pressure Poisson part in the Navier–Stokes equations, e.g.) and starting point for the implementation of more difficult operators, such as the diffusion–convection operator or the diffusion operator for jumping material parameters.

The remainder is organised as follows: In Section 2, we introduce the adaptive cartesian grid our algorithm is based on. Section 3 is concerned with defining a traversal order (a linearisation) for the cells of this grid and exposing a vertex handling scheme, proving that two extra administrative bits per vertex are sufficient, both to store the complete grid structure including the geometry and to solve the equation system. Afterwards, in Section 4, we apply a hierarchical domain decomposition technique to end up with an algorithm whose communication data scales linearly with regard to the maximum number of vertices on the boundary of any partition. In Section 5, we present an upper bound for the corresponding constant, showing it is quasi–optimal. Finally, in Section 6, some numerical results for the Dirichlet Poisson problem are given, showing the efficiency with respect to both memory access and the parallelisation. Some final remarks in Section 7 conclude the discussion.

2 The Adaptive Grid

We create our grid using a hypercube $[0,1]^d$ and embed the computational domain into it. Then, the grid is refined in a recursive way, splitting up each cell into three parts along every coordinate axis. The depth of recursion and, hence, the resolution depends on both the boundary approximation and the numerical accuracy to be obtained. Following the notion of a spacetree (e.g. [1]) for a binary substructuring, we call these trees Peano spacetrees. A more formal definition as well as a reason for the division into three will be given later on.

On this grid, we use a nodal generating system [4] for the operator evaluation, that is a nodal basis on every grid level. Hereby the support of any shape function (hat), suitably scaled and dilated on a level k, shall be $[0, \frac{2}{3^k}]^d$. Consequently, a strictly element–wise assembly of the operators [1] is feasible, whereas within every geometric element only the element's vertices are needed. Since one degree of freedom is assigned to every vertex in this paper, the terms vertex and degree of freedom are used equivalently. Before implementing a solver on such a grid, one has to mention five important facts:

– If the values of an approximation are stored as hierarchical coefficients of the generating system $\hat{u}$ on the vertices, the inverse hierarchical transform (mapping from the hierarchical representation into a nodal basis representation

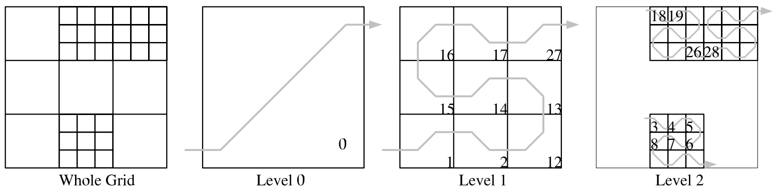


Fig. 1. An adaptive Peano grid and Peano spacetree of height three with corresponding cell order in two dimensions

of the finest level) $u = P\hat{u}$ can be done within one top–down traversal of the cell tree.

– If a value r is given on a vertex of the fine grid, the Galerkin hierarchical transform (mapping the other way round) $\hat{r} = P^T r$ of this value may happen during one bottom–up traversal.
– If a matrix–vector operation $Au = r$ with A generated by a 3^{d-1} stencil is given on any grid level, the result can be computed element–wise. Thus, all elements of this level have to be traversed once. Furthermore, the result value can be stored within the vertices directly, such that an explicit setup of matrix A is not needed at any time.
– Because of the last issue, both a residual computation and a Jacobi update step on any level can be done traversing all geometric elements of this level only once:

$$u^{(n+1)}_{level\ k} = u^{(n)}_{level\ k} + \omega\ diag^{-1}(A)\left(b - Au^{(n)}_{level\ k}\right). \tag{1}$$

– Combining equation (1) with the top–down–bottom–up arguments given above, one is able to implement an additive multigrid scheme with additive smoother [4], doing one depth–first sweep on the cell tree per iteration:

$$\begin{aligned}\hat{u}^{(n+1)} &= \hat{u}^{(n)} + \omega\ diag^{-1}(P^T AP)P^T\left(b - AP\hat{u}^{(n)}\right)\\ &=: \hat{u}^{(n)} + \omega\ diag^{-1}(P^T AP)\hat{r}^{(n)}.\end{aligned} \tag{2}$$

A detailed description of the actual realisation of such a solver can be found in [7,13]. In the following, we will focus on the development of a well–suited depth–first traversal of the grid, on the vertex management, and on the parallelisation. Thereby, regarding the operator evaluation, we focus on a strict element–wise evaluation scheme, where only the 2^d vertices of the current element have to be available at any time. As a result, every vertex is used 2^d times per iteration.

3 Grid Traversal Using a Peano Curve

Space–filling curves [15] are well known to simplify a lot of different tasks, due to their good locality properties ([3,5,6,7,8,10,12,13] e.g.). Their recursive, self–similar definition implies a depth–first traversal of the corresponding cell tree and, therefore, an enumeration of the cells of all levels. We are using the Peano curve as illustrated in Figure 1.

A Peano spacetree is a tree corresponding to a d–dimensional adaptive cartesian grid, where each node has either 0 or 3^d children. There is an order on the tree nodes (i.e. the geometric elements / grid cells) defined top–down by the Peano space–filling curve. Note that, if one inverts the Peano curve on the root level, the order on the child nodes on every level also is inverted. The resulting tree is again a Peano spacetree, which means this set of trees is closed under the invert–traverse operation.

Now, as a result of choosing this hierarchical grid and the Peano traverse, we have to provide a data structure such that the traversal algorithm is able to access the elements' vertices within every node for element–wise operator evaluation. This is not a new problem, e.g. [5] uses a hash function derived from the space–filling curve to access the vertices and shows some nice properties of such a scheme with respect to parallelisation and load balancing. We chose a different approach, exploiting the properties of the curve as well. Here, this idea is explained for the two–dimensional case. The recursive extension to arbitrary dimensionality is very technical, but is based on exactly the same ideas [7,9]. Where necessary, the basic construction ideas for $d > 2$ are presented:

First of all, one can observe that every continuous traverse splits up all vertices of the grid into left and right ones (in terms of their position with respect to the Peano traverse). This is formalised by a left–right classifier function

$$c_{LR_2} : vertices \mapsto \{L, R\} = \{0, 1\} \qquad \text{in } \mathbb{R}^2. \tag{3}$$

Given a $d \geq 2$ there are $d-1$ mappings of both the vertices and the space–filling curve onto the planes $(x_1, x_2), (x_1, x_3), \ldots$. For the Peano curve, the projection property holds [15], i.e. every projection onto the subplanes given before is a Peano curve again (see Figure 2). Thus, on each plane one can evaluate c_{LR_2} and combine the $d-1$ classifiers resulting in a $d-1$–dimensional left–right classifier function

$$c_{LR} : vertices \mapsto \{0, 1\}^{d-1}. \tag{4}$$

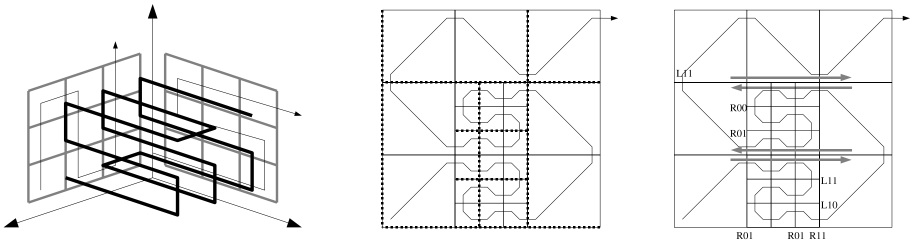


Fig. 2. On the left–hand side one can see the projection property, i.e. the projections of the Peano curve are again Peano curves. In the middle the alternating edge colouration is illustrated (even/odd indicated by dotted/solid), whereas on the right–hand side one can observe the palindrom / stack property (fat arrows). To some vertices their classifier value c is added.

The second idea is to have a look at the chronology a vertex is needed on a two–dimensional grid: If a "left" vertex is needed before another "left" vertex, the next time, the vertex is needed for operator evaluation, it is the other way round. Figure 2 shows this fact using grey arrows. Our idea is to use stacks, since they meet the resulting requirements: put a record (vertex) on the top of a stack after using it the first time, and pop a record from the stack when it is needed the second time. In addition to the left and right stack, one has to add a third idea, the stack colouring, within a hierarchical grid as pointed out first by [6], to avoid access conflicts due to the top–down bottom–up steps of the traverse in the generating system since there might be more than one degree of freedom per vertex.

So, the third idea is to colour the edges of a two–dimensional grid alternating along every axis in an even–odd manner. For example the left and the bottom edge of the root element are coloured. On any refinement level first of all the colours of the edges of the parent element are inherited, then the other edges are coloured again alternating (see Figure 2). Since every vertex is element of two edges, we get an additional qualifier

$$c_{col} : vertices \mapsto \{0,1\}^2 \qquad \text{in } \mathbb{R}^2 \tag{5}$$

defining the colour of a vertex. Combining (4) and (5), we end up with a classifier function

$$c : vertices \mapsto \{0,1\}^3 \qquad c = c_{LR} \circ c_{col} \qquad \text{in } \mathbb{R}^2 \tag{6}$$

for every vertex.

Lemma 1. *For $d = 2$ one is able to implement the whole vertex handling using $2^3 + 2$ stacks only.*

Assume there is a vertex stream, the vertices being ordered according to the very first vertex access. The first time a vertex v is required, it is read from the input stream. After the first usage, the vertex is stored on a stack $c(v)$. Next time it is needed for element–wise evaluation, it lays on top of stack $c(v)$. After the fourth usage, it is written to an output stream. As soon as one iteration is done, you can invert the Peano spacetree traversal order and switch input and output stream using them as stacks. For $d > 2$ this access scheme is extended in a recursive way regarding the axes, and the whole vertex handling can be done using $2^{2d-1} + 2$ stacks [9].

Implementing this algorithm, every vertex is augmented by two administrative bits: The first bit describes whether the vertex is inside or outside the domain. On the second bit we define an or–refinement semantic: An element is refined, if at least the refinement bit of one of the 2^d element's vertices is set. Thus, if n' is the number of vertices of the Peano spacetree, only $2n'$ bits are required for both the geometry and the grid description. During depth–first traversal, the whole traversal order can be reconstructed, evaluating the bits of the vertices of the current element and the current traversal state.

Using stacks is the first key for the high cache efficiency reported in [7,9], since for them the data access is highly local (no jumps within the memory), which is often named spatial locality. Using the Peano curve, the spatial locality [11] of the traverse results in very small temporary stacks and, therefore, good temporal locality of the stack access, which is the second key.

4 Domain Decomposition

Space–filling curves are well known within the parallel community for their good load–balancing and good spatial locality properties, i.e. ratio of surface divided by partition volume. For the Peano spacetree, we define a tree partition, based on an existing fine–grid domain decomposition, in a bottom-up manner.

We assume that we have a Peano domain decomposition of a fine grid into disjoint partitions. A Peano spacetree partition is a Peano spacetree minimal with respect to the number of tree nodes. Within this tree, all nodes of the given fine–grid partition as well as their fathers are contained and are called active. All vertices adjacent to the active elements are called active, too. Besides the active elements, the Peano spacetree partition contains all elements of the global tree, which are adjacent to the active vertices. The additional elements are called passive. As a result, every vertex a degree of freedom is assigned has again 2^d adjacent geometric elements within each spacetree partition it is contained.

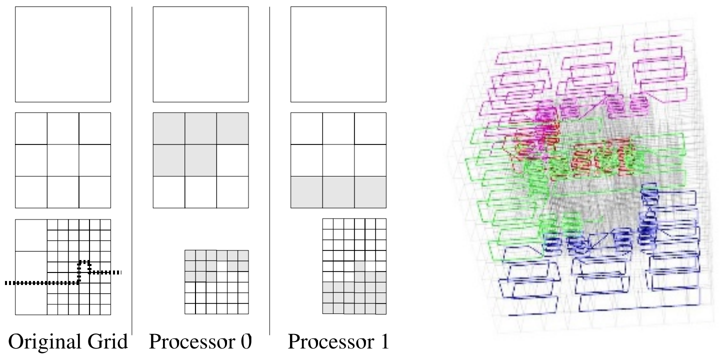


Fig. 3. The left–hand side shows a domain decomposition into two hierarchical partitions. The grey cells are held on a processor, but not evaluated since they do not belong to the processor (passive elements). The example on the right–hand side just gives an idea how a three–dimensional partitioning might look like.

Figure 3 shows two Peano spacetree partitions belonging together: Only the sets of active fine–grid elements are disjoint. For coarse grid elements this may not hold. Now, every processor has to traverse its Peano spacetree partition, whereas on the passive elements no calculation is done. In our additive multigrid algorithm, the restriction part of equation (2) is done on every processor autonomously without any master process. As we added the passive elements to the partition, the vertex management does not have to be modified and all the vertices, even those for which the process computed only a part of the residual, are transferred to the output stack:

$$\begin{aligned}\hat{r}^{(n)} &= P^T(b - AP\hat{u}^{(n)}) \\ &= \underbrace{P^T(b_{p0} - (AP\hat{u}^{(n)})_{p0})}_{\text{processor } 0} + \underbrace{P^T(b_{p1} - (AP\hat{u}^{(n)})_{p1})}_{\text{processor } 1} + \dots \end{aligned} \qquad (7)$$

When implementing the algorithm, we split up the output stream into two streams, one holding only vertices other processors are interested in. Either of them contains a subset of the global vertex stream that would correspond to a single processor run, and the global order of the vertices is preserved on all the output streams. Every vertex, with at least two processors interested in, has got a set of processors needing its residual contribution. This contribution might be sent to the other processors immediately, before the vertex is stored on the output stack, resulting in an asynchronous communication scheme. It is shown in [12] how to compute the set of interested computers on the fly. Furthermore, it is a good idea to buffer the elementary messages, depending on the hardware used.

After one iteration, all the residual contributions received and the own data, stored on one output stream, have to be merged. Since the order on the vertices is preserved, this can be done in $\mathcal{O}(s)$, where s is the number of vertices that had to be sent. Furthermore, this does not have to be done within a dedicated merge phase, but can be done during the next top–down traversal.

5 Efficiency of the Parallel Algorithm

Prior sections have shown how to implement an algorithm, linear in the number of unknowns n, in a (technically) efficient way on a parallel machine with p nodes without any major intrinsic serial part. According to [5], the performance of our algorithm, where the results have to be synchronised after every iteration, solely depends on the amount of data s'_k to be sent by a node k, such that the computational time per iteration is given by

$$t(n,p) = C_{solver}\frac{n}{p} + C_{startup} + C_{comm}\max_k\{s'_k\}, \qquad (8)$$

if one is not able to do the communication in an asynchronous way. The algorithm becomes quasi–optimal [5] for

$$\max_k\{s'_k\} \le C\left(\frac{n}{p}\right)^{1-1/d}, \qquad C \ge \sqrt[d]{\frac{2d^{d-1}\pi^{d/2}}{\Gamma(d/2)}}, \qquad (9)$$

reflecting the continuous Hölder continuity with parameter $\frac{1}{d}$ of a continuous space–filling curve [5,15].

In our case, the amount of data depends on the tree's height and the surface s of the fine–grid partition. There is a lot of published work on the interfaces of space–filling curves' partitions (e.g. [3,8]). Since most of this work deals with Hilbert and Lebesgue curves only, the proofs given there have to be transfered into the Peano curve case and have to be augmented by the tree issue.

In the following, we examine regular refined grids with $n \in (3^d)^{\mathbb{N}}$ geometric elements using p processors. The workload (number of geometric elements) is distributed equally among them.

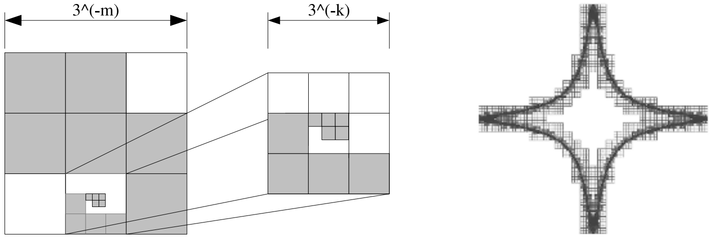


Fig. 4. Construction of a trivial upper bound of the surface of a partition induced by the Peano space–filling curve (grey), and the star shaped domain used in Section 6

Lemma 2. *The number of boundary vertices — vertices adjacent to passive geometric elements — on the fine grid of any partition is bounded by*

$$s' \leq \frac{4d}{1-3^{1-d}} 3^{d-1} \left(\frac{n}{p}\right)^{1-1/d} . \tag{10}$$

Proof. The proof follows the argumentation of [8]: Let M be the maximal tree depth, and m be the maximal tree level one would be able to embed the $\frac{n}{p}$ cells of the partition into one geometric element. On level m, the partition is contained in at most two elements, such that the bounding box of the two neighbouring geometric elements s_m is an upper bound for the continuous surface, if the domain was represented in the level's resolution (compare to Figure 4). On the finer levels $k > m$, there might be at most two appendices (cells containing not only active subcells), since the space–filling curve used is compact and continuous (therefore, all the children of a node are visited, before the next node is processed). Their boundary box surface is already considered, but the possibly resulting concave surface parts s_k have to be added to the result. This surface is bounded by the bounding box of an element of level k. Finally, the continuous surface $s(n,p)$ is divided by the fine grid element face size, which is $3^{-M(d-1)}$, giving the number of fine–grid vertices up to a small constant:

$$\left(3^d\right)^M = n \qquad \left(3^d\right)^{M-m-1} \leq \frac{n}{p} \leq \left(3^d\right)^{M-m} \tag{11}$$

$$s(n,p) \leq 2 \sum_{k=m}^{M-1} s_k \leq 2 \sum_{k=m}^{M-1} 2d \left(\frac{1}{3}\right)^{(d-1)k} \leq \frac{4d}{1-3^{1-d}} 3^{(1-d)m}$$

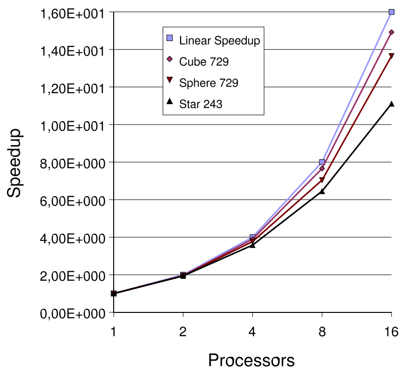
$$s' \leq \frac{4d}{1-3^{1-d}} 3^{(1-d)m+M(d-1)} \leq \frac{4d}{1-3^{1-d}} 3^{d-1} \left(\frac{n}{p}\right)^{1-1/d} . \quad (12)$$

The amount of data sent by one processor is bounded by a geometric series with argument $\left(\frac{1}{3}\right)^{d-1}$ for the grid levels m to M scaled by s', as the number of vertices decreases with this factor for each coarsening step. For the levels $0 \ldots -1$ the number of active cells enclosing the partition is bounded by two. Therefore, the number of boundary vertices is bounded by $3 \cdot 2^{d-1}$ for each level.

$$s = s' \sum_{k=m}^{M} \left(\frac{1}{3}\right)^{(d-1)k} + m\,\frac{3}{2} 2^d \leq \frac{3^{d-1}}{1-3^{1-d}} \underbrace{p^{1/d-1}}_{\leq 1} s' + 3 \cdot 2^{d-1} \log_3 \sqrt[d]{p}. \quad (13)$$

6 Results

Figure 5 gives the parallel behaviour of the code presented in this paper for three dimensions. This code is not optimised yet, but already shows all the properties stated in this paper for a Dirichlet–Poisson problem on the cube, a sphere, or a star domain (see Figure 4), as well as the excellent cache behaviour (see [6,7,9,10,12,13], e.g.). The star domain experiment suffers from the lack of dynamic load balancing not implemented yet: Since the ratio of inner cells to cells outside the domain, where no operator evaluation is necessary, is unfavourable, a simple equidistant curve partitioning fails. The same reasoning holds for the sphere.



	cube 729^3	sphere 729^3	star 243^3
real dofs	$\approx 4.0 \cdot 10^8$	$\approx 2.4 \cdot 10^7$	$\approx 4.3 \cdot 10^5$
S(2)	1.95	1.95	1.94
S(4)	3.9	3.77	3.58
S(8)	7.66	7.04	6.45
S(16)	14.92	13.65	11.11
L2 CHR	99.96%	99.94%	99.95%

Fig. 5. Some parallel performance results for $d = 3$ on a Myrinet cluster of Dual Pentium III 800 MHz with 2GByte RAM per node [12]. $S(p)$ denotes the speedup on p processors, L2 CHR abbreviates level 2 cache–hit rate.

7 Concluding Remarks

In this paper, we have presented a parallel multigrid PDE solver based on the Peano spacetree, handling all the vertices solely using stacks. Since this approach has proven to be of value with respect to memory requirements, parallelisation, and cache efficiency, it is our strategy to use this algorithm within a more complex environment. In fact, we have already used exactly the same approach to prototype a Navier–Stokes solver [14]. Furthermore, it has been shown that our algorithmic approach is well–suited for a posteriori refinement [13]. Since we are working on trees, dynamic load balancing can be implemented in a very natural way by forking trees [10]. Right now we are integrating all these aspects into one d–dimensional PDE solver, embedded into a fluid–structure interaction application framework [2].

It is work in progress, how to extend the scheme to higher order stencils and to provide better estimates on the amount of data to be communicated. Furthermore, the behaviour on a massively parallel cluster and different load balancing strategies have to be evaluated.

Special thanks to Markus Pögl and Markus Langlotz, for doing a first implementation of the algorithm presented and solving many implementation issues.

References

1. Bader M., Frank A.C., Zenger Ch.: An Octree-Based Approach for Fast Elliptic Solvers. High Performance Scientific and Engineering Computing. Springer-Verlag, Berlin Heidelberg (2001)
2. Brenk M., Bungartz H.J., Mehl M., Neckel T.: Fluid–Structure Interaction on Cartesian Grids: Flow Simulation and Coupling Environment. Fluid-Structure Interaction, LNCS (to appear)
3. Gotsman C., Lindenbaum M.: On the Metric Properties of Discrete Space–Filling Curves. IEEE Transactions on Image Processing vol. 5 (**5**) (1996) 794-797
4. Griebel M.: Multilevel algorithms considered as iterative methods on indefinite systems. SFB-Bericht **342/29/91** (1991)
5. Griebel M., Zumbusch G.: Hash–Storage Techniques for Adaptive Multilevel Solvers and Their Domain Decomposition Parallelization. Proceedings of Domain Decomposition Methods 10, DD10 **218** (1998) 279–286
6. Günther F., Krahnke A., Langlotz M., Mehl M., Pögl M., Zenger Ch.: On the Parallelization of a Cache-Optimal Iterative Solver for PDEs Based on Hierarchical Data Structures and Space-Filling Curves. Recent Advances in Parallel Virtual Machine and Message Passing Interface, LNCS **3241** (2004) 425-429
7. Günther F., Mehl M., Pögl M., Zenger Ch.: A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. SIAM Journal on Scientific Computing (to appear)
8. Hungershöfer J., Wierum J.M.: On the Quality of Partitions based on Space–Filling Curves. International Conference on Computational Science 2002, LNCS **2331** (2002): 31-45
9. Hartmann J.: Entwicklung eines cache–optimalen Finite–Element–Verfahrens zur Lösung d-dimensionaler Probleme. Diploma thesis, Technical University Munich (2004)
10. Herder W.: Entwicklung eines cache–optimalen Finite–Element–Verfahrens zur Lösung d-dimensionaler Probleme. Diploma thesis, Technical University Munich (2005)

11. Kowarschik M., Weiß C.: An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. Proceedings of the GI-Dagstuhl Forschungseminar: Algorithms for Memory Hierarchies, LNCS **2625** (2003) 213-232
12. Langlotz M.: Parallelisierung eines Cache–optimalen 3D Finite–Element–Verfahrens. Diploma thesis, Technical University Munich (2005)
13. Mehl M., Weinzierl T., Zenger C.: A cache–oblivious self–adaptive full multigrid method. Numerical Linear Algebra With Applications (to appear)
14. Neckel T.: Einfache 2D–Fluid–Struktur–Wechselwirkungen mit einer cache–optimalen Finite–Element–Methode. Diploma thesis, Technical University Munich (2005)
15. Sagan H.: Space-Filling Curves, Springer-Verlag, Berlin Heidelberg (1994)