1

A Sparse Grid PDE Solver; Discretization, Adaptivity, Software Design and Parallelization

G. W. Zumbusch[1]

ABSTRACT Sparse grids are an efficient approximation method for functions, especially in higher dimensions $d \geq 3$. Compared to regular, uniform grids of a mesh parameter h, which contain h^{-d} points in d dimensions, sparse grids require only $h^{-1}|\log h|^{d-1}$ points due to a truncated, tensor-product multi-scale basis representation. The purpose of this paper is to survey some activities for the solution of partial differential equations with methods based sparse grid. Furthermore some aspects of sparse grids are discussed such as adaptive grid refinement, parallel computing, a space-time discretization scheme and the structure of a code to implement these methods.

1.1 Introduction

Quite lot of phenomena in science and engineering can be modeled by boundary value problems of ordinary differential equation or partial differential equation type. Further assumptions to simplify the model like axis- and radial-symmetries often give rise to a PDE in one or two dimensions ($d = 1, 2$), which can be treated numerically more easily. However, the accurate solution of similar problems in three dimensions, or time-dependent problems in two or three space dimensions requires more computational power. Fortunately, most phenomena in physics live in a three-dimensional space, plus one time dimension. Higher-dimensional problems, which would be extremely expensive or simply impossible to solve numerically, referred to as the 'curse of dimension', due not occur in literature often. However, there are a lot of higher-dimensional problems $d > 3$ around, but due to their complexity only few approaches address these problems di-

[1]Institute for Applied Mathematics, University Bonn, Germany,
Email: `zumbusch@iam.uni-bonn.de`,
URL: `http://wwwwissrech.iam.uni-bonn.de/people/zumbusch.html`

rectly. We mention higher-dimensional problems in financial engineering, in quantum physics, in statistical physics and even the four-dimensional problems in general relativity. Hence there is a need for the solution of higher-dimensional problems, but due to the fact that standard methods fail or are extremely expensive, such problems are usually not considered.

1.2 Sparse grids

Sparse grids are a multi-dimensional approximation scheme, which is known under several names such as 'hyperbolic crosspoints', 'splitting extrapolation' or as a boolean sum of grids. Probably Smolyak [Smo63] was the historically first reference. Directly related to the boolean construction of the grids was the construction of a multi-dimensional quadrature formula. Both quadrature formulae and the approximation properties of such tensor product spaces were subject to further research, see Temlyakov [Tem89] and others. The curse of dimension was also subject to general research on the theoretical complexity of higher-dimensional problems. For such theoretical reasons, sparse grids play an important role for higher-dimensional problems. Besides the application to quadrature problems, sparse grids are now also used for the solution of PDEs.

Sparse grids were introduced for the solution of elliptic partial differential equations by Zenger [Zen91], where a Galerkin method, adaptive sparse grids and tree data structures were discussed. At the same time a different discretization scheme based on the extrapolation of solutions on several related, regular grids was proposed, see [GSZ92].

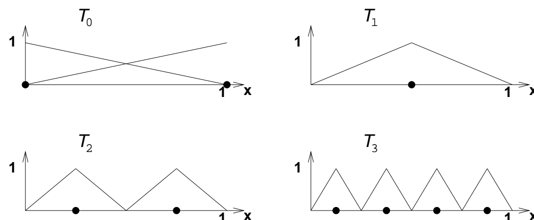

FIGURE 1.1. The hierarchical basis.

The multi-dimensional approximation scheme of sparse grids can be constructed as a subspace of the tensor-products of one-dimensional spaces represented by a hierarchical multi-resolution scheme, such as the hierarchical basis see Figure 1.1 and the historical reference [Fab09], or generally any basis system of pre-wavelets or wavelet [Har95]. Each one-dimensional basis function can be derived from a model function ϕ by a scaling of 2^{-l}

(also called level l) and a translation by a multiple of 2^{-l}. In the case of the hierarchical basis, the model function ϕ is a hat function.

Let us consider piecewise linear functions. The space of functions on the interval $[0, 1]$ of dyadic level l is based on a mesh with mesh parameter $h = 2^{-l-1}$. There are $2^{l+1} + 1$ nodes. In nodal basis, there are $2^{l+1} - 1$ interior hat functions of support $2h$ and two boundary functions of support h, each associated with one node (or grid point). In the hierarchical basis, there are two boundary functions (T_0) and one global function (T_1) of support 1, see figure 1.1, two functions of support $1/2$ (T_2), four functions of support $1/4$ (T_3) and so on. All hierarchical basis functions of support $2h$ and larger span the function space of level l.

Based on the one-dimensional case, now we construct spaces piecewise linear interpolants on a d-dimensional unit hyper-cube. The space of functions on the regular grid of dyadic level l has a mesh parameter $h = 2^{-l-1}$. It can be represented by the space of all tensor-products (also referred to as direct products) of one-dimensional nodal or hierarchical basis functions of level l, i.e. support larger than 2^{-l-1}. On level l, the two-dimensional space e.g. can be written as

$$T_{ll} = \langle T_i \otimes T_j \rangle_{i,j \leq l} \,.$$

In contrast to the regular grid, the corresponding sparse grid space consists of products of hierarchical basis functions with support larger than a d-dimensional volume of size 2^{-l-1}, see Figure 1.2. On level l, the two-dimensional space e.g. can be written as

$$T_{ll} = \langle T_i \otimes T_j \rangle_{i+j \leq l} \,.$$

This is a subset of the regular grid space. A regular grid has about $2^{d \cdot l}$ nodes, which is substantially more than the $2^l \cdot l^{d-1}$ nodes of the sparse grid.

The major advantage of sparse grids compared to regular grids is their smaller number of nodes (or grid points) for the same level l and resolution 2^{-l}, which means a smaller number of basis functions (shape functions in FEM) and therefore a smaller dimension of the function space and fewer degrees of freedom in a discretization. This is especially true in higher dimensions $d \gg 1$.

However, the question whether sparse grids have an advantage compared to regular grids does also depend on the discretization accuracy of a solution obtained on a grid. We are interested in a comparison of accuracy versus number of nodes for both types of grids, a regular and a sparse one. We define the storage ε-complexity of an approximation method by the accuracy ε, which can be achieved with a storage of N nodes. The accuracy depends on the smallest mesh parameter h and an approximation order p like $\varepsilon = \mathcal{O}(h^p)$ for smooth data. For regular grids the number of nodes depends on the space dimension d as $N_{storage} = h^{-d}$, which results in $\varepsilon =$

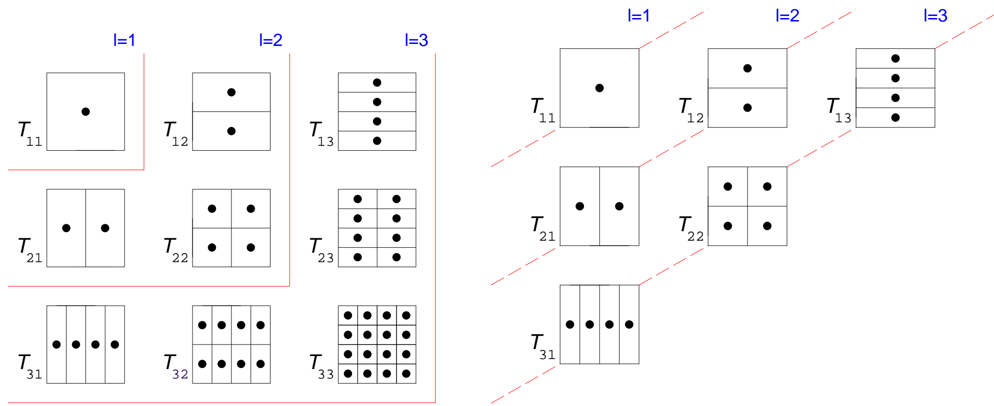

FIGURE 1.2. Tableau of supports of the hierarchical basis functions spanning a two-dimensional regular space (left) and the corresponding sparse grid space. The space T_{ij} is defined by the tensor product $T_i \otimes T_j$.

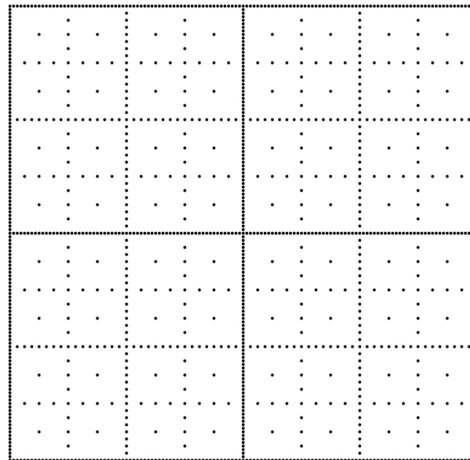

FIGURE 1.3. A sparse grid on the unit square. Depicted are the nodes associated to the hierarchic basis functions.

	$d=1$	$d=2$	$d=3$	$d=1$	$d=2$	$d=3$
first order	$\mathbf{N^{-1}}$	$N^{-1/2}$	$N^{-1/3}$	$\mathbf{N^{-1}}$	$\mathbf{N^{-1+\gamma}}$	$\mathbf{N^{-1+\gamma}}$
second order	N^{-2}	$\mathbf{N^{-1}}$	$N^{-2/3}$	N^{-2}	$N^{-2+\gamma}$	$N^{-2+\gamma}$
third order	N^{-3}	$N^{-3/2}$	$\mathbf{N^{-1}}$	N^{-3}	$N^{-3+\gamma}$	$N^{-3+\gamma}$

TABLE 1.1. Storage-complexity of a regular (left) and a sparse grid discretization. Terms printed in bold face indicate linear ε-complexity, terms below indicate better complexity.

$\mathcal{O}(N^{-p/d})$. In the case of sparse grids, the dependence on the dimension d is much weaker and we denote $\varepsilon = \mathcal{O}(N^{-p+\gamma})$ for every $\gamma > 0$ and an approximation order p. The sparse grid approximation is said to break the curse of dimensionality.

Let us assume that a sparse grid approximation is of first order $p = 1$, which of course depends on the discretization order, the error norm, the smoothness of the solution and the sparse grid approximation itself. Then the sparse grid is competitive to a second-order method in two dimensions and to a third-order method in three dimensions, which is usually much more expensive and harder to construct, see Table 1.1.

Furthermore the number-of-operations complexity is of interest, because it is an estimate for the computing time a specific algorithm needs. In some cases the work count is proportional to the number of nodes. This is true for a single time-step of a standard explicit finite difference code. We will see that this is also true for the corresponding sparse grid code. However, the work count usually is higher than the number of nodes for implicit discretizations of time-dependent problems and for stationary problems involving the solution of (non-) linear equation systems, and for time-dependent problems in total. The number of time-steps for an evolution problem of a fixed time interval depends on the spatial resolution h e.g. due to a CFL stability condition, which leads to a higher work count complexity in N. On regular grids we obtain $N_{work} = \mathcal{O}(h^{-d-1})$, which is equivalent to the storage complexity in $d+1$ space dimensions. This means that any reduction in storage $N_{storage}$, e.g. through sparse grids, may reduce the number of operations even further.

We still have to check the assumption on the approximation order $p = 1$ (or some other constant) of a sparse grid discretization. Up to now such orders had been verified for the extrapolation method, for the interpolation error, for the energy error of the Galerkin method, and for the consistency of the finite difference method. However, either quite strong regularity assumptions or model problems were considered. Furthermore, numerical experiments indicate that in more general cases and in connection with adaptive sparse grids the approximation orders obtained for model situations hold.

1.3 Discretization of a PDE

There are several, completely different ways to discretize a partial differential equation. Many methods provide grid-based discretization schemes and can be generalized to sparse grids. Very popular are the Finite Element Method (FEM) based on the Galerkin approach, Finite Differences, the Finite Volume Method and spectral methods. Sparse grid discretizations have been so far constructed by the Galerkin method [Zen91, BDZ96], finite differences [Gri98, Sch98], by extrapolation based on finite differences [GSZ92] and finite volumes [Hem95], and by a spectral method [Kup97] based on a fast Fourier transform defined on sparse grids.

1.3.1 The extrapolation method

Probably the simplest way to solve a PDE on a sparse grid is to use the extrapolation method, also called the 'combination'- [GSZ92] or 'splitting extrapolation'-technique [LLS95]. The idea is to combine solutions computed on several different regular grids to a more accurate sparse grid solution. The approach uses the extrapolation idea of numerical analysis to cancel out some low order error terms by the combination of solutions and to achieve a smaller discretization error.

Let us assume that we have a standard PDE solver for regular grids of $n_1 \cdot n_2 \cdots n_d$ nodes. This can be any software capable of uniform grids on rectangular shaped domains or anisotropic grids on the unit square respectively. Let us denote such an anisotropic grid as $G_{h_1,h_2,\ldots,h_d}$. Furthermore the mesh parameters h_i will always be of the form $h_i = 2^{-j_i}$ with a multi-index j. A sparse grid of level l can be decomposed into the sum of several regular grids

$$G_l^{\text{sparse}} = \bigcup_{|j|=l} G_j\,,$$

see also figure 1.4. The idea is now to use this decomposition and the numerical solution of the PDE and to decompose the solution u into solutions u_j on regular grids. This can be done as

$$u_l^{\text{sparse}} := \sum_{|j|=l} u_j - \sum_{|j|=l-1} u_j\,, \tag{1.1}$$

which is an extrapolation formula with weights $+1$ and -1. If we depict the grid points of a grid G_j, each node is associated with a shape function in a FEM discretization. The solution originally obtained only in the nodes is extended by the shape functions of a linear FEM discretization to a piecewise linear function on the whole domain. Now we can sum up functions from several grids e.g. $G_{2,0}$, $G_{1,1}$ and $G_{0,2}$. The union of the nodes gives the nodes of the corresponding sparse grid, compare Figures 1.3 and 1.4. Computationally, the combination of the grids requires interpolation

on each grid, or the evaluation of the shape functions on the sparse grid points, because the sparse grid contains always a superset of nodes of each of the regular grids. The remaining nodes are determined by interpolation. For the correct extrapolation we also need to subtract several coarse grids, which can be done in the same way.

FIGURE 1.4. The sparse grid combination technique. Several solutions of the PDE obtained on different regular grids by standard PDE codes are summed up to form a sparse grid solution of higher accuracy. The weights of +1 and −1 are depicted next to the grids. The grids are arranged analogous to the tableau in Figure 1.2. The union of nodes of all regular grids results in the nodes of the corresponding sparse grid, see Figure 1.3.

The fact that the extrapolation solution equals the solution of the PDE on the sparse grid can be proven by an error expansion of the regular grid solutions and the cancelation of the lower order error terms, see [GSZ92]. The quality of an extrapolation formula, i.e. the sum of the remaining error terms, depends on higher derivatives of the solution u and can be measured in higher order Sobolev norms (e.g. $H^{2d}(\Omega)$). Given sufficient smoothness, it is possible to combine solutions obtained by Finite Elements, Finite Volumes or Finite Differences. The latter two require an appropriate, order-preserving interpolation scheme for the nodal values obtained on single grids. The FEM provides a natural extension of the solution from the nodes to the whole domain through the shape functions used. Error bounds in different norm are available and the error usually compares to the regular grid error, with a logarithmic deterioration $|\log h|$.

Besides its simplicity, there are several advantages with this method: The solution can be computed concurrently, that is on different processors of a parallel computer (almost embarrassingly parallel, see [Gri92]) or one after the other requiring only little computer memory. Multiple instances of a standard PDE code can be used and only little coding is necessary to implement the method. Furthermore such a code may be a very efficient

code optimized and tuned for structured grids on a specific computer. The prerequisites of such a code are an interpolation procedure, which defines the solution in the whole domain, and a standard solver, which is able to deal with anisotropic discretizations i.e. a grid of $n_1 \cdot n_2 \cdots n_d$ nodes with different n_i. Direct solvers and several types of iterative solvers can be used here, for example the semi-coarsening version of a multigrid solver.

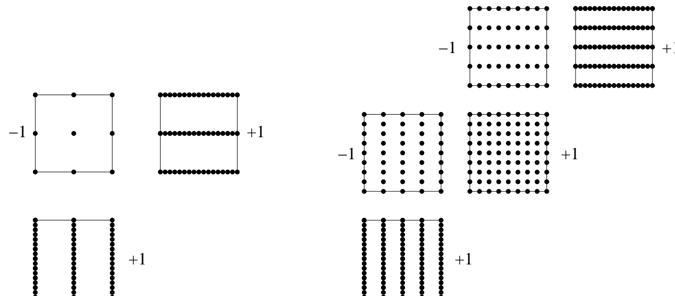

FIGURE 1.5. Sparse grid extrapolation schemes: splitting extrapolation (left), 'semi'-sparse grid (right). A smaller number of solutions than for the sparse grid extrapolation technique are combined to form a higher accuracy solution. Again, each solution on a regular grid can be obtained by any suitable software package or PDE solver.

Here is where the first problems occur. Think of a fluid flow problem or some convection problem discretized on an anisotropic grid with $3 \times 2^{l-1} + 1$ nodes. A discretization may have problems with stability, or at least the numerical linear algebra is difficult. The extrapolation does work as long as there is a suitable error-expansion, which depends on the problem and the discretization. If the solution of extremely anisotropic refined grid is useless for some larger convection terms, the extrapolation does not work properly. A possible fix is to restrict the aspect ratios of the grids under consideration and to remove these from the extrapolation. This can be done by some additional restriction to the index j

$$u_l^{\text{sparse}} := \sum_{|j|=l,\ j_i \leq \gamma l} u_j - \sum_{|j|=l-1,\ \leq \gamma l} u_j$$

with some parameter $\gamma < 1$. For three-dimensional fluid flow problems this is heavily used by Hemker [Hem95], see figure 1.5. Further modifications of the extrapolation method consists in a piecewise constant interpolation for some finite volume discretizations [Hem95] and in the combination of few grids [LLS95]:

$$u_l^{\text{sparse}} := \sum_{k=1}^{d} u_{j+(l-|j|)e_k} - u_j \,.$$

The idea is to refine the grid in each coordinate direction only once. Of coarse this gives a slightly different solution than the originally proposed method, see figure 1.5, but we will see the relation in chapter 1.4.1 on adaptive grids.

We have mentioned the stability problem for anisotropic discretizations. But there are more possible pitfalls: The solution of time-dependent problems and of non-linear problems can be done in several ways. The cheapest way would be to extrapolate the solution of the last time-step or the final non-linear iterate. Under some circumstances this is a legal approach. However, in the presence of phenomena not resolved on some grids, the extrapolation might cause a disaster. The error expansion of the discretization scheme is not accurate enough on such coarse scales and the final result is polluted by higher-order terms. A more expensive version of the extrapolation methods is to combine the solution at every time-step or every non-linear iteration steps. This requires more CPU-time, some more changes in the code and does not parallelize so well. However, it definitely more accurate. Further difficulties may arise with the resolution of coefficients, boundary conditions and the source terms on every grid, with non-linearities, and transport terms and singularities.

Adaptive grids are more difficult to incorporate into the extrapolation scheme, because the grid refinement of the single grids has to match. An adaptive refinement procedure has to use a complicated collaborative grid refinement, which is why this has not yet been implemented.

1.3.2 The Galerkin method

The standard approach to discretize an elliptic differential equation on a sparse grid or even on an adapted sparse grid is the Galerkin method (FEM). Given a sparse grid along with the functions on that grid, it is straightforward to apply a Galerkin scheme with test and trial functions ϕ_i. These functions are defined by the hierarchical basis and some other multi-scale resolution, each one related to a node. The equation system is derived from the variational form $a(.,.)$ of the differential equation:

$$\begin{aligned} a(u,v) &= f(v) \qquad \forall v \in H \\ a_{i,j} &= a(\phi_i, \phi_j) \\ f_j &= f(\phi_j) \\ \sum_i a_{i,j} u_i &= f_j \qquad \forall j \end{aligned} \tag{1.2}$$

This method can be applied to any set of linear independent shape functions in a finite-dimensional setting and furthermore for a complete basis in the case of infinite-dimensional function spaces like H^1. The idea now is to

use shape functions of the sparse grid, which are direct products of one-dimensional functions of the hierarchical basis or some other (pre-) wavelet basis. At a coordinate x in space

$$\phi_i(x) = \prod_{j=0}^{d} \phi_{i_j}(x_j)$$

hold. In the case of the hierarchical basis, the functions ϕ_i are the standard multi-dimensional hat functions also used in FEM. This discretization is symmetric as long as the bilinear form $a(.,.)$ is self-adjoint. The discretization error can be estimated by the interpolation error of the sparse grid. Error bounds for the energy norm of the Laplacian are available and compare to the regular grid error, with a logarithmic deterioration $|\log h|$. However, sharp L_2 norm estimates are not known yet for the Galerkin solution, while pure best approximation results in L_2 norm do not give the logarithmic deterioration $|\log h|$, see [Osw98]. However, computationally some logarithmic deterioration is observed also in L_2 norm, see [Bun92].

There are two main drawbacks with a naive implementation of this approach: The stiffness-matrix $a_{i,j}$ is not sparse, unlike the FEM case, and the computation of f_j is quite expensive. As a consequence, the performance of the sparse grid degrades to the performance of a full h^{-d} grid. This is basically due to the fact that a lot of shape functions do have a large support. For the FEM discretization, these supports are small and only a limited number of shape functions might interact in the computation of $a(\phi_i, \phi_j)$.

The fix is not to assemble the stiffness matrix, but to use an algorithm for a matrix multiply or Gauss-Seidel step on the fly. This algorithm can be formulated in terms of tree traversals and has indeed linear complexity [Zen91]. This means that if the number of matrix multiplies and a Gauss-Seidel iterations is limited, for example by some suitable preconditioner or accelerator, the equation system can be solved in optimal complexity. Extensions of this algorithm to several dimensions and to certain types of variable coefficients have been developed, see [BDZ96] and references therein. The main difficulty with this respect is the symmetry of the discretization. The optimal order algorithms are usually based on the assumptions of constant coefficients. The treatment of the variable coefficients case is complicated. In order to maintain optimal complexity, the coefficient function is approximated on the sparse grid. This approximation can cause asymmetries in the operator for a naive implementation. Further difficulties come from jumping coefficients not aligned to a grid axis and from more complicated differential operators.

1.3.3 The finite difference method

A different way of a sparse grid discretization is a finite difference scheme [Gri98, Sch98], which we will employ further on in this paper. It is simpler

to implement and to apply to different types of equations, but there is not that much known analytically.

We define the hierarchical transformation $\mathbf{H}$ as the hierarchical basis transformation on the regular grid from nodal values to hierarchical values, which are restricted to the sparse grid nodes. Both, the nodal basis and the hierarchical basis span the space of piecewise linear functions. Hence any function of the space is uniquely represented as linear combination of basis function, and such a representation can be uniquely converted to the representation of another basis. However, computationally such a transformation has to be fast in order for the transformation to be useful. All wavelet-type of basis functions provide such fast $\mathcal{O}(n)$ transformation to and from the nodal basis representation. The transformation is especially simple for the one-dimensional hierarchical basis: Given the nodal values u_j with $j = 0, 1, \ldots, 2^{l+1}$, the hierarchical representation for interior points can be obtained by

$$\hat{u}_j \;=\; u_j - \frac{1}{2}(u_{\textbf{left father}} + u_{\textbf{right father}}) \tag{1.3}$$

and the boundary nodes u_0 and $u_{2^{l+1}}$ remain unchanged. The nodal values are replace by their hierarchical excess or deterioration, compared to the value obtained by interpolation between on the next coarser level grid. The inverse transformation can be implemented similarly.

$$u_j \;=\; \hat{u}_j + \frac{1}{2}(u_{\textbf{left father}} + u_{\textbf{right father}}) \tag{1.4}$$

However, the coarse nodes have to be computed before the finer grid nodes. Furthermore, the transformation can be implemented in place, without an auxiliary vector. The hierarchical basis transformation $\mathbf{H}$ is also abbreviated by the stencil [1/211/2].

Based on the hierarchical basis transformation $\mathbf{H}$, we define the action of a one-dimensional finite difference operator for the discretization of a differential operator, see Figure 1.6: We apply the associated standard difference stencil $\mathbf{D}_i$ along the x_i-axis to values located on the sparse grid nodes in a specific basis representation. To this end the values are given in nodal basis in direction i and in hierarchical basis representation in all other directions $I \setminus \{i\}$. The associated transformation is denoted by $\mathbf{H}_{I\setminus\{i\}}$. The stencil $\mathbf{D}_i$ for each node itself is chosen as the narrowest finite difference stencil available on the sparse grid. It is equivalent to the corresponding stencil on a regular, anisotropic refined grid. The finite difference stencil can by a 3-point Laplacian $\frac{1}{h^2}[1 \ -2 \ 1]$, an upwind-stabilized convection term $\frac{\partial}{\partial x_i}$, some variable coefficient operators and so on. In nodal values the finite difference operator reads

$$\frac{\partial}{\partial x_i} u \;\approx\; \mathbf{H}^{-1}_{I\setminus\{i\}} \circ \mathbf{D}_i \circ \mathbf{H}_{I\setminus\{i\}}\, u\,. \tag{1.5}$$

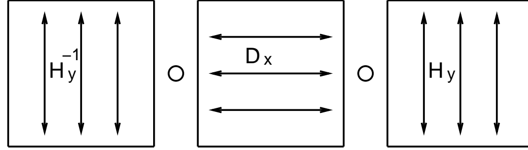

FIGURE 1.6. Scheme for a finite difference operator in x-direction.

A general difference operator is then obtained by dimensional splitting. A linear convection-diffusion equation, as a simple example, can be discretized in nodal basis representation as usual as

$$\nabla \cdot u \approx \sum_{i=1}^{d} \mathbf{H}_{I \setminus \{i\}}^{-1} \circ \mathbf{D}_i \circ \mathbf{H}_{I \setminus \{i\}} u , \tag{1.6}$$

where the one-dimensional difference operators $\mathbf{D}_i$ may be chosen as a two-point upwind stencil $c \cdot \frac{1}{x_i - x_{i-1}} \cdot [-1 \; 1 \; 0]$ (convection term) or plus the three point centered Laplacian $a \cdot \frac{1}{(x_{i+1} - x_{i-1})^2}[1 \; -2 \; 1]$ (diffusion term). On adaptively refined grids, the nearest neighbor nodes are chosen, which may lead to asymmetric stencils, i.e. non-uniform one-dimensional stencils. Further higher order modifications of the stencils have been tested, too. In the presence of a transport term in the equation, the unsymmetry is believed to be no problem. There are many ways to create discretizations of all kind of equations, e.g. for the Navier-Stokes equations [Sch98] or some hyperbolic conservation laws [GZ98a].

There is not that much of theory known for the finite difference discretizations. However, the consistency error has been analyzed for some model problems. It behaves like the consistency of regular grids. The second ingredient of a convergence analysis about the stability is still missing, but numerical experiments indicate that the stability deteriorates by a logarithmic factor, which results in similar convergence results as the Galerkin method on sparse grids.

The algorithmic part of the finite difference discretization is much simpler than for the Galerkin method if one accepts the fact that the discretization is unsymmetric. As usual only the matrix multiply is implemented and the matrix is not assembled for complexity reasons. The sparsity pattern, see Figure 1.7 shows that the matrix is not really sparse and a matrix assembly would deteriorate the overall complexity. Analytically it is known that the average number of non-zero entries per row in the stiffness matrix with $N \approx 2^l \cdot l^{d-1}$ unknowns is roughly $2^l/l$ for the Galerkin method, i.e. the matrix is almost dense, see [Bal94]. The finite difference discretization results in a lower average of l^{d-1} entries, which is still too much to be stored in a matrix. Due to to unsymmetric finite difference discretization, any iterative solver for the equation solvers has to deal with the unsymmetry

and methods like BiCGstab and GMRES are quite popular.

Basically three steps have to be implemented:

- transform a vector of nodal values to hierarchical basis: $\mathbf{H}_{I\setminus\{i\}}$
- apply a one-dimensional finite difference stencil along the coordinate axis x_i: $\mathbf{D}_i$
- transform the vector back to nodal basis $\mathbf{H}_{I\setminus\{i\}}^{-1}$

The basis transformation can be implemented as a sequence of one-dimensional transformations $\mathbf{H}_i$ and $\mathbf{H}_i^{-1}$, each along one coordinate axis x_i. Furthermore, we have to be able to sum up the resulting vectors. A complete PDE solver furthermore includes at least one iterative solver, some output or post-processing features and a treatment of different boundary conditions.

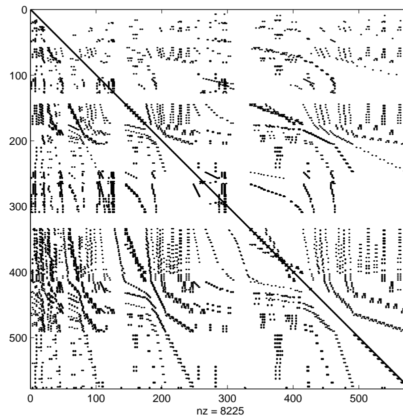

FIGURE 1.7. Sparsity pattern of a 2D finite difference discretization.

1.4 Adaptivity

The quality of the approximation of a function on sparse grids depends on the smoothness on the function. The smoother the function is, i.e. the smaller several higher order mixed derivatives are, the better is the sparse grid representation. Often functions are of interest, which violate this smoothness condition. The solution of a PDE may develop singularities at the boundary of the domain. Also some steep gradients and boundary layers are not efficiently represented on a sparse grid. In order to use the nice multi-dimensional properties of sparse grids also in conjunction with such functions and solutions, adaptive grid refinement can be employed.

The grid is not longer a regular one, but it is specially adapted to the function, which has to be represented. This can be done in several ways. However, the general approach is to insert additional nodes or to remove present ones to adapt the grid. Both approaches, adding nodes or removing nodes, can be used to define the term of an 'adaptive sparse grid'.

However, such general definitions of sparse grids include all types of regular grids their adaptively refined versions. A regular grid of level l can be obtained from a sparse grid, either by inserting nodes into a sparse grid of level l, or by removing nodes from a sparse grid of level dl, which contains the regular grid already. A more precise definition of adaptivity can be obtained if one describes the adaptive refinement algorithm instead of the grid.

1.4.1 A priori adapted spaces

An a priori way of grid adaptivity is to optimize sparse grids towards special classes of functions. Classes of interest are space with special smoothness properties. Either a general higher-order smoothness is required, or a smoothness appropriate to the tensor product structure of the grid. This can be done by spaces like H^{2d}, where special smoothness restrictions are imposed on mixed derivatives of the function. Further optimization can be geared towards different error norms. Form a best approximation point of view, the sparse grids are optimal in an L_2 sense. However, another norm like the energy norm or equivalently the H^1 norm gives rise to other grids of sparse grid constructed in [Bun92]. Here emphasis is put on strong anisotropic grids, while some of the isotropic grids can be removed. This results in an overall number of nodes of h^{-1} and breaks the curse of dimension completely. Any other positive and negative Sobolev norm leads to a different sparse grid, which can be constructed systematically, see [GK98]. An extreme case is the splitting extrapolation grid of [LLS95].

1.4.2 Adaptive grid refinement

A completely different type of adaptivity is the a posteriori approach: The sparse grid is refined locally in space, to resolve jumps, singularities and related features of a specific solution. The grid refinement is done during computation. The general approach to adaptive grid refinement consists of an error indicator or estimator, a selection procedure and a geometric grid refinement procedure. Along with a solution method for the equation system, this forms the ingredients of an iterative refinement cycle. This works very well for other types of grid such as unstructured grids based on triangles or tetrahedra, based on composite grids as an overlapping system of regular grids, and for more complicated unstructured grids with mixed element types. The theoretical foundation of error estimators is well understood in the elliptic case, given some regularity of the solution, see

[Ver96]. The grid refinement for a FEM scheme is also sound in the elliptic case.

However, for global approximation schemes like sparse grids, or transport equations, where a small error in one nodal point affects large parts of the domain and many other nodes, this method has to be modified. In the finite element case, an approximation is local and a change in one node does only affect its neighbor nodes immediately. A sparse grid discretization uses hierarchical basis transformations, which link whole lines of nodes. Hence, many more nodes can be affected. The discretization scheme is said to be non-local.

Let us take a closer look, why the adaptive refinement cycle usually does work: Regular parts of the solution, say H^2 do not need adaptive refinement. Additional refinement, which has been done accidently or due to refinement nearby, does not hurt, on the other hand. So we have to take care of the rough parts of the solution, in the elliptic case singularities of $H^{1+\alpha}$ type. The precise knowledge of the one-dimensional case for x^α singularities along with their approximation by piecewise polynomials gives rules for the optimal grading of a mesh. An optimal grid in d-dimensions can be derived approximately from the 1D case. A point singularity requires a graded radial to the point, while tangentially the grid can be arbitrary. Line singularities require a mesh grading radial to the line, and so on. An adaptive grid refinement procedure will try to mimic this grid grading. This can be achieved by element bisection schemes, which do not refine steep enough compared to an optimal grid, but sufficiently for the efficient resolution of the singularity. An adaptive procedure, which bisects elements with large errors for example, can produce such sub-optimal grids. It is enough to refine all elements next to a node with a large error, because the elements contribute to the error, and a refinement decreases the local error. Accidental further refinement does not spoil the performance, loosely speaking, as long as the number of unnecessary nodes does not exceed the required nodes.

Let us return to sparse grids. They are a non-local approximation scheme. Given a node of the grid with a large error, there are a lot of elements and other neighbor nodes which contribute to the error. Their number is proportional to the grid level and may be large. This is in contrast for the FEM, where the number of neighbors is limited. Hence local refinement is quite expensive. Furthermore, any change in one node does affect large parts of the domain. It is not longer possible to localize the error and to approximate the elliptic differential operator by its diagonal part (i.e. the diagonal entries of the stiffness matrix). As a consequence, we cannot expect that the (expensive) local grid refinement does actually reduce the error of the node under consideration.

One way to improve the situation is to look at error estimators constructed for other global phenomena like transport equations. Here influence factors are computed instead of local errors, which describe the in-

fluence of a node onto the (global) error, see [BR95]. Instead of the local error, a number which indicates the influence of the respective node onto the error is used for refinement. Such indicators can be obtained from the solution of a dual problem based on the current solution of the original problem. An adaptive refinement procedure will refine the grid in the vicinity of nodes with a large influence onto the global error. This coincides with nodes of large errors in the elliptic case with a FEM discretization. However, for transport problems the location where the error occurs and where the error originates differs, because the error is transported along with the solution. Hence it is essential not to refine where the error is measured, but to remove the source of the error.

A similar procedure can be used for global approximation schemes of sparse grid type: Given some global error criterion such as the H^1 or L_2 norm of the error, it is possible to compute the influence of each node onto the global error [Bun98]. For efficiency reasons, the dual problem may be solved on a coarser scale than the original problem. However, the problem remains how to refine the sparse grid after selecting a set of nodes with a large error contribution. Due to the global support of many of the shape functions, it is not clear which nodes in the vicinity of the selected nodes should be created. It is even unclear, what a vicinity in the presence of global functions means. Hence we have to introduce a further extension of the refinement procedure. The adaptive grid refinement procedure does work for FEM, because the differential operator is spectrally similar to its diagonal part. However, we are able to transform the equation system to a basis where the sparse-grid operator is almost a diagonal operator: The appropriate wavelet basis, where the condition number of the operator is a constant independent from the mesh parameter h allows one to neglect the global coupling of the operator. Hence the coefficients of the estimate error, represented in this wavelet basis, lead to a better and provable refinement criterion [Osw98]. The question of geometric grid refinement is therefore partially solved. However, the algorithmically side is still open.

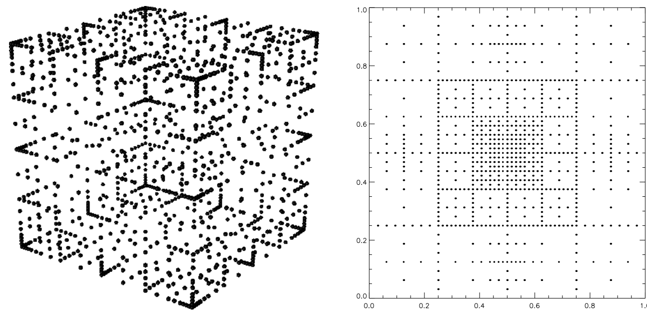

FIGURE 1.8. Adaptive sparse grids. ‘Standard’ refinement (left) and a ‘shield’ grid (right).

Another approach to adaptive sparse grids are the shield grids [HBZ95]. The idea is to 'shield' the singularity by several boxes of nodes in order to avoid the pollution outside the box. Standard sparse grids can be used outside. The grid is constructed as the union of nodes of a standard sparse grid and the boxes. Each quadratic box is aligned to the grid and contains all nodes on the outline of the square. The nodes are located with a distance $h_{\min}$. The effect is that all nodes, whose shape functions support contains the singularity, are shielded, i.e. their next neighbors in all coordinate directions are very close, $h_{\min}$ away.

Numerical experiments indicate the effectiveness of this approach for certain point singularities caused by a singular source term. There are also adaptive refinement procedures to construct such grids. However, the complexity of the grid, that is the number of grid points, is higher than standard sparse grids: The largest box has a side length of 1/2 and a surface of $2d\frac{1}{2}^{d-1}$ (in several dimensions). This means that the nodes required to cover the surface of the box with a mesh of nodes at a distance of h is $h^{-d+1}\log h$ compared to $h\log^{d-1} h$ nodes of a regular sparse grid. As a consequence the curse of dimension is present in this discretization and one of the advantages of sparse grids for higher-dimensional problems is lost. Furthermore, it does cause harm to add further nodes to a shield grid accidently near the singularity, because a point-type singularity might not be shielded by the box any longer. A conceptual difficulty lies in the global parameter $h_{\min}$ of the boxes for a specific grid, because grid refinement is often considered a local procedure. Think of a problem with several areas of refinement, where the grid needs different strength of mesh grading. A global parameter would be adjusted to the strongest refinement and would therefore spend too much effort on weaker refinement regions.

1.5 Software abstractions

The goal for the development of our finite difference sparse grid code is to be able to test and to verify different types of discretizations on sparse grids and to tackle different types of partial differential equations. Hence a flexible and modular design is a must.

Techniques such as abstract data types and object oriented programming provide such a flexibility, see [BN94, ABL97]. However, they can easily lead to slow and inefficient code due to an over-use of design features. For example, overloading the arithmetic operators '+' and '*' for vectors in C++ sounds attractive. However, it is usually less efficient than providing directly a *saxpy* operation for expressions of $\alpha\vec{v}+\vec{w}$ type with multiple arguments. This is due to memory management for intermediate results and due to a lack of compiler optimization.

Other approaches to implement numerical algorithms focused on data

structures such as trees for the representation of a grid. They incorporated the numerical algorithms into the algorithms which manipulate the data structures, e.g.arithmetic operations are done during a tree traversal, i.e. one addition at each leave of the tree for a global *saxpy* operation [Zen91]. This is of course much slower than an ordinary *saxpy* operation on vectors. An 'object-oriented' programming style now leads to a separation of the tree traversal (as an iterator) and the arithmetic operation, resulting in additional overhead, see [Rüd92]. Hence, splitting a large code into many small functions and loops may lead to inefficiencies, which cannot be resolved by an optimizing compiler. A very careful code design is needed.

We have based our code on several fundamental higher-level abstractions, which are well separated both in functionality and in implementation. Each implementation of an abstraction itself is coded in a more classical style. This guarantees that we do not loose efficiency in a substantial way due to this separation. We have identified the following building blocks, see figure 1.9 (left). They are ordered from low-level, computationally expensive and efficient, to high-level routines, where efficiency is achieved through call of some efficient subroutines. Similar abstractions can be found in other object oriented software packages for partial differential equations such as Diffpack, see [BL97].

- *vector*: a large container for real numbers, including *Blas* level 1 arithmetic
- *grid*: geometric description of an adaptively refined sparse grid. It provides I/O, refinement and addressing, and even standard grids
- *field*: a (solution) function on a sparse grid. It provides a mapping between a grid and a vector and gives and an interpretation of the vector data as (collocated) scalar or vector field
- *operator*: the finite difference operators that are defined on a sparse grid and operate on vectors, see figure 1.9 (right)
- *solver*: different iterative (Krylov) solvers, uses a differential operator

1.5.1 Numerical data: vectors

Large amounts of data are stored efficiently in vectors. We avoid its storage inside some other data structures such as trees or hash maps which will be explained in chapter 1.5.2. The declaration of a vector class, based on a primitive C vector, which allows also for an efficient interface to Fortran subroutines, see [BN94], reads as follows

```
class Vec {
protected:
  real *r;
  int n;
```

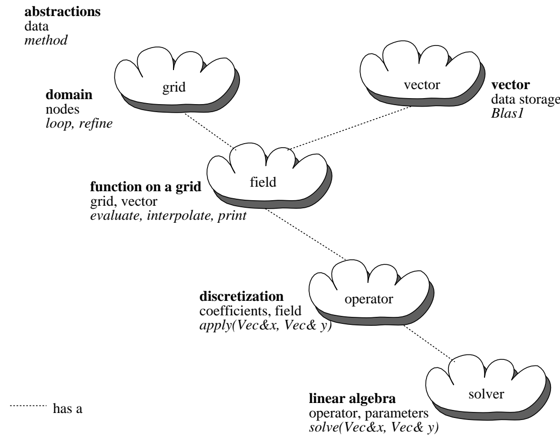

FIGURE 1.9. Abstractions used in the sparse grid code.

```
...
};
```

The member functions implement some management and the functionality of Blas level 1 operations such as the following vector plus vector operation. These member functions can either be optimized by the compiler or used to call an optimized library.

```
void Vec:: add(const Vec& a, const Vec& b)
{
  assert((a.n==n)&&(b.n==n));
  for (int i=0; i<n; i++)
    r[i] = a.r[i] + b.r[i];
}
```

Additionally a short vector *SVec* is defined. It provides a memory efficient way to handle e.g. coordinates. If the code is optimized for three-dimensional applications for example, the underlying data structure is *real[3]*. There are some administration member functions.

1.5.2 The sparse grid

One of the central abstractions is the grid, see figure 1.10. Along with standard grids and adaptive girds with hanging nodes [GZ98b], there are sparse grids available. Grids represent the shape of the domain Ω along with the set of nodes, but without any numerical data such as a discretization or a function. The sparse grid is represented by a description of the (hyper-) cube shaped domain Ω and a set of nodes in relative coordinates Ω. The methods provided for a sparse grid are different types of grid refinement, where an indication what to refine is needed, and building blocks for loops

over all nodes. Furthermore there are functions to convert different representations of nodes and its coordinates.

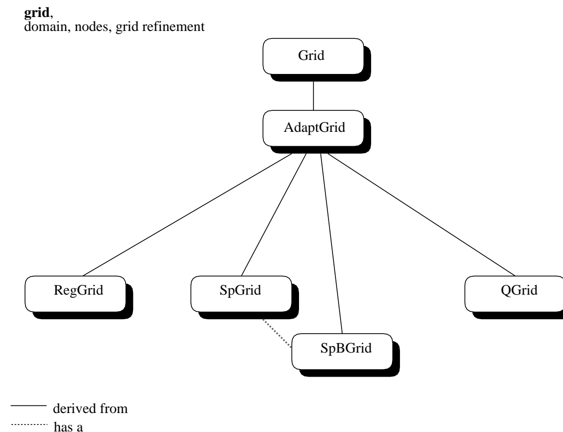

FIGURE 1.10. The abstractions for the geometric properties of a grid.

The approach to store adaptive sparse grids has been so far tree data structures as described in [Zen91]. Each node contains one pointer to its parent and two pointers to its children in each direction. This is in total an amount of $3d$ pointers per node. Using the pointers, different types of tree traversal can be implemented and many numerical algorithms developed for sparse grids are based on these algorithms. The large number of pointers requires a lot of memory and a lot of operations to administer them. For large scale problems however, one is especially interested in a low operation count and low memory consumption implementation. Of course there are modifications of the tree data structure to reduce these requirements slightly, see [BDZ96]. Imagine that an algorithm accesses all allocated memory in some random order, the algorithm will be faster if one is able to reduce the amount of allocated memory. Furthermore, we are interested in the parallelization of such a code. Each pointer poses a potential problem, because the administration of a pointer pointing to data on a distant processor is troublesome.

So there are two goals for us to chose a different data structure: We want to get rid of pointers and we want to separate data storage from the algorithm. The first goal is accomplished by a key-based addressing and for the second one we choose a hash table, but there are of course alternatives, see [Sam90].

Key-based addressing substitutes the memory address stored in a pointer with an integer value uniquely describing the entity. In our case, each node can be characterized by its position in space, that is the local coordinates. Let us begin with a one-dimensional scheme. We number the nodes of a

```
0                                   -1
0                1                  -1
0       2        1        3         -1
0   4   2   5    1   6    3    7    -1
0 8 4 9 2 10 5 11 1 12 6 13 3 14 7 15 -1
```

FIGURE 1.11. Key values of the nodes in five levels of a one-dimensional grid

regular grid level by level. At each level the additional nodes are given a number, while the other numbers stay the same. Depicted in Figure 1.11 are five levels of a one-dimensional grid. Any grid can be represented by the set of its nodes which are stored as keys. The key of a node and its coordinates can be uniquely converted into another. The nodes which are actually present in a grid will later be numbered in the sparse grid. This numbering can be very crude if the grid was created by adaptive refinement and the nodes were numbered in the order of their creation.

```
0                                                                  -1
0                               1                                  -1
0               10              1               11                 -1
0      100      10      101     1      110      11      111        -1
0 1000 100 1001 10 1010 101 1011 1 1100 110 1101 11 1110 111 1111 -1
```

FIGURE 1.12. Key values of the nodes in binary representation

Looking at the binary representation of the keys in figure 1.12, we see some more structure. Hence the level of a node can be computed by its binary logarithm, the relative coordinate is counted from left to right. Transformations from coordinate to key and backwards are available. In the multi-dimensional case, a coordinate vector can be transformed into a vector of numbers. The enumeration scheme, along with many others, can be used to name nodes uniquely.

Furthermore we need some storage mechanism, where nodes can be accessed by their respective keys. We choose a hash map, where the key is mapped to an address by a hash function. Hashing is a very general storage concept in computer science, see [Knu75], often used in data base system, but also in compilers or e.g. the Unix shell. It is used to store and retrieve large amounts of data without relying on any special structure or distribution of the data. A very general universe of all possible key values is mapped to a finite number of cells in an ordinary vector in memory by a hash function. The key and its related data is stored in that vector cell, given that no other key occupies it, which is called a collision. In that case some collision resolution mechanism assigns another storage location to the key. The hash implementation of SGI's version of the STL [Aus99] we use, is based on a collision resolution scheme by chaining, see also Figure 1.13. Keys mapped to the same address are stored in a linked list at that address. The overall performance of the hash storage scheme depends on the

data and the hash function. A nearly statistically random distribution of the nodes to the hash table entries is the ideal case. Many heuristic hash function work very well in this respect.

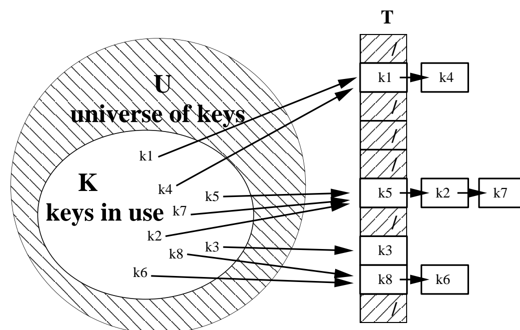

FIGURE 1.13. Hash table, collision resolution with chaining. Each node of a grid is mapped to an index in a vector by a hash function. The node and its node number are stored in the vector.

We have mentioned PDE solvers where the numerical data is stored in a tree data structure. There has also been an approach to store the numerical data directly in a hash table, see figure 1.14 (top) and [Sch98]. This means that all numerical algorithms are tied to the hash table. Even operations which are independent of any geometric information of the grid or the nodes such as the `saxpy` operation have to loop trough the hash table. This means unnecessary overhead and a performance penalty for this type of operation. Furthermore, for software design reasons one is interested to separate the geometric information from the algebraic data, i.e. the sparse grid from the vectors. A fixed amount of storage related to each hash table entry limits the implementation of numerical algorithms such as a BiCGstab solver or the solution of systems of PDEs.

Assume that we want to run a conjugate gradient iteration. In addition to the solution vector and the right hand side, five auxiliary vectors are required. We have to reserve five scalar variable in each hash table entry for this purpose, if we actually store data in the hash table. Assume furthermore that there are auxiliary fields for grid refinement, for the differential operator, for coefficients and so on. It is quite difficult to know in advance, how much auxiliary memory is needed in each hash table entry. Hence standard vectors separate from the hash table, which can be allocated and deleted easily, increase the flexibility of the overall code, see figure 1.14 (right). The hash table maps the node to a number, which serves as an index for all vectors. Hence floating point vectors, bit-fields and integer vectors can be addressed in the same way, without the danger of memory

alignment problems.

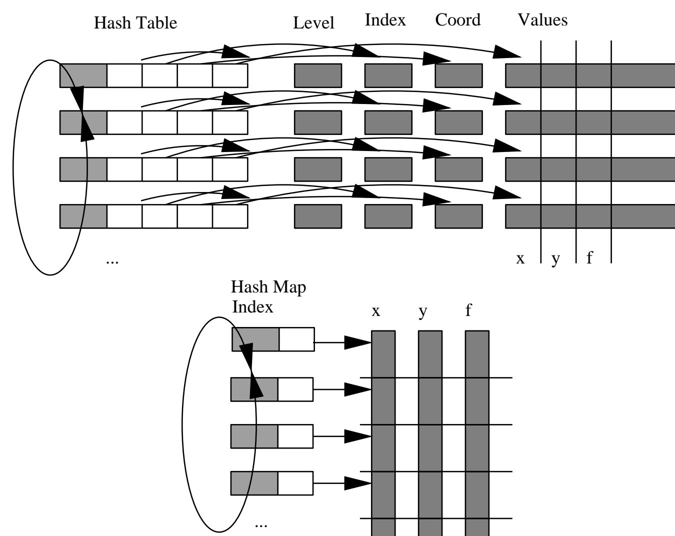

FIGURE 1.14. Numerical data storage within a hash table (upper half) and indexing of a hash table into standard vectors of different types located elsewhere. In the first case all numerical algorithms have to access the hash table in order to manipulate the data. In the second case however, numerical data is separate and some algorithms such as **saxpy** do not have to touch the hash table. Geometric (grid) and algebraic data (vectors) are separate.

1.5.3 Data on a grid: field

The solution of the partial differential equation, the right hand side and other functions represented on the sparse grid are stored as *fields*. The numerical data is contained in a *vector*, while the geometry is stored in a *grid*. The *field* abstraction glues both together and gives an interpretation of the vector as a continuous function. There are scalar fields and vector fields, both aligned to the grid (collocated) and in a staggered grid configuration, see figure 1.15. The *field* abstraction contains an interpolation procedure and it can perform I/O. Several output formats for different graphics packages are implemented for several types of fields. An input procedure is given below, which evaluates a scalar function on all nodes of a grid and stores the values at the appropriate locations in the numerical vector. The sparse grid is represented as a hash map, where each key is mapped to its node number. The STL implementation provides an iterator (*SpGrid::iterator*) to loop through all nodes in the hash table. The iterator points to a tuple of values, the coordinate vector and the node number, which can be accessed as *first* and *second*. The coordinate vector is transformed from local to global coordinate system by the reference point *x0* and the side lengths of the domain *h* and the key to coordinate conversion procedure *index1*.

The scalar function f is evaluated at that point and the value is stored in the vector *vec*, which is glued to the sparse grid by this instance of a field *SpField*.

```
void SpField:: readSF(ScalFunction& f)
{
  int d = grid->dim();
  SVec c(d);
  SpGrid::iterator i;            // STL iterator
  for (i = grid->hash.begin (); i != grid->hash.end (); i++) {
    for (int j=0; j<d; j++)
      c[j] = grid->x0[j] + grid->h[j] * grid->index1( (*i).first[j] );
    (*vec) [ (*i).second ] = f(c);     // data vector
  }
}
```

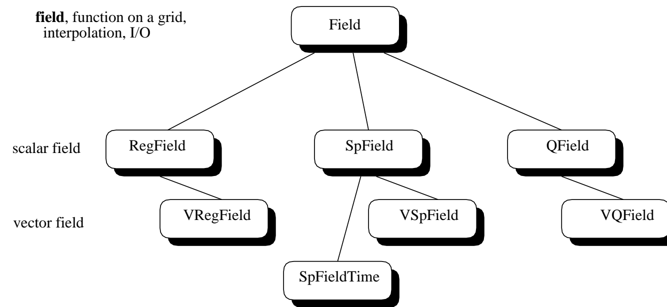

FIGURE 1.15. The abstractions for the functions defined on a grid.

The finite difference discretization on sparse grids is defined by hierarchical basis transformations $\mathbf{H}$ of a vector, one-dimensional finite difference stencils $\mathbf{D_i}$ and back-transforms $\mathbf{H}^{-1}$, see formula 1.5. The vectors are stored as instances of the class *Vec*, the transformations defined on a sparse grid are implemented in the *field* class and the difference stencils are located in the *operator* classes. The hierarchical basis transformation $\mathbf{H}$ and its inverse $\mathbf{H}^{-1}$ are coded within the *field* classes, because they depend heavily on the type of function representation encoded there. Furthermore, they are used in other parts of the code besides the finite difference operators. The transformation $\mathbf{H}$ is heavily used for the definition of finite difference operators. The implementation of the one-dimensional transformation of the nodal to hierarchical basis representation is given as an example and follows formula 1.3. The vector $\mathbf{x}$ contains the nodal representation and the hierarchical representation is stored in $\mathbf{y}$. The transformation is done along coordinate axis *dir*.

```
void SpField:: node2hierDir(const Vec& x, Vec& y, int dir) const
{
  assert(x.dim() == y.dim());
```

```
  for (SpGrid::iterator i = grid->begin(); i != grid->end(); i++)
    y[ (*i).second ] = x[ (*i).second ] - interpolDir(x, dir, (*i).first);
}

real SpField:: interpolDir(const Vec& x, int n, SIVec p) const
{
  if ((p[n] != 0)&&(p[n] != -1)) {     // boundary node?
      int il, ir;
      grid->neighbour(p[n], il, ir);  // compute neighbour keys
      p[n] = il;
      SpGrid::const_iterator l = grid->find(p);  // STL lookup
      p[n] = ir;
      SpGrid::const_iterator r = grid->find(p);
      return .5 * ( x[ (*l).second ] + x[ (*r).second ] ); // interpolation
  }
  return 0.;
}
```

1.5.4 Differential operators

The heart of the sparse grid research code are the partial differential operators. The straightforward representation of a discretized operator is a matrix, which is one implementation (a dense and a sparse matrix). Of course this matrix has to be initialized (or assembled). However, the number of non-zero entries for large sparse grids is prohibitively large and mainly the *apply* method of the operator is needed (i.e. matrix multiply). Hence another implementation of the operator is the finite difference sparse grid operator which implements '*apply*' on the fly. The sparse grid finite differences are implemented as defined in formula 1.5, i.e. by the hierarchical basis transformation $\mathbf{H}$ and back-transform $\mathbf{H}^{-1}$ coded within the *SpField* classes and the one-dimensional finite difference stencils $\mathbf{D}_i$ which are implemented in the various operator classes. Further implementations of operators include finite differences and finite elements on regular grids.

The *operator* classes are based on a certain type of function representations, given as a *field* class. Furthermore the hierarchical basis transform of the field is used. The *operator* abstraction contains some parameter data, such as the coefficient functions and references to a *field* and the underlying sparse *grid*. The class provides a method for applying the operator to some vector. This matrix multiplication is implemented here, calling the necessary basis transforms and applying finite difference stencils. Each PDE results in its own operator class.

We chose here that an operator works on vectors instead of fields both for several reasons: We are using the operator abstraction in a pure linear algebra context in a linear solver, where a *field* does not make sense. Furthermore, operating directly on vectors improves the performance of the code and we do not want to implement operators, where source and target field are represented on different types of grids. However, it would

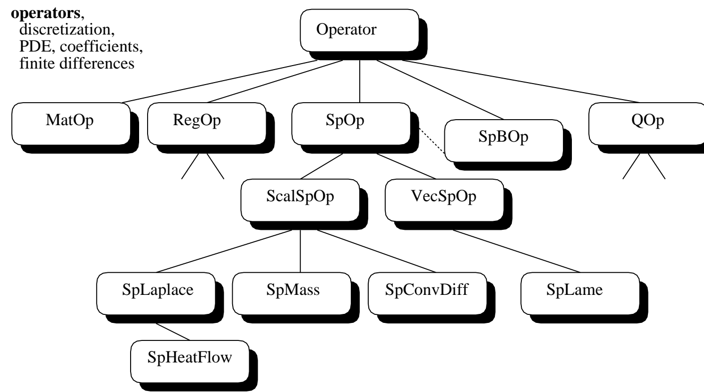

FIGURE 1.16. The discretized differential operators. The abstract *Operator* class is implemented as a matrix based operator, a finite element discretization on a regular grid, the sparse grid finite difference methods *SpOp*, a block grid version of sparse grids *SpBOp*, and a general finite difference discretization on adaptive grids *QOp*. There are scalar equations *ScalSpOp* and systems of equations *VecSpOp* and model problems like the Laplacian, convection-diffusion or the Lamé equation.

be possible to create a grid independent operator class, which operates on arbitrary fields. This would lead to more general concepts of algorithms.

1.5.5 Linear algebra

The goal of the linear algebra abstraction of an iterative Krylov solver is to provide a collection of algorithms, see figure 1.17. The abstraction is based on *vectors* and *operators*.

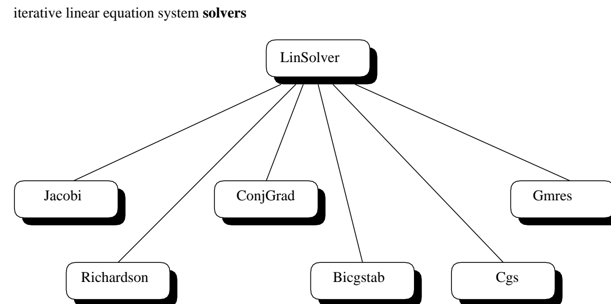

FIGURE 1.17. The iterative equation solvers.

We present the implementation of a conjugate gradient method to demonstrate that the code looks pretty much like a pseudo-code of a numerical analysis text book. A reference to the operator is stored as *op* and the vectors are fed as arguments b and x.

```
void ConjGrad:: solve(const Vec& b, Vec& x)
{
  assert(b.dim()==x.dim());
  int n = x.dim();
  Vec r(n), p(n), ap(n);
  iter = 0;
  real beta, alpha, rtr, rtrold;
  op->apply(x, ap);             // ap = A * x
  p.sub(b, ap);                 // p  = b - ap
  r.copy(p);                    // r  = p
  rtr = r.prod(r);              // rtr = <r,r>

  while (rtr>tol) {
    op->apply(p, ap);           // ap = A * p
    alpha = rtr / p.prod(ap);   // alpha = rtr / <p, ap>
    x.add(x,  alpha, p);        // x = x + alpha * p
    r.add(r, -alpha, ap);       // r = r - alpha * ap
    rtrold = rtr;
    rtr = r.prod(r);            // rtr = <r,r>
    beta = rtr / rtrold;
    p.add(r, beta, p);          // p = r + beta * p
    iter++;
    printIter("ConjGrad it ", " res= ", rtr);
    if ((iter>=maxiter)||(error>=HUGE)) break;
  }
  error = rtr;
}
```

Of course there are also preconditioned versions of the iterative Krylov solver. They look very similar to the un-preconditioned counterparts and take a preconditioning operator as argument.

1.5.6 Putting it together

Finally we print a short, but complete main program for the solution of a Poison equation on a sparse grid. We combine instances of all abstractions discussed above and hope that this type of coding is comprehensible and does not require further comments.

```
main() {
  int dim = 2;
  SVec x0(dim), x1(dim);        // lower left and upper right corner
  x0.set(-1.0);
  x1.set(1.0);
  SpGrid grid(dim, x0, x1);     // sparse grid on the square
  grid.refineAll();
  uint n = grid.nodes();
  Vec x(n), f(n);
  x.set(0.0);
  f.set(1.0);
  SpField field(x, grid);       // scalar solution field
  SpLaplace lap(field, 1.0);    // differential operator, coefficient
  Bicgstab j;                   // iterative solver
```

```
  j.attach(lap);
  j.setTol(1e-8);
  j.setMaxIter(10000);
  j.setVerbose(1);
  j.solve(f, x);                 // solve equation system
  cout<<j<<endl;                 // write statistics
  ofstream of("lap.vtk");
  field.print(of, Grid::vtk);   // dump solution in Vtk format
}
```

1.6 Parallel algorithms

The parallelization of an adaptive code usually is non-trivial and requires a substantial amount of code for the parallelization only. In this respect the parallelization of the sparse grid extrapolation method has a big advantage. If we are interested in a parallel version for adaptive sparse grids, however, we have to consider a more complicated approach to be described now.

Hierarchies of refined grids, where neighbor elements may reside on different processors, have to be managed [BH95]. That is, appropriate ghost nodes and elements have to be created and updated, when the parallel algorithm performs a communication operation. This happens both in the numerical part, where an equation system is set up and solved, and in the non-numerical part, where grids are refined and partitioned, see also [Bas98, JP97].

The key point of any dynamic data partition method is efficiency. We look for a cheap, linear time heuristic for the solution of the partition problem. Furthermore the heuristic should parallelize well. Here, parallel graph coarsening is popular. It results in a coarser graph on which then a more expensive heuristic on a single processor can be employed. However, graph coarsening is at least a linear time algorithm itself and lowers the quality of the heuristic further. This is why we look for even cheaper partition methods. They are provided by the concept of *space-filling curves.*

1.6.1 Space-filling curves

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

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

One of the oldest and most prominent space-filling curve, the Hilbert curve can be defined geometrically [Hil91], see also [Sag94]. If the interval I can

be mapped to Q by a space-filling curve, then this must be true also for the mapping of four quarters of I to the four quadrants of Q, see Figure 1.18. Iterating this sub-division process, while maintaining the neighborhood relationships between the intervals leads to the Hilbert curve in the limit case. The mapping is defined by the recursive sub-division of the interval I and the square Q.

FIGURE 1.18. Construction of a Hilbert space-filling curve. The open and the closed curve.

Often some curves as intermediate results of iterative construction procedure are more interesting than the final Hilbert curve itself. The construction begins with a generator, which defines the order in which the four quadrants are visited. The generator is applied in every quadrant and their sub-quadrants. By affine mappings and connections between the loose ends of the pieces of curves, the Hilbert curve is obtained.

There are basically two different Hilbert curves for the square, modulo symmetries, see Figure 1.18. An open and a closed space-filling curve can be constructed by the Hilbert curve generator maintaining both neighborhood relations and the sub-division procedure. The resulting curve is indeed continuous and space-filling. Although each curve of the iterative construction is injective and does not cross itself, the resulting Hilbert curve is not injective. This can be demonstrated easily by looking at the point $(1/2, 1/2) \in Q$, which is contained in the image of all four quadrants. Hence several points on I are mapped to this point of Q. In three dimensions, there are 1536 versions of Hilbert curves for the unit cube [AN98].

The Hilbert curve may be the most prominent curve. However it is not the oldest one. The discovery of 'topological monsters', as they were called initially started when Cantor in 1878 proved the existence of a bijective mapping between two smooth, arbitrary manifolds of arbitrary, but finite dimension. A year later Netto proved that such mappings are non-continuous. This means that they are not curves in our previous definition. However, without the property of an injective mapping, Peano constructed a space-filling curve in 1890, see Figure 1.19. Later on several other curves were constructed by Hilbert, Moore, Lebesgue and Sierpinski and many others.

The Peano space-filling curve can be constructed by a sub-division scheme of the square Q into nine squares of side length $1/3$. The generator can be given by the order of the nine squares. In contrary to the Hilbert curve there are 273 different curves, which maintain neighborhood relations. The

original construction of the Peano curve was based on a triangulation of the domain and a bisection scheme for triangles, which defines the order in which the sub-triangles are visited. We will use this scheme later on.

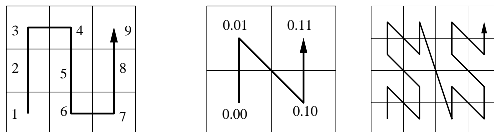

FIGURE 1.19. The construction of a Peano (left) and Lebesgue curve (middle and right).

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

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

The function f defined on the Cantor set C is extended to the unit interval I by linear interpolation. The generator of the Lebesgue curve looks like in Figure 1.19. Although constructed by digit shuffling, the curve is continuous and a space-filling curve. The order of the quadrants imposed by the generator of the curve can be found in the depth-first traversal of oct-trees and related algorithms.

1.6.2 Applications of space-filling curves

Space-filling curves had been created for purely mathematical purposes. However, nowadays there is a number of applications of space-filling curves. Basically, multi-dimensional data are mapped to a one-dimensional sequence. This mapping is useful for load-balancing of parallel simulations on a computer, for data locality of memory or disc accesses inside a computer in geographic information systems [ARR+97], for finding shortest paths in optimization [BP88, BG88] and ordering data in computer graphics [PKK92, WZ93, Voo94] and in other applications [BS98].

Another interesting application of space-filling curves are particle or n-body problems which are defined by the interaction of n entities by some interaction forces. This model describes different phenomena like the movement of planets or dust under gravitational forces in astrophysics and the dynamics of atoms or groups of atoms in molecular dynamics. The number of particles of interest easily reaches the range of $10^6 - 10^9$. The model leads to a system of n ordinary differential equations. The right hand side of each equation consists of the $n - 1$ forces of particles, which interact with the particle under consideration. Usual model forces decay with the distance of the particles, which can be exploited by efficient approximation algorithms like the fast-multipol and the Barnes-Hut algorithm. However, there is still a global coupling of the particles, which cannot be neglected. Furthermore, particle can be distributed randomly and can form clusters. Hence a parallel particle simulation code requires efficient load-balancing strategies for the enormous amount of data. The particle moves in space due to the acceleration imposed by the interaction-forces. This means, that a re-balancing or a dynamic load-balancing is needed for parallel computing, which can be done by space-filling curves [WS95, SWW, PB96].

A slightly different situation can be found in adaptive discretizations of partial differential equations. Now a grid consisting of n nodes and elements or volumes has to be distributed to a parallel computer. The nodes and elements can be found at arbitrary positions (completely unstructured grids) or at fixed positions, which are a priori known (structured grids) or at least computable by grid-refinement rules from a coarse grid (adapted grids). The degrees of freedom are coupled locally, usually between neighboring nodes. However, algorithms for the solution of the resulting equation systems couple all degrees of freedom together, which imposes difficulties on the parallelization of the solver. For the solution of stationary problems, no nodes are moved in general. However, due to adaptive grid refinement, new nodes are created during the computation. This requires dynamic load-balancing, which can also be done by space-filling curves [OPF94, PB96, GZ98b, RKCHC98].

1.6.3 Parallel sparse grid algorithms

First of all we want to convert the sequential finite difference sparse grid code into a parallel code. There will be some extensions and slight modifications, but the general design and most of the lines of code are retained. The parallel version, run on a single processor, will only introduce very little overhead. However, the parallel version run on a larger distributed memory parallel computer with p processors should ran ideally at p times the speed of the sequential code, at least for problems large enough, where enough computational load can be distributed to the processors. We are interested in high number of processors. Hence we consider a message passing programming model, where each processor can access its own memory only.

Smaller shared memory parallel computers offer more convenient programming models and have been used for a similar code based on automatic parallelization by loop-parallelism, see [Sch98].

In the last chapter we have introduced a key-based node addressing scheme, where each node is stored in a hash map. The advantage in the sequential version was simplicity and little administration overhead. The parallel version is based on the distribution of nodes to the processors. Each processor owns a subset of the sparse grid. Each node is present on exactly one processor. The appropriate process stores the node in a local hash map. The difference to the sequential code is, that the hash map does not contain all nodes any longer, but the nodes owned by the processor. Additionally, at some stages of the algorithm there are ghost nodes present, which contain the values of nodes belonging to other processors, but whose values are required by the algorithm. The values of the ghost nodes are updated or filled in a communication step prior to the actual computation. During the creation of such ghost nodes, processors have to communicate which nodes are required: Processor 1 determines which ghost nodes it needs. It finds out that some of the nodes originally belong to processor 2 and asks for them. Hence processor 2 knows that during the communication step, it has to send this data to processor 1. This negotiations would be quite complicated in a code based on pointers, because a reference to local memory does not make sense on another processor's memory. However, using keys, i.e. a unique id for each ode derived from its coordinates, there is no problem at all. The global key is understood by all processors and can be used for all kind of requests. Furthermore, the space-filling curve, which provides a unique mapping of nodes to processors, immediately reveals, which processor to ask for a node. Other grid partitioning heuristics in contrast would require a substantial bookkeeping effort to decide where a node belongs to.

One detail is still missing: In an adaptive sparse grid, it is not exactly clear, where to look for a neighbor node, while father and son nodes are determined. Hence there has to be some searching procedure for the appropriate neighbor node. Several requests for potential nodes may have to be raised. The node might or might not exist and might or might not be the nearest node in a certain direction. Note that these request might be addressed to different processors, because each requested node can belong to another processor.

We list some hints on the parallel version of the stages of the sparse grid discretization and equation solver:

- Krylov iteration: Conjugate-gradient type of Krylov methods like BiCG, BiCGstab and CGS can be parallelized such that, the parallel version of an iterative Krylov method looks exactly like the sequential one. Each processor operates on the unknowns related to its own nodes. The Blas level one operations of *saxpy* type do not require communication. The communication of a Blas level one scalar

products can be implemented by local summation and a standard reduce operation over all processors. This communication library call is hidden in the vector classes scalar product in our implementation. The main source of trouble is the application of the operator (matrix multiply) and a preconditioner, which is implemented separately.

- Finite difference operator: The operator

$$\sum_{i=1}^{d}\Big(\prod_{j=1,\ j\neq i}^{d}\mathbf{H}_j^{-1}\Big)\circ\mathbf{D}_i\circ\Big(\prod_{j=1,\ j\neq i}^{d}\mathbf{H}_j\Big) \tag{1.7}$$

  is composed of three basic operations, the transform to hierarchical basis $\mathbf{H}_j$, the one-dimensional finite difference stencil $\mathbf{D}_i$ and the transform back to nodal basis $\mathbf{H}_j^{-1}$, which are implemented separately, see also section 1.3.3.

  - Transform to hierarchical basis $\mathbf{H}_j$: Each processor computes the values related to its own nodes. Prior to the computation, in a communication step the required ghost nodes are filled. The ghost nodes for this operation are determined by the direct parent nodes of nodes on the processor.
  - Transform to nodal basis $\mathbf{H}_j^{-1}$: This operation can be done in place and requires more communication than the previous one. The sequential implementation cycles through a tree top down, so that the parent nodes are processed before their children. A straightforward parallelization would be to insert a communication step before each tree level is traversed. However, this results in a number of communication steps (= communication latencies) proportional to the maximum number of levels, which is unacceptable for large sparse grids.

    We propose an alternative implementation here, which is based on a single communication step before the computation: Along with with the parents of a node, the whole tree of their grandparents and so on are required as ghost nodes on a processor. When the ghost nodes are filled, the computation can be done top down, such that the values on all nodes owned by the processor and additionally their parents, grand-parents and so on are computed. Hence, this implementation requires a larger amount of computation and a larger volume of communication than the straightforward version. However, the overall execution time is smaller because of the number of communication steps is reduced to one.
  - Finite difference operator $\mathbf{D}_i$: First the appropriate ghost nodes for the difference stencil is filled and afterwards the stencils are applied to all nodes, which belong to the processor. The main

point here is the searching procedure for the neighbor nodes that are necessary for adaptive refined sparse grids. We create the necessary ghost nodes, so that the sequential search algorithm can be re-used in this situation.

- Adaptive grid refinement: Following some refinement rules, new nodes are created. This can be done also in parallel. Afterwards, a repartitioning has to takes place, which is also responsible for the elimination of multiple instances of a node that might have been created during the grid refinement. The partitioning of nodes is done by the space-filling curve heuristic and can be implemented as cutting a sorted list of all nodes into equal sized pieces. Of course, one has to avoid storing all nodes on a single processor. That is why the sorting is also executed in parallel by a bucket sort algorithm. The old partitioning serves as the buckets, each one mapped to one processor. Afterwards the new partitioning is computed. This procedure can be run completely in parallel and scales very well. The execution time of this repartitioning step usually is so low that it is below .01 of the execution time spent in the numerical algorithms, see also [Zum99].

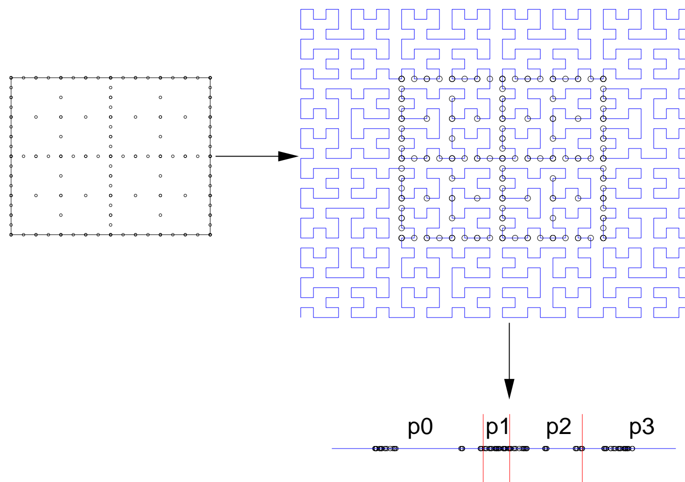

FIGURE 1.20. An example of a sparse grid partitioned and mapped to four processors. The sparse grid (left) can be represented by its nodes in coordinate space. A Hilbert type space-filling curve, which fills the domain, is laid over the grid (right). Each node lies on the curve. Now we straighten the space-filling curve with the nodes fixed to the curve (bottom). The interval is cut into four sub-intervals assigned to one processor, each containing the same number of nodes.

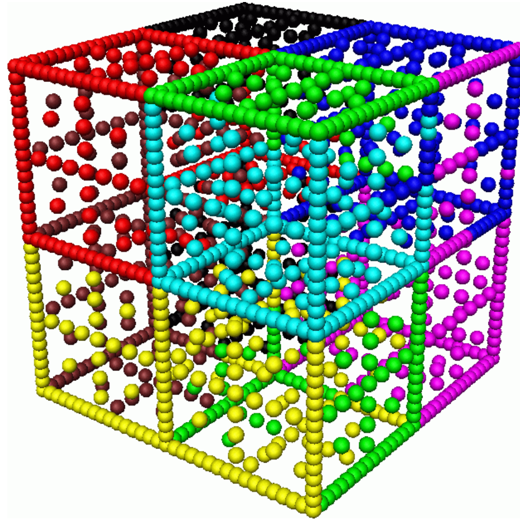

FIGURE 1.21. A three-dimensional sparse grid, partitioned to eight processors (color coded).

All numbers reported are scaled CPU times measured on our parallel computing cluster 'Parnass2'. It consists of dual processor Pentium II 400MHz BX boards with at least 256 Mbytes of main memory, interconnected by a Myrinet network in a fat-tree configuration. The MPI message passing protocol implementation Mpich-PM showed a bandwidth between each two boards of 850 Mbit/s, see also [SZG99].

In the first test we consider the solution of a three-dimensional convection-diffusion equation discretized on sparse grids with up-winding (standard refinement). Table 1.2 shows wall clock times for the solution of the equation system on a sparse grid of different levels using different numbers of processors.

For a fixed number of processors, we observe a scaling of a factor slightly above of 2 from one level to the next finer level, which corresponds to a similar factor of increase in the amount of unknowns on that level. Furthermore, for a fixed level the measured times scale roughly with $1/p$ of the number of processors p up to a parallel efficiency of 0.4 for 64 processors. However, the 32 and 64 processors perform efficiently only for sufficiently large problems, i.e. for problems with more than some thousands degrees of freedom. If we fix the amount of work, that is the number of nodes per processor, we obtain the scale-up. Comparing a time at one level l and a number of processors p with the time of one level finer $l+1$ and $2p$ processors, we obtain very well scaling of the method. Note that in this case of uniform grid refinement, some a priori partition schemes would be superior to our dynamic partitioning scheme. However, our dynamic load balancing

time		processors						
nodes	1/h	1	2	4	8	16	32	64
81	4	0.03	0.03	0.07	0.08	0.11		
225	8	0.12	0.09	0.09	0.11	0.16	0.20	0.20
593	16	0.63	0.41	0.33	0.32	0.38	0.44	0.53
1505	32	3.78	2.29	1.60	1.34	1.26	1.33	1.53
3713	64	22.1	13.3	8.79	6.39	5.17	4.47	4.47
8961	128	68.1	40.7	24.8	16.2	11.9	8.89	7.56
21249	256	201	119	66.1	40.1	28.0	18.6	13.5
49665	512	575	379	169	106	71.6		28.0
114689	1024	1630			275	179		62.6

TABLE 1.2. Parallel execution times for a 3D convection-diffusion problem on sparse grids. Solution of the equation system in seconds on Parnass2.

scheme performs well and introduces only little overhead and results in good partitions, see figure 1.21.

In the next test we consider adaptively refined sparse grids for a problem with singularities, where the sparse grids are refined towards a singularity located in the lower left corner, see also Figure 1.8. Table 1.3 depicts times in the adaptive case. These numbers give the wall clock times for the solution of the equation system again, now on different levels of adaptive grids and on different numbers of processors. Due to the solution-dependent adaptive refinement criterion, the single processor version contained slightly more nodes, indicated by *. For the same reason, the equation systems have been solved up to rounding error instead of the weaker discretization error condition in the uniform sparse grid experiment.

time		processors					
nodes	1/h	1	2	4	8	16	32
81	4	0.03	0.04	0.05	0.07	0.11	
201	8	0.07	0.05	0.05	0.07	0.08	0.09
411	16	0.21	0.13	0.12	0.13	0.17	0.20
711	32	0.78	0.48	0.38	0.36	0.41	0.51
1143	64	2.60	1.49	1.06	0.93	0.92	1.14
1921	128	8.69	5.99	3.70	2.88	2.70	2.83
3299	256	39.3*	20.7	13.8	9.62	7.79	7.32
6041	512	177*	91.0	56.8	39.5	28.6	22.0
11787	1024	949*	525	271	177	138	88.2
22911	2048			1280	761	660	

TABLE 1.3. Parallel execution times for adaptive sparse grids. A 3D convection-diffusion problem is solved and the solution times in seconds on Parnass2 are given.

We obtain a good scaling, both in the number of unknowns and in the number of processors, i.e. the times are proportional to the number of unknowns for a fixed number of processors and are indirect proportional to the number of processors. Increasing the number of processors speeds up the computation accordingly. The parallel efficiencies are somewhat smaller than for the uniform refinement case, due to the imbalance in the tree of nodes. This is also the case for other parallel adaptive methods. Hence this parallelization approach does perform very well, even in the range of higher number of processors 16 and 32, where a number of other strategies are not competitive.

1.7 Application to time-dependent problems

Sparse grid discretizations have been applied so far to stationary problems and to time-dependent, transient problems. Besides methods based on the extrapolation method, where standard time-dependent codes can be re-used, there are also attempts to solve parabolic problems [Bal94] and Navier-Stokes equations [Sch98] by native sparse grid discretizations. We will discuss some features of a sparse grid finite difference discretization of space-time for scalar parabolic and hyperbolic equations.

We are interested in the numerical solution of

$$\begin{array}{rcll} u_t + \nabla \cdot f(u) & = & q(u) & \text{for } u(x,t), \\ x & \in & \Omega \subset \mathbb{R}^d & \\ t & \in & [0, t_0] & \end{array} \tag{1.8}$$

written as an initial-boundary value problem. The standard procedure for the numerical solution of (1.8) is to discretize the space Ω and the initial value $u^0 = u(x,0)$ on Ω and to step forward in time. The solution u^{t+1} at time step $t+1$ is computed from u^t and the boundary conditions. This *'time stepping'* scheme is iterated until $t = t_0$ is reached.

1.7.1 *Space-time discretization*

An alternative solution algorithm uses a discretization of (1.8) in the *'space-time'* domain $\Omega \times [0, t_0]$, see figure 1.22. The transient problem can be re-written as a boundary value problem

$$\begin{array}{rcll} \nabla \cdot F(u) & = & q(u) & \text{for } u(x), \\ x & \in & \Omega \times [0, t_0] \subset \mathbb{R}^{d+1} & \end{array} \tag{1.9}$$

with F given by the components $F_0(u) = u$ and $F_{1,\dots,d}(u) = f(u)$. A standard finite difference or finite element discretization, implicit in time, on a regular grid leads to a sequence equation systems, for one time step each. However, the corresponding sparse grid discretization, where different

$2/h$	L_1	L_2	L_∞	L_1	L_2	L_∞
32	1.8973	1.8713	1.7746	1.4347	1.6189	1.6752
64	1.9348	1.9194	1.8696	1.5539	1.7186	1.7834
128	1.9610	1.9534	1.9297	1.6518	1.7985	1.8652
256				1.7247	1.8558	1.9235
512				1.7746	1.8932	1.9606

TABLE 1.4. Convergence rates on regular (left) and on sparse grids, sine wave initial data for the linear advection equation.

step sizes in time and in space are coupled, requires the numerical solution of a single large (non-) linear equation system and returns an approximation of u at all time steps at once. The hierarchical basis transformation glue all time slices and all location in space together. Together with an iterative equation solver on the global equation system, this approach is related to the waveform relaxation [Van92]. It has been used for standard grids by many authors, see e.g. [McC89] and with the Galerkin method for periodic parabolic equations on sparse grids in [Bal94].

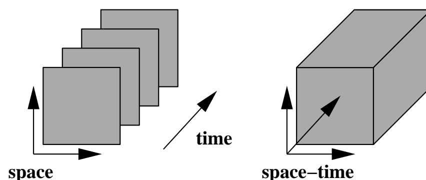

FIGURE 1.22. Time stepping vs. space-time discretization.

We present two test cases of finite difference sparse grid discretizations for problems in space-time. The heat conduction equation (parabolic, $u_t = \Delta u$) and the linear advection equation (hyperbolic, $u_t + u_x = 0$). We prescribe some simple initial values and use a zero source term $q \equiv 0$. The boundary conditions of the heat conduction as well as the inflow of the advection are set to zero. The solutions are depicted in figures 1.23 and 1.24 resp. 1.25. Errors and convergence rates are given in table 1.4. The ratio of number of unknowns to global error is much better on sparse grids than on standard grids. We obtain a weaker dependence on the dimension for sparse grid discretizations, as predicted. However, the actual performance depends on the smoothness of the solution in space-time, which is present for the heat conduction problem, but is missing for typical solutions of hyperbolic equations. That is why we consider adaptive grid refinement next. For details and further considerations and numerical schemes for the solution of hyperbolic equations on sparse grids we refer to [GZ98a].

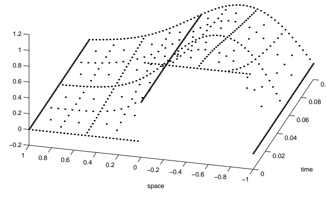

FIGURE 1.23. Sparse grid solution of the parabolic heat conduction equation in 2D space-time.

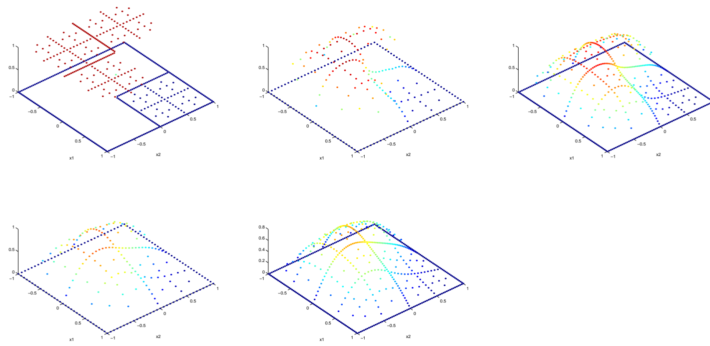

FIGURE 1.24. Sparse grid solution of the parabolic heat conduction equation in 3D space-time. Snapshots at different times t

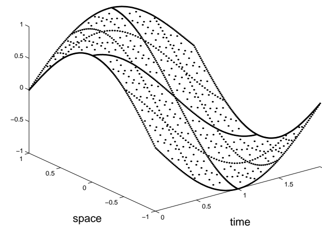

FIGURE 1.25. Sparse grid solution of the hyperbolic linear advection equation in 2D space-time with a sine wave as initial data.

1.7.2 Adaptive grid refinement and macro time stepping

The storage requirements of the space-time formulation are often considered as prohibitively high. However, by the sparse grid technique the additional dimension in storage is affordable. Furthermore there is the advantage of easy adaptive grid refinement in space-time. In any stage of the computation it is possible to introduce a finer grid or additional nodes, which gives better resolution in space and local time steps. This is often difficult or even impossible for time stepping algorithms, where the time-steps have to be computed again at a finer scale.

Adaptive grid refinement added some features to the solution algorithm for time-dependent problems: There is only one error indicator operating in the space-time domain, instead of several indicators, which operate separately in space and time and are coupled through a CFL type of stability condition. Grid refinement now enhances local space resolution and at the same time introduces local time stepping. However, due to the discretization implicit in time, there is no temporal stability condition.

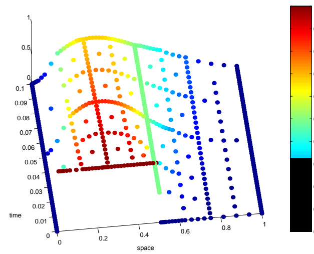

FIGURE 1.26. Adaptive sparse grid solution in space-time.

Initial value problems, where the final time t_1 is unknown, cannot be discretized in space-time straightforward. However, variable time steps can be introduced with a macro time stepping. Each macro step is based on a single sparse grid in space-time. Putting the macro steps together in the way of time stepping methods gives the whole time interval $[t_0, t_1]$, see figure 1.27.

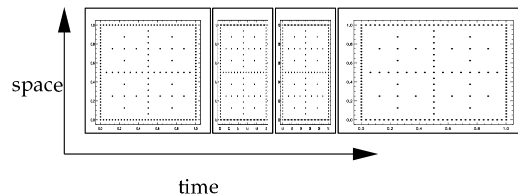

FIGURE 1.27. Macro time stepping in a space-time scheme.

1.8 Conclusion

We have introduced sparse grids, several ways to discretize partial differential equations on sparse grids and strategies to create adaptive sparse grids. This includes the extrapolation method, the Galerkin method and the finite difference method, error estimators and grid refinement algorithms. Furthermore an object oriented software design for a finite difference sparse grid PDE solver was proposed. The parallelization of such a code by means of space-filling curves was discussed, along with experimental results of the parallel code on a cluster of PCs. Applications of the sparse grid discretizations to time-dependent parabolic and hyperbolic equations in space-time concluded this survey.

1.9 References

[ABL97] E. Arge, A. M. Bruaset, and H. P. Langtangen. Object-oriented numerics. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, Basel, 1997.

[AN98] J. Alber and R. Niedermeier. On multi-dimensional Hilbert indexings. In *Proc. of the Fourth Annual International Computing and Combinatorics Conference (COCOON'98), Taipei 1998*, Lecture Notes in Computer Science. Springer, 1998.

[ARR+97] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181:3–15, 1997.

[Aus99] M. H. Austern. *Generic Programming and the STL.* Addison-Wesley, 1999.

[Bal94] R. Balder. *Adaptive Verfahren für elliptische und parabolische Differentialgleichungen auf dünnen Gittern.* PhD thesis, TU München, Inst. für Informatik, 1994.

[Bas98] P. Bastian. Load balancing for adaptive multigrid methods. *SIAM J. Sci. Comput.*, 19(4):1303–1321, 1998.

[BDZ96] H.-J. Bungartz, T. Dornseifer, and C. Zenger. Tensor product approximation spaces for the efficient numerical solution of partial differential equations. In *Proc. Int. Workshop on Scientific Computations*, Konya, 1996. Nova Science Publishers, Inc. to appear.

[BG88] D. Bertsimas and M. Grigni. On the spacefilling curve heuristic for the Euclidean traveling salesman problem. Technical report, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[BH95] K. Birken and C. Helf. A dynamic data model for parallel adaptive PDE solvers. In B. Hertzberger and G. Serazzi, editors, *Proceedings of HPCN Europe 1995*, volume 919 of *Lecture Notes in Computer Science*, Milan, Italy, 1995. Springer.

[BL97] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In M. Dæhlen and A. Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics.* Birkhäuser, Basel, 1997.

[BN94] J. J. Barton and L. R. Nackman. *Scientific and Egineering C++ – An Introduction with Advanced Techniques and Examples.* Addison-Wesley, Reading, Ms., 1994.

[BP88] J. Bartholdi and L. K. Platzman. Heuristics based on spacefilling curves for combinatorial optimization problems. *Management Science*, 34:291–305, 1988.

[BR95] R. Becker and R. Rannacher. Weighted a posteriori error control in FE methods. In *Proc. ENUMATH 95*, 1995.

[BS98] I. Beichl and F. Sullivan. Interleave in peace, or interleave in pieces. *IEEE Computational Science & Engineering*, 5(2):92–96, 1998.

[Bun92] H.-J. Bungartz. *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung.* PhD thesis, TU München, Inst. für Informatik, 1992.

[Bun98] H.-J. Bungartz. *Finite Elements of Higher order on Sparse Grids.* PhD thesis, TU München, Inst. für Informatik (Habilitation), 1998.

[Fab09] G. Faber. Über stetige Funktionen. *Mathematische Annalen*, 66:81–94, 1909.

[GK98] M. Griebel and S. Knapek. Optimized approximation spaces for operator equations. Technical Report 568, University Bonn, SFB 256, 1998.

[Gri92] M. Griebel. The combination technique for the sparse grid solution of PDEs on multiprocessor machines. *Parallel Processing Letters*, 2:61–70, 1992.

[Gri98] M. Griebel. Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences. In *Proc. Large Scale Scientific Computations, Varna, Bulgaria*. Vieweg, 1998.

[GSZ92] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In P. de Groen and R. Beauwens, editors, *Iterative Methods in Linear Algebra*, pages 263–281. IMACS, Elsevier, 1992.

[GZ98a] M. Griebel and G. Zumbusch. Adaptive sparse grids for hyperbolic conservation laws. In *Proceedings of Seventh International Conference on Hyperbolic Problems, Zurich*. Birkhäuser, 1998.

[GZ98b] M. Griebel and G. Zumbusch. Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization. In J. Mandel, C. Farhat, and X.-C. Cai, editors, *Proc. Domain Decomposition Methods 10*, volume 218 of *Contemporary Mathematics*, pages 279–286, Providence, Rhode Island, 1998. AMS.

[Har95] A. Harten. Multi-resolution representation of data: A general framework. *SIAM J. Numer. Anal.*, 33:1205–1256, 1995.

[HBZ95] S. Hilgenfeldt, S. Balder, and C. Zenger. Sparse grids: Applications to multi-dimensional Schrödinger problems. Technical Report 342/05/95, TU München, Inst. für Informatik, SFB 342, 1995.

[Hem95] P. W. Hemker. Sparse-grid finite-volume multigrid for 3D-problems. *Advances in Computational Mathematics*, 4:83–110, 1995.

[Hil91] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[JP97] M. T. Jones and P. E. Plassmann. Parallel algorithms for adaptive mesh refinement. *SIAM J. Sientific Computing*, 18(3):686–708, 1997.

[Knu75] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1975.

[Kup97] F. Kupka. *Sparse Grid Spectral Methods for the Numerical Solution of Partial Differential Equations with Periodic Boundary Conditions*. PhD thesis, Universität Wien, Inst. für Math., 1997.

[LLS95] C. B. Liem, T. Lu, and T. M. Shih. *The Splitting Extrapolation Method: A New Technique in Numerical Solution of Multidimensional Problems*, volume 7 of *Applied Mathematics*. World Scientific, 1995.

[McC89] S. F. McCormick. *Multilevel Adaptive Methods for Partial Differential Equations*. Frontiers in Applied Mathemtics. SIAM, 1989.

[OPF94] J. T. Oden, A. Patra, and Y. Feng. Domain decomposition for adaptive hp finite element methods. In *Proc. Domain Decomposition 7*, volume 180 of *Contemporary Mathematics*, pages 295–301. AMS, 1994.

[Osw98] P. Oswald. Best N-term approximation of singularity functions in two Haar bases. Technical report, Bell Labs, Lucent Technologies, 1998.

[PB96] M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawai International Conference on System Sciences*, 1996.

[PKK92] A. Pérez, S. Kamata, and E. Kawaguchi. Peano scanning of arbitrary size images. In *Proc. Int. Conf. Pattern Recognition*, pages 565–568, 1992.

[RKCHC98] S. Roberts, S. Kalyanasundaram, M. Cardew-Hall, and W. Clarke. A key based parallel adaptive refinement technique for finite element methods. In *Proc. Computational Techniques and Applications: CTAC '97*. World Scientific, 1998. to appear.

[Rüd92] U. Rüde. Data structures for multilevel adaptive methods and iterative solvers. Technical Report I-9217, TU München, Inst. für Informatik, 1992.

[Sag94] H. Sagan. *Space-Filling Curves*. Springer, New York, 1994.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[Sch98] T. Schiekofer. *Die Methode der Finiten Differenzen auf dünnen Gittern zur Lösung elliptischer und parabolischer partieller Differentialgleichungen*. PhD thesis, Universität Bonn, Inst. für Angew. Math., 1998. to appear.

[Smo63] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. *Dokl. Akad. Nauk SSSR*, 4:240–243, 1963.

[SWW] J. K. Salmon, M. S. Warren, and G.S. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *International Journal of Supercomputer Applications*, 8(2).

[SZ98] T. Schiekofer and G. Zumbusch. Software concepts of a sparse grid finite difference code. In W. Hackbusch and G. Wittum, editors, *Proceedings of the 14th GAMM-Seminar Kiel on Concepts of Numerical Software*, Notes on Numerical Fluid Mechanics. Vieweg, 1998.

[SZG99] M. A. Schweitzer, G. Zumbusch, and M. Griebel. Parnass2: A cluster of dual-processor PCs. In W. Rehm and T. Ungerer, editors, *Proceedings of the 2nd Workshop Cluster-Computing*, number CSR-99-02 in Informatik Berichte. University Karlsruhe, TU Chemnitz, 1999.

[Tem89] V. N. Temlyakov. Approximation of functions with bounded mixed derivative. *Proc. of the Steklov Institute of Mathematics*, 1, 1989.

[Van92] S. Vandewalle. *Parallel Multigrid Waveform Relaxation for Parabolic Problems.* Teubner, Stuttgart, 1992.

[Ver96] R. Verfürth. *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques.* J. Wiley & Teubner, 1996.

[Voo94] D. Voorhies. Space-filling curves and a measure of coherence. In J. Arvo, editor, *Graphics Gems II*, pages 26–30. Academic Press, 1994.

[WS95] M. Warren and J. Salmon. A portable parallel particle program. *Comput. Phys. Comm.*, 87:266–290, 1995.

[WZ93] R. E. Webber and Y. Zhang. Space diffusion: An improved parallel halfoning technique using space-filling curves. In *Proc. ACM Comput. Graphics Ann. Conf. Series*, page 305ff, 1993.

[Yse86] H. Yserentant. On the multilevel splitting of finite element spaces. *Numer. Math.*, 49:379–412, 1986.

[Zen91] C. Zenger. Sparse grids. In W. Hackbusch, editor, *Proc. 6th GAMM Seminar*, Kiel, 1991. Vieweg.

[Zum99] G. Zumbusch. Dynamic loadbalancing in a lightweight adaptive parallel multigrid PDE solver. In B. Hendrickson, K. Yelick, C. Bischof, I. Duff, A. Edelman, G. Geist,

M. Heath, M. Heroux, C. Koelbel, R. Schrieber, R. Sinovec, and M. Wheeler, editors, *Proceedings of 9th SIAM Conference on Parallel Processing for Scientific Computing (PP 99)*, San Antonio, Tx., 1999. SIAM.