A Particle-Partition of Unity Method–Part IV: Parallelization

Michael Griebel and Marc Alexander Schweitzer

Institut für Angewandte Mathematik, Universität Bonn, Bonn, Germany.

Abstract In this sequel to [7, 8, 9] we focus on the parallelization of our multilevel partition of unity method for distributed memory computers. The presented parallelization is based on a data decomposition approach which utilizes a key-based tree implementation and a weighted space filling curve ordering scheme for the load balancing problem. We present numerical results with up to 128 processors. These results show the optimal scaling behavior of our algorithm.

1 Introduction

Numerical simulations with millions of degrees of freedom require the use of optimal order algorithms. Here, the number of operations necessary to obtain an approximate solution within a given accuracy should be of the order $O(\mathsf{dof})$. But still the overall computing time can be unacceptably large. Another problem with large simulations is their storage demand. These issues make the use of parallel computers with distributed memory a must for large numerical simulations. Here, the data have to be partitioned into almost equally sized parts and assigned to different segments of the distributed memory to exploit the available memory. This partitioning, however, should allow for a single processor to complete most operations on its assigned part of the data independently of the other processors. Furthermore, the number of operations per processor should be almost identical to utilize all available processors for the larger part of the computation.

In this paper we present the parallelization of our multilevel particle-partition of unity method for the approximate solution of an elliptic partial differential equation. The main ingredients are a key-based tree implementation and a space filling curve load balancing scheme. The overall method can be split into three major steps: The initial tree construction and load balancing step, the assembly step where we set up the stiffness matrices A_k and interlevel transfers on all levels $k = 0, \ldots, J$, and finally the solution step where we use a multiplicative multilevel iteration to solve the linear (block-)system $A_J \tilde{u}_J = \hat{f}_J$. The complexities of the tree construction and load balancing step is given by $O(\frac{N}{\wp} J + (\frac{N}{\wp})^{\frac{d-1}{d}} + J(\log \wp)^2 + \wp \log \wp)$ where N denotes the number of leaves of the tree, J denotes the number of levels of the tree ($J \simeq \log N$ for a balanced tree) and $\wp$ is the number of processors. The assembly of the stiffness matrices is trivially parallel with a complexity of $O(\frac{N}{\wp})$, and the complexity of the solution step is the well-known complexity $O(\frac{N}{\wp} + (\frac{N}{\wp})^{\frac{d-1}{d}} + J + \log \wp)$ of a multiplicative multilevel iteration [26]. The results of our numerical experiments with up to 128 processors and 42 million degrees of freedom clearly show the scalability of our method.

The remainder of the paper is organized as follows. In §2 we shortly review the Galerkin discretization and multilevel solution of an elliptic partial differential equation with the particle-partition of unity method. The parallelization of our method is presented in §3. Here, a fundamental ingredient is a key-based tree implementation which we introduce in §3.2 and §3.3. We focus on the load balancing problem in §3.4. There, we present a cheap numerical scheme based on space filling curves to (re-)partition data in parallel. The algorithmic changes we have to apply to our particle-partition of unity for parallel computations are presented in §3.5, §3.6 and §3.7. The results of our numerical experiments are presented in §4. Finally, we conclude with some remarks in §5.

2 Particle-Partition of Unity Method

In the following, we give a short recap of our particle-partition of unity method for the Galerkin discretization of an elliptic partial differential equation of second order and its multilevel solution, see [7, 8, 9, 19] for details.

2.1 Construction of Partition of Unity Spaces

In a partition of unity method (PUM), we define a global approximation u^{PU} simply as a weighted sum of local approximations u_i,

$$u^{\mathrm{PU}}(x) := \sum_{i=1}^{N} \varphi_i(x) u_i(x). \tag{2.1}$$

These local approximations u_i are completely independent of each other. The local supports $\omega_i := \operatorname{supp}(u_i)$, the local basis $\{\psi_i^n\}$ and the order of approximation p_i for every single $u_i := \sum_n u_i^n \psi_i^n$ can be chosen independently of all other u_j. Here, the functions φ_i form a partition of unity (PU), i.e. $\sum_i \varphi_i \equiv 1$. They are used to splice the local approximations u_i together in such a way that the global approximation u^{PU} benefits from the local approximation orders p_i, yet it still fulfills global regularity conditions. For further details see [7, 19].

The starting point for any meshfree method is a collection of N independent points $P := \{x_i \in \mathbb{R}^d \,|\, x_i \in \overline{\Omega}, i = 1, \dots, N\}$. In the partition of unity approach we first have to construct a PU $\{\varphi_i\}$ on the domain of interest Ω. Then we can choose local approximation spaces $V_i^{p_i} = \operatorname{span}\langle \psi_i^n \rangle$ on the patches ω_i to define the PUM space

$$V^{\mathrm{PU}} := \sum_i \varphi_i V_i^{p_i} = \sum_i \varphi_i \operatorname{span}\langle \{\psi_i^n\} \rangle = \operatorname{span}\langle \{\varphi_i \psi_i^n\} \rangle$$

and an approximate solution (2.1). Here, the union of the supports $\operatorname{supp}(\varphi_i) = \overline{\omega_i}$ have to cover the domain, i.e. $\overline{\Omega} \subset \bigcup_{i=1}^N \omega_i$. Given a cover $C_\Omega := \{\omega_i \,|\, i = 1, \ldots, N\}$ we can define a PU by using *Shepard functions* as φ_i.

The efficient construction of an appropriate cover C_Ω for general point sets P is not an easy task [7, 8, 19] and it is the most crucial step in a PUM. The cover has a significant impact on the computational costs associated with the assembly of the stiffness matrix A, since the cover already defines the (block-)sparsity pattern of the stiffness matrix, i.e. the number of integrals to be evaluated. Furthermore, the cover influences the algebraic structure of the PU functions φ_i, which has to be resolved for the proper integration of a stiffness matrix entry [7, 8, 19]. Throughout this paper we use a parallel version (see §3.5) of a tree-based algorithm for the construction of a sequence of rectangular covers [8, 9].

With the help of weight functions W_k defined on the patches ω_k of the cover C_Ω we can easily generate a PU by Shepard's method, i.e. we define

$$\varphi_i(x) = \frac{W_i(x)}{\sum_{\omega_k \in C_i} W_k(x)}, \tag{2.2}$$

where $C_i := \{\omega_j \in C_\Omega \,|\, \omega_i \cap \omega_j \neq \emptyset\}$ is the set of all geometric neighbors of a cover patch ω_i. We restrict ourselves to the use of cover patches ω_i which are d-rectangular, i.e. the patches ω_i are products of intervals $[x_i^l - h_i^l, x_i^l + h_i^l]$. Therefore, the most natural choice for a weight function W_i is a product of one-dimensional functions, i.e. $W_i\,(x) = \prod_{l=1}^d W_i^l\,(x^l) = \prod_{l=1}^d \mathcal{W}\,(\frac{x^l - x_i^l + h_i^l}{2h_i^l})$ with $\operatorname{supp}(\mathcal{W}) = [0,1]$ such that $\operatorname{supp}(W_i) = \overline{\omega_i}$. It is sufficient for this construction to choose a one-dimensional weight function $\mathcal{W}$ which is non-negative. The PU functions φ_i inherit the regularity of the generating weight function $\mathcal{W}$. We always use a normed B-spline [19] as the generating weight function $\mathcal{W}$, throughout this paper we use a linear B-spline.

In general, a PU $\{\varphi_i\}$ can only recover the constant function on the domain Ω. For the discretization of a partial differential equation (PDE) this approximation quality is not sufficient. Therefore, we multiply the PU functions φ_i locally with polynomials ψ_i^n. Since our cover patches ω_i are d-rectangular, a local tensor product space is the most natural choice. Throughout this paper, we use products of univariate Legendre polynomials as local approximation spaces $V_i^{p_i}$, i.e. we choose

$$V_i^{p_i} = \operatorname{span}\langle\{\psi_i^n \,|\, \psi_i^n = \prod_{l=1}^d \mathcal{L}_i^{\hat{n}_l}, \|\hat{n}\|_1 = \sum_{l=1}^d \hat{n}_l \leq p_i\}\rangle,$$

where $\hat{n} = (\hat{n}_l)_{l=1}^d$ is the multi-index of the polynomial degrees $\hat{n}_l$ of the univariate Legendre polynomials $\mathcal{L}_i^{\hat{n}_l} : [x_i^l - h_i^l, x_i^l + h_i^l] \to \mathbb{R}$, and n is the index associated with the product function $\psi_i^n = \prod_{l=1}^d \mathcal{L}_i^{\hat{n}_l}$.

2.2 Galerkin Discretization

We want to solve elliptic boundary value problems of the type

$$\begin{aligned} Lu &= f \text{ in } \Omega \subset \mathbb{R}^d \,, \\ Bu &= g \text{ on } \partial\Omega \,, \end{aligned}$$

where L is a symmetric partial differential operator of second order and B expresses suitable boundary conditions. For reasons of simplicity we consider in the following the model problem

$$\begin{aligned} -\Delta u + u &= f \text{ in } \Omega \subset \mathbb{R}^d \,, \\ \nabla u \cdot n_\Omega &= g \text{ on } \partial\Omega \,, \end{aligned} \tag{2.3}$$

of Helmholtz type with natural boundary conditions. The Galerkin discretization of (2.3) leads to a definite linear system.[1] In the following let $a(\cdot,\cdot)$ be the continuous and elliptic bilinear form induced by L on $H^1(\Omega)$. We discretize the PDE using Galerkin's method. Then, we have to compute the stiffness matrix

$$A = \left(A_{(i,n),(j,m)}\right), \text{ with } A_{(i,n),(j,m)} = a\left(\varphi_j \psi_j^m, \varphi_i \psi_i^n\right) \in \mathbb{R},$$

and the right hand side vector

$$\hat{f} = \left(f_{(i,n)}\right), \text{ with } f_{(i,n)} = \langle f, \varphi_i \psi_i^n \rangle_{L^2} = \int_\Omega f \varphi_i \psi_i^n \in \mathbb{R}.$$

Throughout this paper we assume that the stiffness matrix is arranged in polynomial blocks $A_{i,j} = (A_{(i,n),(j,m)})$ [9].

The integrands of the weak form of (2.3) may have quite a number of jumps of significant size since we use piecewise polynomial weights W_i whose supports ω_i overlap in the Shepard construction (2.2). Therefore, the integrals of the weak form have to be computed carefully using an appropriate numerical quadrature scheme, see [7, 8].

2.3 Multilevel Solution

We use a multilevel solver developed in [9] for the fast and efficient solution of the resulting large sparse linear (block-)system $A\tilde{u} = \hat{f}$, where $\tilde{u}$ denotes a coefficient (block-)vector and $\hat{f}$ a moment (block-)vector.

[1] The implementation of Neumann boundary conditions with our PUM is straightforward and similar to their treatment within the finite element method (FEM). The realization of essential boundary conditions with meshfree methods is more involved than with a FEM due to the non-interpolatory character of the meshfree shape functions. There are several different approaches to the implementation of essential boundary conditions with meshfree approximations, see [1, 7, 12, 19]. Note that the resulting linear system may be indefinite, e.g. when we use Lagrangian multipliers to enforce the essential boundary conditions. A more natural approach toward the treatment of Dirichlet boundary conditions due to Nitsche [16] leads to a definite linear system.

In a multilevel method we need a sequence of discretization spaces V_k with $k = 0, \dots, J$ where J denotes the finest level. To this end we construct a sequence of PUM spaces V_k^{PU} as follows. We use a tree-based algorithm developed in [8, 9] to generate a sequence of point sets P_k and covers C_Ω^k from a given initial point set $\tilde{P}$. Following the construction given in §2.1 we can then define an associated sequence of PUM spaces V_k^{PU}. Note that these spaces are nonnested, i.e. $V_{k-1}^{\mathrm{PU}} \not\subset V_k^{\mathrm{PU}}$, and that the shape functions $\varphi_{i,k}\psi_{i,k}^n$ are non-interpolatory. Thus, we need to construct appropriate transfer operators $I_{k-1}^k : V_{k-1}^{\mathrm{PU}} \to V_k^{\mathrm{PU}}$ and $I_k^{k-1} : V_k^{\mathrm{PU}} \to V_{k-1}^{\mathrm{PU}}$. With such transfer operators I_{k-1}^k, I_k^{k-1} and the stiffness matrices A_k coming from the Galerkin discretization on each level k we can then set up a standard multiplicative multilevel iteration to solve the linear system $A_J \tilde{u}_J = \hat{f}_J$.

Our multilevel solver utilizes special localized L^2-projections for the interlevel transfers and a block-smoother to treat all local degrees of freedom ψ_i^n within a patch ω_i simultaneously. For further details see [9].

3 Parallel Particle-Partition of Unity Method

In this section we present the parallelization of our PUM. Here, we use a data decomposition approach to split up the data among the participating processors and their respective local memory.

Our cover construction algorithm [8] is essentially a simple tree algorithm. Hence, we need to be concerned with a parallel tree implementation (§3.2 and §3.3). Another cause of concern in parallel computations is the load balancing issue which we discuss in §3.4. We then focus on the parallel cover construction in §3.5 where we construct a sequence of d-rectangular covers C_Ω^k. The assembly of the stiffness matrices A_k on all levels k in parallel is presented in §3.6. Finally, we discuss the multilevel solution of $A_J \tilde{u}_J = \hat{f}_J$ in parallel in §3.7.

Note that neither the assembly phase nor the solution phase make explicit use of the tree data structure. Here, we employ a parallel sparse matrix data structure to store each of the sparse (block-)matrices A_k, I_{k-1}^k and I_k^{k-1} on all levels k. The neighborhoods $C_{i,k} := \{\omega_{j,k} \in C_\Omega^k \,|\, \omega_{j,k} \cap \omega_{i,k} \neq \emptyset\}$ determine the sparsity pattern of the stiffness matrices A_k, i.e. the nonzero (block-)entries of the ith (block-)row. Furthermore, they are needed for the evaluation of (2.2). Once the neighborhoods are known the evaluation of a PU function (2.2) and the matrix assembly are independent of the tree construction. The tree data structure is used only for the multilevel cover construction and for the efficient computation of the neighborhoods $C_{i,k}$.

3.1 Data Decomposition

There are two main tasks associated with the efficient parallelization of any numerical computation on distributed memory computers. The first is to evenly split up the data among the participating processors, i.e. the associated computational work should be well-balanced. The second is to allow for an efficient access to data stored by another processor; i.e. on distributed memory parallel computers also the amount of remote data needed by a processor should be small.

In a data decomposition approach we partition the data, e.g. the computational domain or mesh, among the participating processors [17]. Then, we simply restrict the operations of the global numerical method to the assigned part of the data/domain. A processor has read and write access to its local data but only read access to remote data it may need to complete its local computation. On distributed memory machines these required data have be to exchanged explicitly in distinct communication steps.

The quality of the partition of the domain/data essentially determines the efficiency of the resulting parallel computation. The local parts of the data assigned to each processor should induce a similar amount of computational work so that each processor needs roughly the same time to complete its local computation. Here, a processor may need to access the data of the neighboring sub-domains to solve its local problem. Hence, the geometry of the sub-domains should be simple to limit the number of communication steps and the communication volume. The number of neighboring processors (which determines the number of communication steps) should be small and the geometry of the local boundary (which strongly influences the communication volume) should be simple, i.e. its size should be small.

The data structure which describes the computational domain in our PUM is a d-binary tree (quadtree, octree) used for the cover construction [8] and the fast neighbor search for the evaluation of the Shepard PU functions (2.2). In a conventional implementation of a d-binary tree the topology is represented by storing links to the successor cells in the tree cells. Note that this data structure does not allow for random access to a particular cell of the tree and special care has to be taken on distributed memory machines if a successor cell is assigned to another processor. These issues make the use of a conventional tree implementation rather cumbersome on a distributed memory parallel computer.

3.2 Key Based Tree Implementation

A different implementation of a d-binary tree which is more appropriate for distributed memory machines was developed in [23, 24]. Here, the tree is realized with the help of a hashed associative container. A unique label is assigned to each possible tree cell and instead of linking a cell directly to its successor cells the labeling scheme implicitly defines the topology of the tree and allows for the easy access to successors and ancestors of a particular tree cell. Furthermore, we can randomly access any cell of the tree via its unique label. This allows us to catch accesses to non-local data and we can easily compute the communication pattern and send and receive all necessary data to complete the local computation.

The labeling scheme maps tree cells $\mathcal{C}_L = \bigotimes_{i=1}^d [c_L^i, c_L^i + h_L^i] \subset \mathbb{R}^d$ to a single integer value $\mathsf{k}_L \in \mathbb{N}_0$, the *key*. For instance, we can use the *d-binary path* as the key value k_L associated with a tree cell $\mathcal{C}_L$. The d-binary path k_L is defined by the search path that has to be completed to find the respective cell in the tree. Starting at the root of the tree we set $\mathsf{k}_L = 1$ and descend the tree in the direction of the cell $\mathcal{C}_L$. Here we concatenate the current key value (in binary representation) and the d Boolean values 0 and 1 associated with the decisions to which successor cell the descent continues to reach the respective tree cell $\mathcal{C}_L$. In Table 1 we give the resulting path key values k_L for a two dimensional example. Note that the key value $\mathsf{k}_L = 1$ for the root cell is essentially a stop bit which is necessary to insure the uniqueness of the key values.

Table1. Path key values for the successor cells of a tree cell $\mathcal{C}_L = \bigotimes_{i=1}^d [c_L^i, c_L^i + h_L^i]$ with associated key k_L in two dimensions.

successor cell	binary key value	integer key value
$[c_L^1, c_L^1 + \frac{1}{2}h_L^1] \times [c_L^2, c_L^2 + \frac{1}{2}h_L^2]$	$\mathsf{k}_L 00$	$4\mathsf{k}_L$
$[c_L^1, c_L^1 + \frac{1}{2}h_L^1] \times [c_L^2 + \frac{1}{2}h_L^2, c_L^2 + h_L^2]$	$\mathsf{k}_L 01$	$4\mathsf{k}_L + 1$
$[c_L^1 + \frac{1}{2}h_L^1, c_L^1 + h_L^1] \times [c_L^2, c_L^2 + \frac{1}{2}h_L^2]$	$\mathsf{k}_L 10$	$4\mathsf{k}_L + 2$
$[c_L^1 + \frac{1}{2}h_L^1, c_L^1 + h_L^1] \times [c_L^2 + \frac{1}{2}h_L^2, c_L^2 + h_L^2]$	$\mathsf{k}_L 11$	$4\mathsf{k}_L + 3$

3.3 Parallel Key Based Tree Implementation

The use of a global unique integer key for each cell of the tree allows for a simple description of a partitioning of the computational domain. The set of all possible[2] keys $\{0, 1, \dots, \mathsf{k}_{\max}\}$ is simply split into $\wp$ subsets which are then assigned to the $\wp$ processors. We subdivide the range of keys into $\wp$ intervals

$$0 = r_0 \leq r_1 \leq \dots \leq r_\wp = \mathsf{k}_{\max}$$

and assign the interval $[r_q, r_{q+1})$ to the qth processor, i.e. the set of tree cells assigned to the qth processor is $\{\mathcal{C}_L \,|\, \mathsf{k}_L \in [r_q, r_{q+1})\}$. With this very simple decomposition each processor can identify which processor stores a particular tree cell $\mathcal{C}_L$. A processor only has to compute the key value k_L for the tree cell $\mathcal{C}_L$ and the respective interval $[r_q, r_{q+1})$ with $\mathsf{k}_L \in [r_q, r_{q+1})$ to determine the processor q which stores this tree cell $\mathcal{C}_L$. The question now arises if such a partition of the domain with the path keys k_L is a reasonable

[2] The maximal key value $\mathsf{k}_{\max}$ is a constant depending on the architecture of the parallel computer.

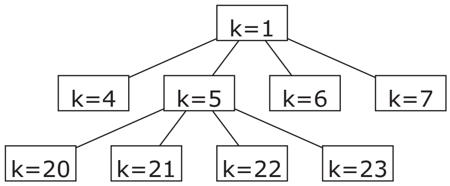

Figure1. The tree is ordered horizontally by the path key values k.

choice? Obviously the partitioning of the tree should be done in such a fashion that complete sub-trees are assigned to a processor to allow for efficient tree traversals. But the path key labeling scheme given above orders the tree cells rather horizontally (see Figure 1) instead of vertically. Therefore, we need to transform the path keys k_L to so-called domain keys k_L^D.

A simple transformation which leads to a vertical ordering of the tree cells is the following: First, we remove the leading bit (the initial root key value) from the key's binary representation. Then we shift the remaining bits all the way to the left so that the leading bit of the path information is now stored in the most significant bit.[3] Assume that the key values are stored as an 8 bit integer and that we are in two dimensions. Then this simple transformation of a path key value $\mathsf{k}_L = 18$ to a respective domain key value $\mathsf{k}_L^D = 32$ is given by

$$\mathsf{k}_L = 0001\underbrace{0010}_{\text{path}} \mapsto \underbrace{0010}_{\text{path}}0000 = \mathsf{k}_L^D. \tag{3.1}$$

With these domain keys k_L^D the tree is now ordered vertically and we can assign complete sub-trees to a processor using the simple interval domain description $[r_q, r_{q+1})$. But the transformed keys are no longer unique and cannot be used as the key value for the associative container to store the tree itself. Obviously, a successor cell $\mathcal{C}_S$ of a tree cell $\mathcal{C}_L$ can be assigned the same domain key as the tree cell, i.e. $\mathsf{k}_S^D = \mathsf{k}_L^D$. Hence, we use the unique path keys k_L for the container and the associated domain keys k_L^D for the domain description, i.e. for the associated interval boundaries $[r_q, r_{q+1})$.

Note that the description of the data partition via the intervals $[r_q, r_{q+1})$ defines a minimal refinement stage of the tree which has to be present on all processors to insure the consistency of the tree. In the following we refer to this top part of the tree as the *common global tree*. The leaves $\mathcal{C}_L$ of the common global tree are characterized by the fact that they are the coarsest tree cells for which all possible successor cells are stored on the same processor, see Figure 2. The domain key values k_S^D of all possible successor cells $\mathcal{C}_S$ lie

[3] This transformation needs $O(1)$ operations if we assume that the current refinement level of the tree is known, otherwise it is of the order $O(J)$, where J denotes the number of levels of the tree.

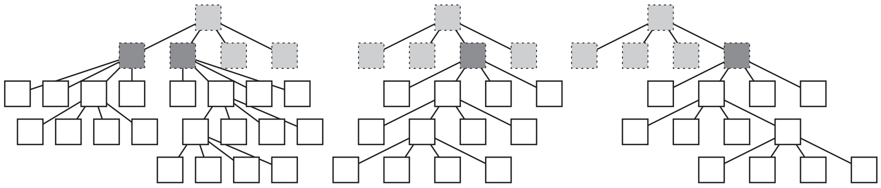

Figure2. Common global tree (dashed, gray shaded) for a partition onto 3 processors. Local sub-tree roots (dark gray shaded) and the local sub-tree cells (white) for the first (left), second (center) and third processor (right).

in the same interval $[r_q, r_{q+1})$ as the domain key k_L^D. We therefore refer to the leaves of the common global tree as *local sub-tree roots*.

3.4 Load Balancing with Space Filling Curves

The order of the tree cells induced by the domain keys k_L^D given above is often referred to as bit-interleaving, the Morton-order or the Z-order (N-order). The curve induced by mapping the domain keys to the associated cell centers corresponds to the Lebesgue curve (Figure 3 (upper left)) which is a space filling curve [18]. There are many space filling curves with different properties which might be more suitable for our needs; e.g. the sub-domains generated by the Lebesgue curve may be not connected [27] even for a d-rectangle, see Figure 3 (upper right). This increases the size of the local boundary and thereby the communication volume and possibly the number of communication steps.

The properties of space filling curves with respect to partitioning data for parallel computations have been studied in [27, 28]. Here, it turns out that the Hilbert curve (Figure 3 (lower left)) is more suitable for partitioning irregular data than the Lebesgue curve. It provides a better data locality, e.g. the constructed sub-domains for a d-rectangle are connected (Figure 3 (lower right)) and the size of the local boundaries is of optimal order. Hence, we use the Hilbert curve instead of the Lebesgue curve to order the tree in our implementation, i.e. we use a different transformation than (3.1) to map the path keys k_L to domain keys k_L^D. This transformation of the path key values to Hilbert curve keys is more involved than the transformation (3.1) to Lebesgue curve keys, but it can also be realized with fast bit manipulations.[4]

[4] In general the transformation of a given key k_L to its associated Hilbert domain key k_L^D needs $O(J)$ operations, even if the current tree level J is known. But since we are interested in the domain keys k_L^D keys for all cells (or at least for all leaves) of the tree we can merge the transformation with the tree traversal which reduces the complexity of the transformation of a single key to $O(1)$.

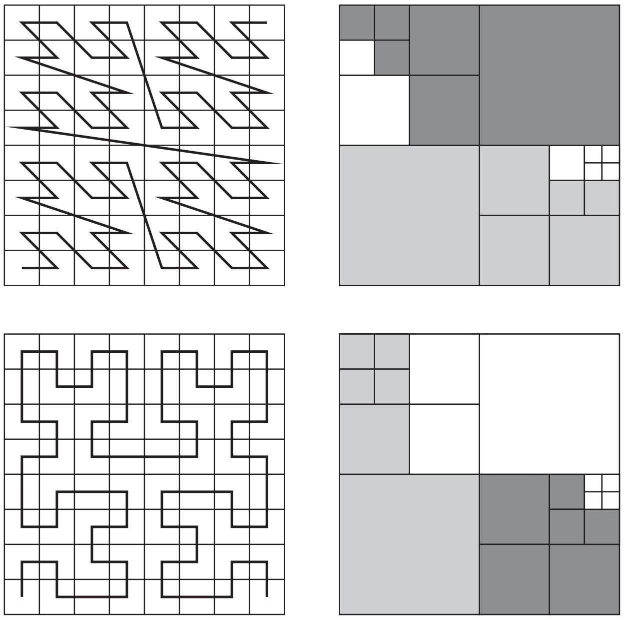

Figure3. The Lebesgue curve (upper left) and the constructed sub-domains (upper right) for a partition onto three processors. The sub-domains are not connected since the curve does not have the locality property. The Hilbert curve (lower left) and the constructed sub-domains (lower right) for a partition onto three processors. The sub-domains are connected due to the locality property of the curve.

The use of the Hilbert curve was also suggested by Warren and Salmon in [23, 25]. In [4, 27] the parallel performance of tree-based algorithms on Hilbert curve induced partitions was studied.

By changing the interval boundaries $\{r_q \,|\, q = 0, \ldots, \wp\}$ we can balance the load among the processors. To this end we assign estimated work loads w_L as weights to the leaves $\mathcal{C}_L$ of the tree. Then we compute the current load estimate $w^{\hat{q}} = \sum w_L$ on every processor $\hat{q}$ and gather all remote load estimates w^q with $q \neq \hat{q}$. Then, the global load estimate $w = \sum_{q=0}^{\wp-1} w^q$, and the balanced load distribution $w_b^q = \frac{qw}{\wp}$ are computed. In the next step every processor iterates over its current set of leaves $\mathcal{C}_L$ of the tree in ascending order of the domain keys k_L^D and sets new (intermediate) interval boundaries $\{\tilde{r}_q \,|\, q = 0, \ldots, \wp\}$ accordingly. Finally, a reduction operation over all sets $\{\tilde{r}_q \,|\, q = 0, \ldots, \wp\}$ gives the new interval boundaries $\{r_q \,|\, q = 0, \ldots, \wp\}$ which balance the estimated load w. Note that this load balancing scheme itself is completed in parallel.

Algorithm 1 (Load Balancing).

1. For all local leaves $\mathcal{C}_L$ of the tree: Assign estimated work load w_L.
2. Compute local estimate $w^{\hat{q}} = \sum_L w_L$.
3. Gather remote estimates w^q with $q = 0, \dots, \wp - 1$ and $q \neq \hat{q}$.
4. Compute global load estimate $w = \sum_{q=0}^{\wp-1} w^q$.
5. Set local estimate $w_g^{\hat{q}} = \sum_{q=0}^{q<\hat{q}} w^q$.
6. Set balanced load distribution $w_b^q = \frac{qw}{\wp}$ for $q = 0, \dots, \wp$.
7. For all local leaves $\mathcal{C}_L$ (in ascending order of domain keys k_L^D): Set local intermediate interval boundary $\tilde{r}_{\tilde{q}} = k_L^D$ where $\tilde{q} \in \{0, \dots, \wp\}$ is the smallest integer with $w_g^{\hat{q}} \leq w_b^{\tilde{q}}$ and update estimate $w_g^{\hat{q}} = w_g^{\hat{q}} + w_L$.
8. Set interval boundaries $r_q = \max_q \tilde{r}_q$ by reducing the intermediate boundaries $\tilde{r}_q$ over all processors, force $r_0 = 0$ and $r_\wp = \mathsf{k}_{\max}$.

The complexity of this load balancing scheme is given by $O(\frac{\mathrm{card}(P_J)}{\wp} + \wp \log \wp)$, where P_J denotes the generating point set for our PUM space V_J^{PU} on the finest level J, i.e. $\mathrm{card}(P_J)$ corresponds to the number of leaves of the tree.[5] We use the number of neighboring patches $\mathrm{card}(C_{L,J})$ on the finest level J as the load work estimate w_L. By this choice we balance the number of (block-)integrals on the finest level among the processors. Under the assumption that the computation of every (block-)integral is equally expensive we balance the assembly of the operator on level J. Since we use a dynamic integration scheme [8] this assumption does not hold exactly but our experiments indicate that the difference in the cost of the integration is small. A slightly better load balance might be achieved if we use the number of integration cells [8] per (block-)row instead of the number of (block-)entries, but still the number of quadrature points may not be balanced. Furthermore, the main influence on the number of quadrature cells is the number of neighboring patches [8].

Currently, our load estimator w_L involves only the neighbors $C_{L,J}$ on the finest level J. But for highly irregular point sets we might need to include an estimate of the computational work on coarser levels as well. To this end we could either include the number of neighbors $\mathrm{card}(C_{L,k})$ on coarser levels $k < J$ or take the local refinement level of the tree into account. Furthermore, the estimator does not involve the local polynomial degrees p_i which influence the cost during the integration. In applications with a large variation of the local polynomial degrees p_i or varying local basis functions ψ_i^n the estimator should also take these features into account.

Note that the computational cost associated with the estimation of the current load can often be reduced. In a time-dependent setting or in adaptive refinement we usually have a pretty good load estimate from a previous time step or a coarser level without extra computations. This estimate can either be used directly to partition the data or it can be updated with only a

[5] The complexity may be reduced to $O(\frac{\mathrm{card}(P_J)}{\wp} + \log \wp)$ only under very restrictive assumptions on the load imbalance.

few operations. Furthermore, we typically have to re-distribute only a small amount of data in these situations.

Let us now consider the solution phase of our PUM where we use our multilevel iteration to solve the linear (block-)system $A_J \tilde{u}_J = \hat{f}_J$. The solver essentially consist of matrix-vector-products and scalar-products. So we need to be concerned with the performance of these two basic operations.

Our load balancing strategy partitions the number of (block-)integrals evenly among the processors so that we have an optimal load balance in the assembly of the stiffness matrix. Hence, the number of (block-)entries in the stiffness matrix A_J per processor are also (almost) identical due to this balancing strategy, i.e. the number of operations in a matrix-vector-product is balanced among the processors. Unlike in grid-based discretizations we have to cope with a varying "stencil size", i.e. the number of (block-)entries per (block-)row in the stiffness matrix is not constant. Therefore, the perfect load balance for the matrix-vector-product does no longer coincide with the load balance for the scalar-product. Since a matrix-vector-product is certainly more expensive than a scalar-product the parallel performance of the overall iteration is dominated by the performance of the matrix-vector-product where we have a perfect load balance. Hence, our balancing scheme leads to an optimal load balance in the discretization phase as well as in the solution phase.

3.5 Parallel Cover Construction

Now that the computational domain is partitioned in an appropriate fashion among the processors we turn to the algorithmic changes for our parallel implementation, e.g. the computation of the communication pattern. The first task in our PUM is the multilevel cover construction [8, 9] which is essentially a post-order tree operation. Due to our tree decomposition which assigns complete sub-trees to processors most work can be done completely in parallel. When we reach elements of the common global tree we need to gather the respective tree cells from remote processors. Then, all processors can complete the cover construction on the common global tree. The parallel version of the multilevel cover construction algorithm [8, 9] reads as:
Algorithm 2 (Parallel Multilevel Cover Construction).

1. Given the domain $\Omega \subset \mathbb{R}^d$ and a bounding box $R_\Omega = \bigotimes_{i=1}^d [l_\Omega^i, u_\Omega^i] \supset \overline{\Omega}$.
2. Given the interval boundaries $\{r_q \,|\, q = 0, \ldots, \wp\}$ and the local part $\tilde{P}_{\hat{q}}$ of the initial point set $\tilde{P} = \{x_j \,|\, x_j \in \overline{\Omega}\}$, i.e. $\mathsf{k}_j^D \in [r_q, r_{q+1})$ for all $x_j \in \tilde{P}_{\hat{q}}$.[6]
3. Initialize the common global d-binary tree (quadtree, octree) according to the $\wp$ intervals $[r_q, r_{q+1})$.

[6] An initial partition can easily be constructed by choosing uniform interval boundaries $\{r_q\}$ and partitioning the initial point set $\tilde{P}$ according to the domain keys on the finest possible tree level.

4. Build parallel d-binary sub-trees over local sub-tree roots, such that per leaf L at most one $x_i \in \tilde{P}_{\hat{q}}$ lies within the associated cell $\mathcal{C}_L := \bigotimes_{i=1}^d [l_L^i, u_L^i]$.
5. Set J to the finest refinement level of the tree.
6. For all local sub-tree roots $\mathcal{C}_L = \bigotimes_{i=1}^d [l_L^i, u_L^i]$:
   (a) If current tree cell $\mathcal{C}_L$ is an INNER tree node:
      i. Descend tree for all successors of $\mathcal{C}_L$.
      ii. Set patch $\omega_L = \bigotimes_{i=1}^d [x_L^i - h_L^i, x_L^i + h_L^i] \supset \mathcal{C}_L$ where $x_L = \frac{1}{2^d}\sum x_S$ is the center of its successors points x_S and $h_L^i = 2\max h_S^i$ is twice the maximum radius of its successors h_S^i.
      iii. Set active levels $l_L^{\min} = l_L^{\max} = \min l_S^{\min} - 1$ and update for all successors $l_S^{\min} = \min l_S^{\min}$.

   (b) Else:
      i. Set patch $\omega_L = \bigotimes_{i=1}^d [x_L^i - h_L^i, x_L^i + h_L^i] \supset \mathcal{C}_L$ where $x_L^i = l_L^i + \frac{1}{2}(u_L^i - l_L^i)$ and $h_L^i = \frac{\alpha_l}{2}(u_L^i - l_L^i)$.
      ii. Set active levels $l_L^{\min} = l_L^{\max} = J$.
7. Broadcast patches ω_L associated with local sub-tree roots $\mathcal{C}_L$ to all processors.
8. For common global root $\mathcal{C}_L = \bigotimes_{i=1}^d [l_L^i, u_L^i]$:
   (a) If current tree cell $\mathcal{C}_L$ is not the root of any complete processor sub-tree and an INNER tree node:
      i. Descend tree for all successors of $\mathcal{C}_L$.
      ii. Set patch $\omega_L = \bigotimes_{i=1}^d [x_L^i - h_L^i, x_L^i + h_L^i] \supset \mathcal{C}_L$ where $x_L = \frac{1}{2^d}\sum x_S$ is the center of its successors points x_S and $h_L^i = 2\max h_S^i$ is twice the maximum radius of its successors h_S^i.
      iii. Set active levels $l_L^{\min} = l_L^{\max} = \min l_S^{\min} - 1$ and update for all successors $l_S^{\min} = \min l_S^{\min}$.
9. For $k = 0, \ldots, J$:
   (a) Set $P_{\Omega \hat{q}}^k = \{x_L \,|\, l_L^{\min} \le k \le l_L^{\max} \text{ and } \mathsf{k}_L^D \in [r_{\hat{q}}, r_{\hat{q}+1})\}$.
   (b) Set $C_{\Omega \hat{q}}^k = \{\omega_L \,|\, l_L^{\min} \le k \le l_L^{\max} \text{ and } \mathsf{k}_L^D \in [r_{\hat{q}}, r_{\hat{q}+1})\}$.

Here, the parameter α_l in step 6(b)i is only dependent on the order l of the spline $\mathcal{W}$ used in the construction of the PU, see [8]. Throughout this paper we use a linear spline $\mathcal{W}$ to generate the partition of unity with $\alpha_l = 1.3$. Note that this cover construction algorithm introduces additional points, i.e. $\operatorname{card}(P_J) = \sum_{q=0}^{\wp-1} \operatorname{card}(P_{Jq}) \ge \operatorname{card}(\tilde{P}) =: \tilde{N}$, to insure the shape regularity of the cover patches, see [8] for details. Also note that we have assumed $R_\Omega = \overline{\Omega}$ for ease of notation in Algorithm 2, in general we only need to consider tree cells $\mathcal{C}_L$ which overlap the domain Ω, i.e. $\mathcal{C}_L \cap \Omega \neq \emptyset$. The complexity of this parallel multilevel cover construction including the setup of the tree is given by $O(\frac{\operatorname{card}(P_J)}{\wp} J + \wp \log \wp)$.

3.6 Parallel Matrix Assembly

Now that we have constructed the covers C_Ω^k in a distributed fashion, we come to the Galerkin discretization of (2.3) in parallel. Here, we simply restrict the assembly of the stiffness matrix (and the transfer operators) on each of the $\wp$ processors to the (block-)rows associated with its assigned patches $\omega_{i,k}$. A processor $\hat{q}$ computes all (block-)entries

$$(A_k)_{i,j} = \left(A_{k\,(i,n),(j,m)}\right), \text{ with } A_{k\,(i,n),(j,m)} = a\left(\varphi_{j,k}\psi_{j,k}^m, \varphi_{i,k}\psi_{i,k}^n\right) \in \mathbb{R}, \tag{3.2}$$

where $\varphi_{i,k}$ is the PU function associated with one of its assigned patches $\omega_{i,k}$, i.e. the domain key $\mathsf{k}_{i,k}^D = \mathsf{k}_i^D$ associated with the patch $\omega_{i,k}$ is element of $[r_{\hat{q}}, r_{\hat{q}+1})$. The (block-)sparsity pattern of the respective (block-)row is determined by the neighborhood $C_{i,k} = \{\omega_{j,k} \in C_\Omega^k \,|\, \omega_{i,k} \cap \omega_{j,k} \neq \emptyset\}$. Hence, a processor needs to access all geometric neighbors $\omega_{i,k} \cap \omega_{j,k} \neq \emptyset$ of its patches $\omega_{i,k}$ to compute its assigned part of the stiffness matrix A_k on level k. In fact these neighbors are already needed to evaluate the local PU functions (2.2).

Although most neighbors $\omega_{j,k}$ of a patch $\omega_{i,k}$ are stored on the local processor, the patch $\omega_{i,k}$ may well overlap patches which are stored on a remote processor. Hence, a processor may need copies of certain patches from a remote processor for the assembly of its assigned (block-)rows of the global stiffness matrices A_k. The computation of a single (block-)entry (3.2) involves $\varphi_{i,k}$ and $\varphi_{j,k}$. Hence, it seems that we not only need remote patches $\omega_{j,k}$ but also all their neighbors $\omega_{l,k} \in C_{j,k}$ for the evaluation of the integrands (3.2) involved in the (block-)row corresponding to the local patch $\omega_{i,k}$. This would significantly increase the communication volume and storage overhead due to parallelization. But since all function evaluations of $\varphi_{j,k}$ are restricted to the support of $\varphi_{i,k}$—recall that the integration domain for the block entry is $\Omega \cap \omega_{i,k} \cap \omega_{j,k}$—every neighboring patch $\omega_{l,k} \in C_{j,k}$ that contributes a nonzero weight $W_{l,k}$ to the PU function $\varphi_{j,k}$ (on the integration domain) must also be a neighbor of $\omega_{i,k}$. Hence, it is sufficient to store copies of remote patches $\omega_{j,k}$ which are direct neighbors of a local patch $\omega_{i,k}$. There is no need to store neighbors of neighbors for the assembly of the stiffness matrix.

But how does a processor determine which neighbors $\omega_{j,k}$ exist on a remote processor? A processor cannot determine which patches to request from a remote processor. But a processor can certainly determine which of its local patches $\omega_{i,k}$ overlap the remote sub-trees. Hence, a processor can compute which local patches a remote processor may need to complete its neighbor search. We only need to perform a parallel communication step where a processor sends its local patches which overlap the remote sub-trees prior to the computation of the neighborhoods $C_{i,k}$.

Our cover construction algorithm constructs patches with increasing overlap on coarser levels $k < J$ to control the gradients $\nabla\varphi_{i,k}$ for $k < J$. Hence, many local patches $\omega_{i,\hat{k}}$ will overlap a remote sub-tree root patch $\omega_{j,\tilde{k}}$. But for the computation of the neighborhoods $C_{j,\hat{k}}$ on level $\hat{k} > \tilde{k}$ the remote processor may not need the local patch $\omega_{i,\hat{k}}$. The remote patches $\omega_{j,\hat{k}}$ on level

$\hat{k}$ might not overlap $\omega_{i,\hat{k}}$, even though the coarser patch $\omega_{j,\tilde{k}}$ does overlap $\omega_{i,\hat{k}}$. Hence, the patch $\omega_{i,\hat{k}}$ is not needed by the remote processor to complete its computation and $\omega_{i,\hat{k}}$ should not be sent. This problem can be easily cured if we first compute a "minimal cover". Here, the patches associated with the tree cells are computed without increasing the overlap from level to level. This computation of patches with "minimal" overlap can be done with a variant of Algorithm 2. We only need to change steps 6(a)ii and 8(a)ii of Algorithm 2 where we set the patches on coarser levels. Then, we store separate copies of the minimal patches associated with the leaves of the common global tree before we compute the correct cover with Algorithm 2. A processor can now test its local patches with the correct supports against the "minimal" patches associated with remote sub-tree roots to compute the correct overlap with respect to the finest level J. The complexity of this overlap computation is given by $O(J(\log \wp)^2)$ and the communication volume is of the order $O((\frac{\mathrm{card}(P_J)}{\wp})^{\frac{d-1}{d}})$.[7]

For the computation of the neighborhoods $C_{i,k}$ on coarser levels $k < J$ we have to keep in mind that the complete tree is coarsened from level to level. Hence, we may need to coarsen the common global tree and we also have to update the minimal overlaps.[8]

For the interlevel transfer operators we have to compute interlevel neighbors $C_{i,k,k-1} := \{\omega_{j,k-1} \in C_\Omega^{k-1} \mid \omega_{i,k} \cap \omega_{j,k-1} \neq \emptyset\}$ and $C_{i,k,k+1} := \{\omega_{j,k+1} \in C_\Omega^{k+1} \mid \omega_{i,k} \cap \omega_{j,k+1} \neq \emptyset\}$ for all local patches $\omega_{i,k}$. Hence, we need to compute overlaps within a given level as well as between successive levels. The overlaps between between different levels can be computed in a similar fashion as described above. After the exchange of the overlaps the neighbor search can be completed on each processor just like in a sequential implementation. The complexity of the neighborhood computation is given by $O(\frac{\mathrm{card}(P_J)}{\wp} J)$.

In our implementation we pre-compute the neighborhoods $C_{i,k}$ on all levels $k = 0, \ldots, J$ prior to the assembly of the stiffness matrices A_k. These neighborhoods, i.e. the respective keys, are stored in an additional sparse data structure since they not only determine the sparsity pattern of the stiffness matrix but they are also needed for the function evaluation of the PU functions $\varphi_{i,k}$. Hence, we compute the neighborhoods $C_{i,k}$ only once and utilize the $O(1)$ random access capabilities of our key-based tree implementation so that the single function evaluation of $\varphi_{i,k}$ is of the order $O(1)$. The interlevel neighbors $C_{i,k,l}$ with $k \neq l$ are computed on demand during the

7 The complexity of the overlap computation may be reduced to $O(J \log \wp)$ if we employ a second tree data structure to store a complete copy of the common global tree.

8 Under certain constraints on the overlap parameter α in the cover construction and the regularity of the tree we can compute the neighborhoods $C_{i,k}$ on coarser levels $k < J$ directly from the neighborhoods $C_{i,J}$ on the finest level J and there is no need for an overlap computation of coarser levels. But this does not improve the overall complexity since we still need to search for neighbors on the finest level J.

assembly of the respective transfer operators I_k^l and I_l^k since they are needed for the sparsity structure of the transfer operators only. Hence, the assembly of the stiffness matrices is of the order $O(\frac{\mathrm{card}(P_J)}{\wp})$ whereas the assembly of the transfer operators is of the order $O(\frac{\mathrm{card}(P_J)}{\wp} J)$ due to the necessary neighbor search.

3.7 Parallel Multilevel Solution

The first challenge we encounter in the parallelization of our multilevel solver is the question of smoothing in parallel. In [9] we have used a (block-)Gauß–Seidel iteration as a smoother since its smoothing rate is superior to that of the simpler (block-)Jacobi smoother. The parallelization of a (block-)Gauß–Seidel iteration though is not an easy task especially for unstructured discretizations such as ours. A common approach to circumvent the complete parallelization of the (block-)Gauß–Seidel smoother is a sub-domain-blocking approach. Here, the (block-)Gauß–Seidel iteration is only applied locally within a processor's assigned sub-domain and these local iterates are then merged using an outer sub-domain-block-Jacobi iteration. Note that this approach changes the overall iteration for different numbers of sub-domains, i.e. varying processor numbers. The rate of this composite sub-domain-block-Jacobi smoother with an internal (block-)Gauß–Seidel iteration is somewhat reduced compared with the original (block-)Gauß–Seidel rate but it is still superior to that of the (block-)Jacobi iteration (for large sub-domains). The number of operations of this composite smoother is similar to the number of operations of a (block-)Jacobi iteration. Their communication demands are identical. Hence, the composite sub-domain-block-Jacobi smoother with internal (block-)Gauß–Seidel iteration (in general) outperforms the (block-)-Jacobi smoother and it is therefore used in our multilevel solver.

The second basic operation of our multilevel iteration is the application of the prolongation and restriction operators. In our implementation we completely assemble the prolongation as well as the restriction operators in an analogous fashion as described above for the stiffness matrices A_k. This increases somewhat the storage overhead but on the other hand we do not need an explicit transposition or a transpose matrix-vector-product in parallel. We only need a parallel matrix-vector-product to transfer information between levels.

Since we assign complete sub-trees to a processor most (block-)coefficients per processor are stored locally. Therefore the communication volume in the smoother as well as in the interlevel transfer is small. In [9] we have developed and tested several interlevel transfer operators I_{k-1}^k. First, a global L^2-projection between the involved PUM spaces V_{k-1}^{PU} and V_k^{PU} which turned out to be too expensive to be used in practice. Then, a localized L^2-projection, the so-called Global-to-Local projection, which showed essentially the same approximation qualities as the global L^2-projection at lower computational costs and is applicable to any sequence of PUM spaces. Finally, a completely localized L^2-projection which utilizes our tree based cover construction and is extremely cheap to assemble but also shows similar approximation properties as the global L^2-projection. Furthermore, this so-called Local-to-Local projection has a minimal (block-)sparsity pattern. The Local-to-Local projection therefore has an especially simple communication demand. Here, a (block-)-row of the restriction operator only consists only of a single (block-)entry which corresponds to the coarser cover patch associated with the ancestor tree-cell of the current fine level patch. Most of these ancestors are located on the same processor as the current patch due to our partition of the tree. Hence, the application of the Local-to-Local transfer operators involve very little communication. This though does not change the overall complexity of our parallel multilevel solver which is $O(\frac{\mathrm{card}(P_J)}{\wp} + (\frac{\mathrm{card}(P_J)}{\wp})^{\frac{d-1}{d}} + J + \log \wp)$ as usual.

4 Numerical Results

The model problem we apply our multilevel PUM to is the PDE

$$-\Delta u + u = 0 \text{ in } \Omega = (0,1)^2 \tag{4.1}$$

of Helmholtz type with vanishing Neumann boundary conditions $\nabla u \cdot n_\Omega = 0$ on $\partial\Omega$. In all our experiments we use a linear normed B-spline as the generating weight function $\mathcal{W}$ for the PU construction and $\alpha_l = 1.3$. The initial partition of the domain is a uniform decomposition, i.e. the common global tree is a uniform refined tree with at least $\wp$ leaves. We assign the same number of leaves of the common global tree to each processor. This can be achieved by setting the initial interval boundaries $r_q = q h_{\mathsf{key}}$ where h_{key} is only dependent on the dimension d, the number of processors $\wp$ and the maximal key $\mathsf{k}_{\max}$ (i.e. the bit length of $\mathsf{k}_{\max}$). The given point set $\tilde{P} = \{x_j \,|\, x_j \in \overline{\Omega}, j = 1, \ldots, \tilde{N}\}$ is then partitioned using the finest possible domain keys k_j^D and the uniform interval boundaries $\{r_q\}$. All computations were carried out on the Parnass2 cluster[9] [20] built by our department.

We are concerned with the scaling behavior of the overall parallel algorithm. To this end we (approximately) fix the computational load per processor; i.e as we increase the number of processors $\wp$ we also increase the global work load. Note that we cannot exactly prescribe the computational load since our cover construction introduces additional points, i.e. $\mathrm{card}(P_J) \geq \mathrm{card}(\tilde{P}) = \tilde{N}$. Therefore we expect to see some fluctuations in our measurements which stem from the irregularity of the initial point sets $\tilde{P}$.

We consider several values for the local $\mathsf{load} = \frac{\tilde{N}}{\wp}$ per processor in our experiments. Here, we measure wall clock times for different parts of the

[9] Parnass2 consist of 72 dual processor PCs connected by a Myrinet.

overall algorithm. Our parallel PUM can be split into three major parts. First the computation of the load estimate w and the balancing step (see §3.4). Then, the discretization step where we assemble the discrete operators and the transfer operators on all levels (see §3.6). Finally, the solution step where we solve the linear (block-)system with a multiplicative multilevel iteration (see §3.7).

Example 1 (Halton points). In our first experiment we use a Halton[10] sequence with $\tilde{N}$ points as the initial point set $\tilde{P}$ for our cover construction. The local approximation spaces $V_{i,k}^{p_{i,k}}$ we use in this experiment are linear Legendre polynomials, i.e. we choose $p_{i,k} = 1$ for all i and k. Hence, the number of degrees of freedom dof in our two dimensional example is given by $\mathsf{dof} = 3\,\mathrm{card}(P_J)$ where J is denotes the finest discretization level.

Since the distribution of a Halton point set is uniform our d-binary tree will be balanced and our initial uniform data partition is close to the optimal data partition. Here, we need to redistribute only few data. Hence, it is reasonable to study the scaling behavior of the load balancing step itself. In general, when we have a significant load imbalance, the balancing step, i.e. the computation of the load estimate w, cannot scale since the respective operations are completed on an inappropriate data partition.

Load Balancing. The load balancing step consists of several parts with different scaling behavior. At first we have to compute the cover based on the *initial* data distribution. This post-order operation involves a gather communication step where only very few data have to be sent/received. Therefore, we expect a perfect scaling behavior. The execution times should stay (almost) constant since the amount of work per processor is (almost) constant. This behavior can be observed in Figure 4 (upper left) where we have plotted the measured wall clock times against the number of processors for varying local loads. Although our current load estimator involves only the neighbors on the finest level, we compute the overlap on all levels. This is essentially a reduced neighbor search operation on all levels. Roughly speaking, we determine the surface of our partition on all levels. The computation of the overlap is of the order $O(J(\log \wp)^2)$, see Figure 4 (upper center). In the communication step the computed overlaps are exchanged between the processors. The communication volume is of order $O((\frac{\mathrm{card}(P_J)}{\wp})^{\frac{d-1}{d}})$, see Figure 4 (upper right). From both graphs we can observe that the anticipated scaling behavior is reached for a larger number of processors only. For smaller processor numbers the space filling curve partitioning scheme leads to sub-domains with a relatively large number of geometric neighbors, so that we find an all-to-all communication pattern for small processor numbers. The neighbor search on all levels

[10] Halton–sequences are quasi Monte Carlo sequences with a uniform distribution, which are used in sampling and numerical integration. Consider $n \in \mathbb{N}_0$ given as $\sum_j n_j p^j = n$ for some prime p. We can define the transformation H_p from $\mathbb{N}_0$ to $[0,1]$ with $n \mapsto H_p(n) = \sum_j n_j p^{-j-1}$. Then, the (p,q) Halton–sequence with $\tilde{N}$ points is defined as $\mathrm{Halton}_0^{\tilde{N}}(q,p) := \{(H_p(n), H_q(n)) \,|\, n = 0, \dots, N\}$.

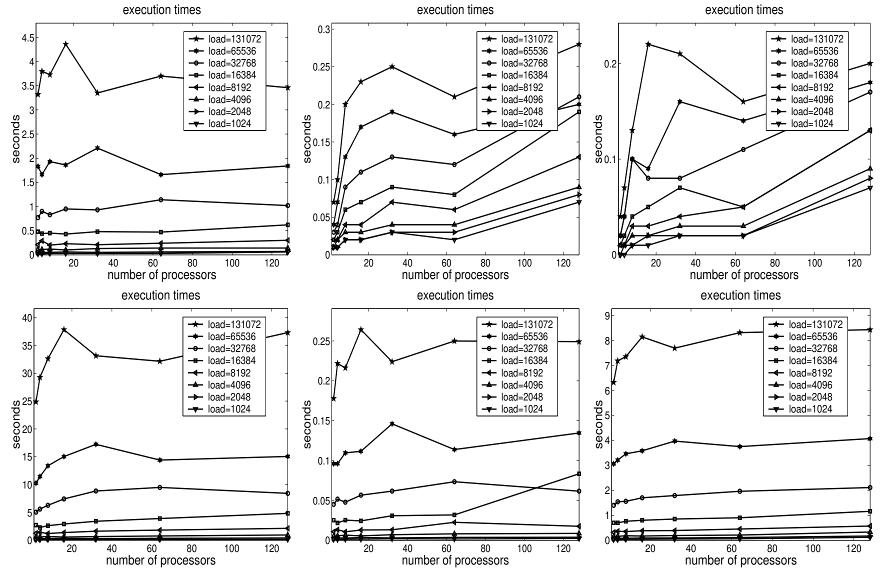

Figure4. Our load balancing step with a weighted space filling curve consists of several parts with different scaling behavior. The cover construction (upper left), the computation of the communication pattern and the overlaps on all levels (upper center), the actual communication of the overlaps (upper right), the computation over all neighbors on all levels (lower left), the update of the data partition $\{r_q\}$ (lower center), and the redistribution of the data including a rebuild of the tree (lower right).

is of the order $O(\frac{\mathrm{card}(P_J)}{\wp}J)$. From the graphs given in Figure 4 (lower left) we can observe only a slight logarithmic scaling behavior.

Note that all these steps are necessary just to compute the load estimate. The actual load balancing step involves only the leaves of the tree and the reduction operation over the interval boundaries. We can observe a very slight logarithmic scaling behavior from the graph depicted in Figure 4 (lower center). After the update of the interval boundaries we have to redistribute the data and insert the data into a tree. Here, we chose to rebuild the complete tree and hence we expect a logarithmic scaling behavior. We can observed a slight logarithmic scaling from the graphs given in Figure 4 (lower right).

Note that about 90% of the total execution time of the complete load balancing phase is spent in the computation of a good load estimate and its associated communication. The balancing step itself involves only a negligible amount of compute time. In a time-dependent setting or in adaptive refinement we might have a pretty good load estimate from the previous time step or previous refinement level and may therefore not need to compute the current load. Yet the computational work associated with our load estimator

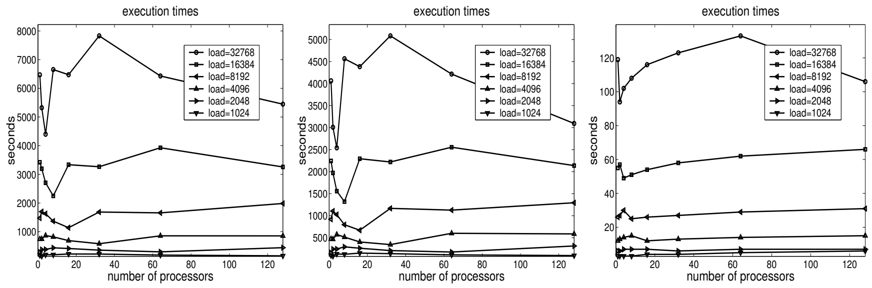

Figure5. Setup times for the assembly of the operator (left), the Global-to-Local transfers (center), and the Local-to-Local transfers (right) on all levels.

is completely negligible compared with the assembly of the stiffness matrices and transfer operators on all levels.

Galerkin Discretization. The next step of our PUM is the discretization phase. Since our load balancing step is aimed to balance the load in the assembly of the operator on the finest level we can now observe the quality of our load estimator and the resulting data partition. Note that the assembly of all operators can be completed without any communication at all.

The timing results are given in Table 2 for the stiffness matrix and the transfer operators, i.e. displayed are the sum of the assembly times on all levels $k = 0, \ldots, J$. The scaling behavior of these assembly steps are depicted in Figure 5. From the numbers displayed in Table 2, we can observe a perfect speed-up for the operator setup as well as for both interlevel transfer operators. Note that the setup times for the interlevel transfer operators includes the assembly times for the prolongation as well as for the restriction.

The scaling behavior of the assembly phase is as expected (almost) perfect. Here, we see some fluctuations in the execution times due to the effect the irregularity of the initial point set has on the number of neighbors. We can observe that the curves for the operator assembly (Figure 5 (left)) and the transfer operators based on the Global-to-Local projection (Figure 5 (center)) are very similar, whereas the curve for the Local-to-Local projections is somewhat different (Figure 5 (right)). This is due to the fact that the Global-to-Local projection involves all geometric neighboring patches just like the operator. The Local-to-Local projection, however, involves hierarchical neighbors [9] only. Hence, the fluctuations in the execution times for the Local-to-Local projections are due to variations in the number of levels of the *global* tree only. But the fluctuations in the measured wall clock times for the assembly of the operators and the Global-to-Local projections stem from the variation of the *local* refinement levels which essentially determine the number of neighbors. Furthermore, the assembly of the Global-to-Local transfers involves the integration of more complicated integrals than the assembly of the Local-to-Local transfers, see [9]. Hence, the logarithmic complexity of the neighbor search completed in the assembly of both types of transfer operators is not really visible in the curves for the Global-to-Local transfers and just slightly visible in the assembly of the Local-to-Local transfers. Note that the improvement in the execution times for 128 processors with $\mathsf{load} = 32768$ again comes from the use of the Halton point set. Here, the resulting cover is closer to the uniform situation and the overall storage utilization is better than for smaller processor numbers.

Multilevel Solution. The parallel scaling behavior of our multilevel solver should essentially be the same as that of classical multigrid methods for mesh-based discretizations which has been studied in many articles [2, 3, 5, 6, 10, 11, 13, 14, 15, 21, 22]. A classical parallel multigrid solver scales with $O(\frac{\mathsf{dof}}{\wp} + (\frac{\mathsf{dof}}{\wp})^{\frac{d-1}{d}} + \log(\mathsf{dof}) + \log(\wp))$. Hence, we expect to see a logarithmic scaling behavior of our multilevel solver as well.

We measure the wall clock times for the solution of the linear system and divide it by the number of completed iterations r to get the average cycle time. The initial value $\tilde{u}_0$ for the multilevel iteration is random valued with $\|\tilde{u}_0\| = 1$. The stopping criterion for the iteration is $\|\tilde{u}_r\| < 10^{-10}$ or $r > 50$. The error reduction rate of the iteration is therefore given by $\rho := \|\tilde{u}_r\|^{\frac{1}{r}}$.

The scaling behavior of the average cycle times are depicted in Figure 6 for $V(\mu, \mu)$-cycles with $\mu = 1, 2, 3$. These graphs show only a very slight logarithmic scaling behavior of our multilevel solver. Note that the multilevel iteration with the Local-to-Local transfer operators is about 10% faster than the corresponding cycle based on the Global-to-Local transfers. This is due to the different (block-)sparsity patterns of the transfer operators. The application of the Local-to-Local transfer operators involves less computational work and less communication. The difference in the cycles times will probably be more severe on parallel machines with a less balanced ratio of bandwidth to flops.

The different transfer operators have almost no effect on the error reduction rates ρ, see [9] for further details. But the two different smoothers, the simple (block-)Jacobi smoother and the composite sub-domain-block-Jacobi smoother with internal (block-)Gauß–Seidel iteration, certainly influence the overall rates ρ. Yet, both smoothers require a similar number of operations and the same communication so that the cycle times are essentially the same for both smoothers.

The error reduction rates ρ using the (block-)Jacobi smoother are not effected by a change of the number of processors since the overall algorithm stays the same independent of the number of processors. But the rates for the composite smoother are dependent on the number of processors. For a $V(1,1)$-cycle we measure reduction rates ρ between 0.3 and 0.4 for the composite smoother, and rates of about 0.6 for the (block-)Jacobi smoother. The rates for a $V(2,2)$-cycle are between 0.1 and 0.2 for the composite smoother, and about 0.3 for the (block-)Jacobi smoother. With three smoothing steps we

Table2. Setup times for the assembly of the operators (upper), the Global-to-Local projections (center), and the Local-to-Local projections (lower) on all levels.

problem size		number of processors							
$\hat{N}$	card(P_J)	1	2	4	8	16	32	64	128
Operator Setup									
1024	1678	178							
2048	3550	289	149						
4096	6454	742	376	196					
8192	14635	1476	744	381	196				
16384	27412	3423	1699	866	437	223			
32768	63388	6471	3196	1628	819	415	221		
65536	101167	10783	5323	2706	1369	690	357	185	
131072	231967	17525	8647	4399	2248	1138	581	297	157
262144	419194			13248	6655	3336	1686	855	442
524288	949798				12939	6474	3265	1658	853
1048576	1837612					15626	7830	3927	1984
2097152	4301116						12872	6431	3258
4194304	6496153							10748	5447
8388608	14252920								8471
Global-to-Local Transfer Setup									
1024	1678	100							
2048	3550	160	83						
4096	6454	474	243	132					
8192	14635	920	468	245	126				
16384	27412	2244	1106	572	293	152			
32768	63388	4063	1972	1025	515	262	139		
65536	101167	6195	3009	1558	796	404	209	112	
131072	231967	10141	4925	2542	1321	671	344	176	96
262144	419194			9141	4565	2294	1168	601	313
524288	949798				8852	4384	2220	1128	587
1048576	1837612					6751	5084	2555	1294
2097152	4301116						5656	4216	2139
4194304	6496153							5950	3095
8388608	14252920								4719
Local-to-Local Transfer Setup									
1024	1678	3							
2048	3550	6	3						
4096	6454	12	6	3					
8192	14635	26	13	7	3				
16384	27412	55	27	14	7	4			
32768	63388	119	57	30	15	7	4		
65536	101167	196	94	49	25	12	6	5	
131072	231967	411	196	102	51	26	13	7	6
262144	419194			214	108	54	27	14	7
524288	949798				231	116	58	29	15
1048576	1837612					244	123	62	31
2097152	4301116						263	133	66
4194304	6496153							210	106
8388608	14252920								208

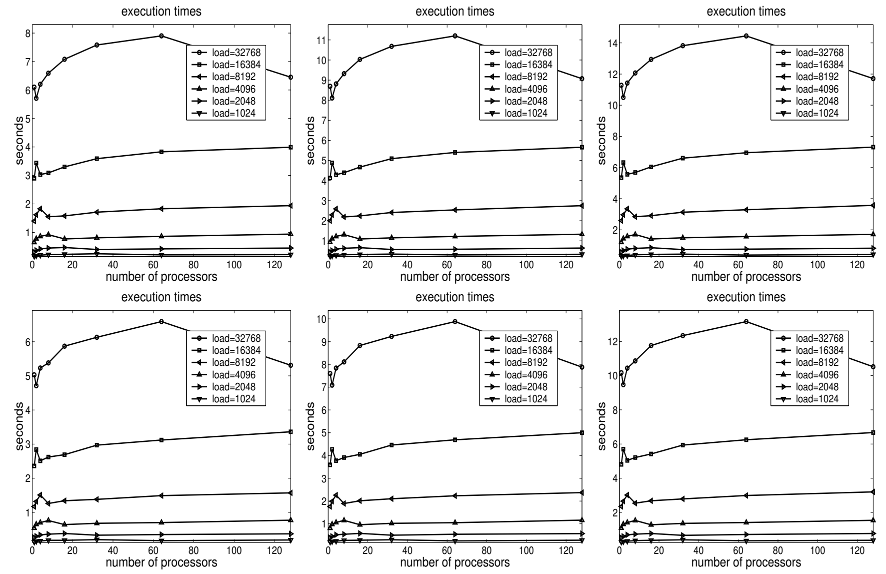

Figure6. Execution times for a single $V(\mu,\mu)$-cycle with the Global-to-Local projections (upper) and the Local-to-Local projections (lower). With $\mu = 1$ (right), $\mu = 2$ (center), and $\mu = 3$ (right).

measure rates ρ of about 0.1 for the composite smoother, and about 0.2 for the (block-)Jacobi smoother.

Example 2 (Graded Halton points). In our second experiment we use the same local approximation spaces $V_{i,k}^{p_{i,k}}$, the linear Legendre polynomials, but we use a more irregular initial point set $\tilde{P}$. Here, we use a grading function $\mathcal{G}$ to transform a Halton point set. The resulting transformed points are then used as the initial point set $\tilde{P}$ for the cover construction. We use the grading function

$$\mathcal{G} : \xi = (\xi_i)_{i=1}^d \in [0,1]^d \mapsto a(\xi)\xi \in [0,1]^d \text{ with } a(\xi) = \begin{cases} \|\xi\|_2 & \text{if } \|\xi\|_2 \leq 1 \\ 1 & \text{else} \end{cases} .$$

Here, our initial uniform decomposition is completely inappropriate and many data have to be re-balanced. The number of initial points strongly varies among the processors, e.g. for $\tilde{N} = 1048576$ and $\wp = 64$ we find from $\mathrm{card}(P_{J\hat{q}}) = 14206$ to $\mathrm{card}(P_{J\hat{q}}) = 214654$ points of the finest point set P_J on level $J = 21$ on different processors $\hat{q}$ prior to our load balancing. Therefore, we cannot expect the load balancing step itself to scale. The purpose of the balancing step is to resolve the load imbalance at low cost so that the overall algorithm utilizes all available resources for the larger part of the computation. Note that the overall execution time for the complete

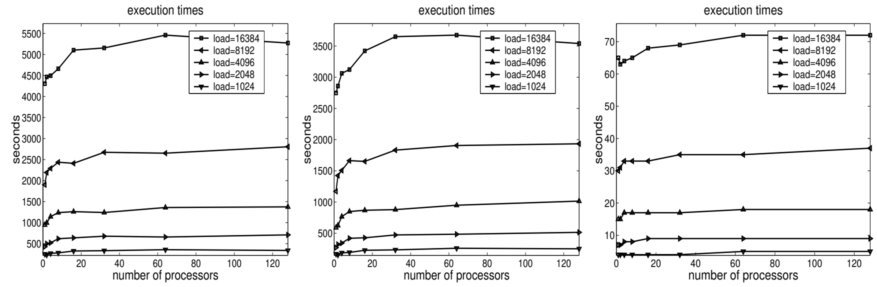

Figure7. Setup times for the assembly of the operator (left), the Global-to-Local transfers (center), and the Local-to-Local transfers (right) on all levels.

balancing step amounts to less than 1% of the time spent in the assembly of the discrete operators.

The measured wall clock times for the discretization phase of our PUM for the graded Halton point set are given in Table 3. From these numbers we can again observe a perfect speed-up for the operator setup as well as for both interlevel transfer operators. The scaling behavior of these assembly steps are depicted in Figures 7. Again, we see the anticipated perfect scaling behavior of the setup phase. Hence, our load balancing step certainly fulfills its purpose.

Also the scaling behavior of our multilevel solver (see Figure 8) is optimal. Here, the convergence rates for a $V(1,1)$-cycle with the Local-to-Local transfers and the composite smoother are between 0.3 and 0.5. The rates of the multilevel iteration with the simple (block-)Jacobi smoother are about $\rho = 0.6$ for the $V(1,1)$-cycle with the Global-to-Local transfers. Note that we may need more than one smoothing step when we use a simple (block-)Jacobi smoother together with the Local-to-Local transfers for highly irregular point sets, see [9] for details.

5 Concluding Remarks

We presented a parallel meshfree method for the discretization of an elliptic partial differential equation and the efficient parallel multilevel solution of the arising linear (block-)system. The main ingredients of our parallelization are a key-based tree implementation and the use of a space filling curve load balancing scheme.

The discretization phase where the discrete operators are assembled is the most expensive step in our particle-partition of unity method. This specific part of the method requires no communication at all and its scaling behavior is only dependent on the quality of the data partition. Hence, our load balancing scheme is aimed to balance this most expensive part of the method.

Table3. Setup times for the assembly of the operators (upper), the Global-to-Local projections (center), and the Local-to-Local projections (lower) on all levels.

problem size		number of processors							
N	card(P_J)	1	2	4	8	16	32	64	128
Operator Setup									
1024	1942	254							
2048	3643	430	236						
4096	7522	947	500	271					
8192	14986	1901	997	522	282				
16384	31495	4304	2194	1144	619	327			
32768	62542		4466	2292	1239	640	334		
65536	121966			4495	2437	1263	678	358	
131072	242506				4661	2411	1242	659	339
262144	484435					5103	2673	1361	710
524288	972988						5155	2652	1376
1048576	1976116							5460	2805
2097152	3976408								5272
Global-to-Local Transfer Setup									
1024	1942	168							
2048	3643	270	154						
4096	7522	590	321	186					
8192	14986	1171	624	343	192				
16384	31495	2748	1424	766	419	227			
32768	62542		2859	1507	848	427	234		
65536	121966			3062	1663	868	473	259	
131072	242506				3123	1651	879	484	250
262144	484435					3422	1832	949	514
524288	972988						3650	1906	1015
1048576	1976116							3675	1933
2097152	3976408								3540
Local-to-Local Transfer Setup									
1024	1942	4							
2048	3643	7	4						
4096	7522	15	7	4					
8192	14986	30	15	8	4				
16384	31495	65	31	17	8	4			
32768	62542		63	33	17	9	4		
65536	121966			64	33	17	9	5	
131072	242506				65	33	17	9	5
262144	484435					68	35	18	9
524288	972988						69	35	18
1048576	1976116							72	37
2097152	3976408								72

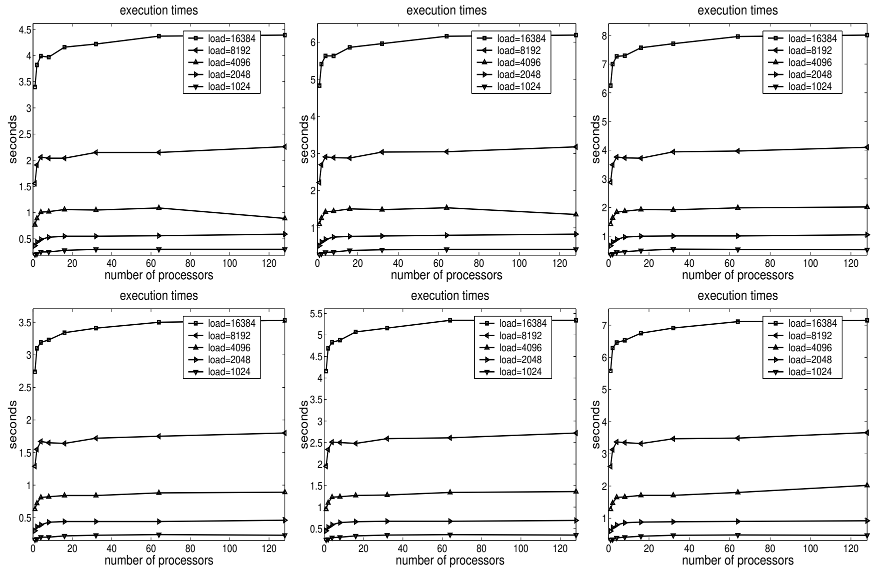

Figure8. Execution times for a single $V(\mu, \mu)$-cycle with the Global-to-Local projections (upper) and the Local-to-Local projections (lower). With $\mu = 1$ (right), $\mu = 2$ (center), and $\mu = 3$ (right).

The results of our numerical experiments showed that (within the expected fluctuations) we achieve a perfect scaling behavior. All other parts of the method, the computation of the load estimate, the setup of the transfer operators (at least the setup of the Local-to-Local transfers) and the multilevel solution are completely negligible with respect to execution times. Yet, our data partition also allows for the optimal scaling behavior of each of these steps. The presented space filling curve balancing scheme provides high quality data partitions independent of the number and distribution of points at a very low computational cost.

References

[1] I. Babuška, U. Banerjee, and J. E. Osborn, *Meshless and Generalized Finite Element Methods: A Survey of Some Major Results*, in Meshfree Methods for Partial Differential Equations, Lecture Notes in Computational Science and Engineering, Springer, 2002.

[2] P. Bastian, *Load Balancing for Adaptive Multigrid Methods*, SIAM J. Sci. Comp., 19 (1998), pp. 1303–1321.

[3] A. Brandt, *Multigrid Solvers on Parallel Computers*, in Elliptic Problem Solvers, M. H. Schultz, ed., Academic Press, 1981, pp. 39–83.

[4] A. Caglar, M. Griebel, M. A. Schweitzer, and G. W. Zumbusch, *Dynamic Load-Balancing of Hierarchical Tree Algorithms on a Cluster of Multiprocessor PCs and on the Cray T3E*, in Proceedings 14th Supercomputer Conference, Mannheim, H. W. Meuer, ed., Mannheim, Germany, 1999, Mateo.

[5] M. Griebel, *Parallel Domain-Oriented Multilevel Methods*, SIAM J. Sci. Comp., 16 (1995), pp. 1105–1125.

[6] M. Griebel and T. Neunhoeffer, *Parallel Point- and Domain-Oriented Multilevel Methods for Elliptic PDE on Workstation Networks*, J. Comp. Appl. Math., 66 (1996), pp. 267–278.

[7] M. Griebel and M. A. Schweitzer, *A Particle-Partition of Unity Method for the Solution of Elliptic, Parabolic and Hyperbolic PDE*, SIAM J. Sci. Comp., 22 (2000), pp. 853–890.

[8] ———, *A Particle-Partition of Unity Method—Part II: Efficient Cover Construction and Reliable Integration*, SIAM J. Sci. Comp., 23 (2002), pp. 1655–1682.

[9] ———, *A Particle-Partition of Unity Method—Part III: A Multilevel Solver*, SIAM J. Sci. Comp., (2002). to appear.

[10] M. Griebel and G. W. Zumbusch, *Hash-Storage Techniques for Adaptive Multilevel Solvers and their Domain Decomposition Parallelization*, in Domain Decomposition Methods 10, The 10th International Conference, Boulder, J. Mandel, C. Farhat, and X.-C. Cai, eds., vol. 218 of Contemp. Math., AMS, 1998, pp. 271–278.

[11] C. E. Grosch, *Poisson Solvers on Large Array Computers*, in Proc. 1978 LANL Workshop on Vector and Parallel Processors, B. L. Buzbee and J. F. Morrison, eds., 1978.

[12] F. C. Günther and W. K. Liu, *Implementation of Boundary Conditions for Meshless Methods*, Comput. Meth. Appl. Mech. Engrg., 163 (1998), pp. 205–230.

[13] J. E. Jones and S. F. McCormick, *Parallel Multigrid Methods*, in Parallel Numerical Algorithms, D. E. Keyes, A. Sameh, and V. Venkatakrishnan, eds., Kluwer Academic Publishers, 1997, pp. 203–224.

[14] O. A. McBryan, P. O. Frederickson, J. Linden, A. Schüller, K. Solchenbach, K. Stüben, C. A. Thole, and U. Trottenberg, *Multigrid Methods on Parallel Computers – A Survey of Recent Developments*, IMPACT Comput. Sci. Engrg., 3 (1991), pp. 1–75.

[15] W. F. Mitchell, *A Parallel Multigrid Method using the Full Domain Partition*, Electron. Trans. Numer. Anal., 6 (1997), pp. 224–233. Special Issue for Proc. of the 8th Copper Mountain Conf. on Multigrid Methods.

[16] J. Nitsche, *Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind*, Abh. Math. Univ. Hamburg, 36 (1970–1971), pp. 9–15.

[17] A. Pothen, *Graph Partitioning Algorithms with Applications to Scientific Computing*, in Parallel Numerical Algorithms, D. E. Keyes,

A. Sameh, and V. Venkatakrishnan, eds., Kluwer Academic Publishers, 1997, pp. 323–368.

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

[19] M. A. Schweitzer, *Ein Partikel–Galerkin–Verfahren mit Ansatzfunktionen der Partition of Unity Method*, Diplomarbeit, Institut für Angewandte Mathematik, Universität Bonn, 1997.

[20] M. A. Schweitzer, G. W. Zumbusch, and M. Griebel, *Parnass$_2$: A Cluster of Dual-Processor PCs*, in Proceedings of the 2nd Workshop Cluster-Computing, Karlsruhe, W. Rehm and T. Ungerer, eds., no. CSR-99-02 in Chemnitzer Informatik Berichte, Chemnitz, Germany, 1999, TU Chemnitz, pp. 45–54.

[21] K. Solchenbach, C. A. Thole, and U. Trottenberg, *Parallel Multigrid Methods: Implementation on SUPRENUM–like architectures and applications*, in Supercomputing, vol. 297 of Lecture Notes in Computer Science, Springer, 1987, pp. 28–42.

[22] L. Stals, *Parallel Implementation of Multigrid Methods*, PhD thesis, Department of Mathematics, Australian National University, 1995.

[23] M. S. Warren and J. K. Salmon, *A Parallel Hashed Oct-Tree N-Body Algorithm*, in Supercomputing '93, Los Alamitos, 1993, IEEE Comp. Soc., pp. 12–21.

[24] ——, *A Portable Parallel Particle Program*, Computer Physics Communications, 87 (1995).

[25] ——, *Parallel, Out-of-core Methods for N-body Simulation*, in Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, M. Heath, V. Torczon, G. Astfalk, P. . E. Bjørstad, A. H. Karp, C. H. Koebel, V. Kumar, R. F. Lucas, L. T. Watson, and D. E. Womble, eds., Philadelphia, 1997, SIAM.

[26] G. W. Zumbusch, *Simultanous h-p Adaptation in Multilevel Finite Elements*, dissertation, Fachbereich Mathematik und Informatik, FU Berlin, 1995.

[27] ——, *Adaptive Parallel Multilevel Methods for Partial Differential Equations*, Habilitation, Insitut für Angewandte Mathematik, Universität Bonn, 2001.

[28] ——, *On the Quality of Space-Filling Curve Induced Partitions*, Z. Angew. Math. Mech., 81 Suppl. 1 (2001), pp. 25–28.