ESAIM: PROCEEDINGS, December 2011, Vol. 34, p. 151-183
E. Cancès, V. Louvet and M. Massot, Editors

ADAPTIVE MULTIRESOLUTION METHODS: PRACTICAL ISSUES ON DATA STRUCTURES, IMPLEMENTATION AND PARALLELIZATION *

K. Brix[1], S. Melian[1], S. Müller[1], M. Bachmann[1]

Abstract. The concept of fully adaptive multiresolution finite volume schemes has been developed and investigated during the past decade. Here grid adaptation is realized by performing a multiscale decomposition of the discrete data at hand. By means of hard thresholding the resulting multiscale data are compressed. From the remaining data a locally refined grid is constructed.

The aim of the present work is to give a self-contained overview on the construction of an appropriate multiresolution analysis using biorthogonal wavelets, its efficient realization by means of hash maps using global cell identifiers and the parallelization of the multiresolution-based grid adaptation via MPI using space-filling curves.

Résumé. Le concept des schémas de volumes finis multi-échelles et adaptatifs a été développé et etudié pendant les dix dernières années. Ici le maillage adaptatif est réalisé en effectuant une décomposition multi-échelle des données discrètes proches. En les tronquant à l'aide d'une valeur seuil fixée, les données multi-échelles obtenues sont compressées. A partir de celles-ci, le maillage est raffiné localement.

Le but de ce travail est de donner un aperçu concis de la construction d'une analyse appropriée de multiresolution utilisant les fonctions ondelettes biorthogonales, de son efficacité d'application en terme de tables de hachage en utilisant des identification globales de cellule et de la parallélisation du maillage adaptatif multirésolution via MPI à l'aide des courbes remplissantes.

1. Introduction

In recent years, a new adaptive concept for finite volume schemes, frequently applied to the discretization of balance equations arising for instance in continuum mechanics, has been developed based on multiscale techniques. First work in this regard has been published by Harten [27, 28]. The basic idea is to transform the arrays of cell averages associated with any given finite volume discretization into a different format that reveals insight into the local behavior of the solution. The cell averages on a given highest level of resolution (*reference mesh*) are represented as cell averages on some coarse level, where the fine scale information is encoded in arrays of *detail coefficients* of ascending resolution.

In Harten's original approach, the multiscale analysis is used to control a hybrid flux computation by which CPU time for the evaluation of the numerical fluxes can be saved, whereas the overall complexity is not reduced but still stays proportional to the number of cells on the uniformly fine reference mesh. Opposite to this strategy,

* *This work has been performed with funding by the Deutsche Forschungsgemeinschaft in the Collaborative Research Centre SFB 401 "Flow Modulation and Fluid-Structure Interaction at Airplane Wings" of the RWTH Aachen University, Aachen, Germany and by the Excellence Initiative of the German federal and state governments.*

[1] Institut für Geometrie und Praktische Mathematik, RWTH Aachen University, Templergraben 55, 52056 Aachen, Germany, email: {brix,melian,mueller,bachmann}@igpm.rwth-aachen.de

threshold techniques are applied to the multiresolution decomposition in [15,37], where detail coefficients below a threshold value are discarded. By means of the remaining significant details, a locally refined mesh is determined whose complexity is substantially reduced in comparison to the underlying reference mesh.

The fully adaptive concept has turned out to be highly efficient and reliable. So far, it has been employed with great success in different applications, e.g., 2D/3D–steady and unsteady computations of compressible fluids around airfoils modeled by the Euler and Navier–Stokes equations, respectively, on block–structured curvilinear grid patches [8], backward–facing step on 2D triangulations [16] and simulation of a flame ball modeled by reaction–diffusion equations on 3D Cartesian grids [41,44]. These applications have been performed for compressible single-phase fluids. More recently, this concept has been extended to two-phase fluid flow of compressible gases, and applied to the investigation of non–stationary shock–bubble interactions on 2D Cartesian grids for the Euler equations [1,2,39]. By now, there is an increasing number of groups working on this subject: Postel et al. [18,19], Schneider et al. [42,43], Burger et al. [11,12], and Domingues et al. [21], Duarte et al. [22] and Koumoutsakos et al. [30]. For a more thorough discussion on adaptive multiresolution finite volume schemes we refer to the recent review [38] and the articles in [17]. Here we will confine ourselves to practical issues on the realization of the multiscale analysis and its parallelization.

The performance of the fully adaptive multiscale solver crucially depends on the data structures used for the implementation of the algorithms. In particular, appropriate data structures have to be designed such that the computational complexity in terms of memory and CPU time is proportional to the cardinality of the adaptive grid. For this purpose, the C++-template class library `igpm_t_lib` [40] has been developed. It has been recently extended with respect to non-Cartesian grid hierarchies, see [48]. In [10], a unified theory is established that shows under which conditions imposed on the grids and refinement rules a sparse data structure can be constructed, that efficiently manages adaptive multilevel grids consisting of triangular and quadrilateral cells (in 2D) and tetrahedral, cuboid and prismatic cells (in 3D) and also hybrid grids composed of different cell types. In view of an optimal memory management and a fast data access the well-known concept of *hash maps* is used. We refer to e.g. [20] for a general overview on hash maps and [9] for application-specific details. Since the multiscale-based grid adaptation requires sweeping through the different refinement levels, it turned out that hash maps are more suited for this purpose rather than tree structures. The latter need some overhead to access children, parents and neighbors in the tree, cf. [44]. Typically the work needed to access an element in the tree is proportional to $\log(N)$, where N is the number of nodes in the tree, whereas it is constant for hash maps, cf. [20]. Therefore the multiscale library, cf. [37], has been realized using hash maps. This library was successfully incorporated into the multi-block finite volume solver Quadflow [7,8].

Although multiscale-based grid adaptation leads to a significant reduction of the computational complexity (CPU time and memory) in comparison to computations on uniform meshes, this is not sufficient to perform 3D computations for complex geometries efficiently. In addition, we need parallelisation techniques in order to further reduce the computational time to an affordable order of magnitude. On a distributed memory architecture, the performance of a parallelized code crucially depends on the load-balancing and the interprocessor communication. Since the underlying adaptive grids are unstructured due to hanging nodes, this task cannot be considered trivial. For this purpose, graph partitioning methods are employed frequently using the Metis software [32,33] together with PETSc [3–5]. Opposite to this approach, we use space-filling curves, cf. [49]. Here the basic idea is to map level-dependent cell identifiers specifying the cells in a grid hierarchy of nested grids to a onedimensional line. The interval is then split into different parts each containing approximately the same number of entries. For this mapping procedure we employ the same cell identifiers as in case of the hash maps.

In [46] the quality of partitioning computed with different types of space-filling curves is compared to those generated with the graph partitioning package Metis. It turned out that Metis computes partitionings with lower edge-cuts than space-filling curves do. However, space-filling curves save both, a lot of time and a lot of memory. This is essential for our instationary applications, because we frequently need to rebalance the load. Due to the dynamics of the flow field, the grid has to be often updated in order to track the waves in the flow field appropriately.

The aim of the present work is to give a self-contained overview on (i) the construction of an appropriate multiscale analysis using biorthogonal wavelets, (ii) its efficient realization by means of global cell identifiers and hash maps and (iii) the parallelization of the multiresolution-based grid adaptation via MPI [25, 26] using space-filling curves. For this purpose, we first summarize in Section 2 the basic ingredients of the general multiresolution-based grid adaptation framework, namely, the multiscale decomposition and data compression. In Section 3 we then discuss details on the realization of a multiscale analysis. In particular, we present the construction of biorthogonal wavelets with higher vanishing moments on arbitrary nested grid hierarchies. When we consider parallelization, we have to care for load-balancing and interprocessor communication. This issue is addressed in Section 4. An optimal balancing of the load can be realized using space-filling curves. By means of the local multiscale transformation we discuss the data transfer at processor boundaries. Finally, in Section 5, we investigate the scalability of the parallelized multiscale-library. For this purpose, we employ the finite volume solver Quadflow, where the fully parallelized multiscale-library is embedded in the sense of a black box tool.

2. Multiresolution-based Grid Adaptation: Basic Ingredients

The key idea of adaptive multiresolution finite volume schemes [37, 38] is to apply data compression to the data at hand. For this purpose a multiscale analysis is performed for the mean values characterizing the flow field at some fixed time instant. In this section we summarize the basic ingredients how to successively decompose a sequence of cell averages given on a uniform fine grid (reference grid) into a sequence of coarse-scale averages and details. The details describe the update between two discretizations on successive resolution levels corresponding to a nested grid hierarchy. They reveal insight in the local regularity of the underlying function. In particular, they become negligibly small giving rise to data compression. From the remaining significant details, an adaptive grid, i.e., a locally refined grid with hanging nodes, is determined. This concept can be applied to any hierarchy of nested grids in one and more space dimensions, no matter whether these grids are structured or unstructured.

2.1. Multiscale Analysis and Grid Adaptation

Nested Grid Hierarchy. Let $\Omega_l := \{V_\lambda\}_{\lambda\in I_l}$ be a sequence of different meshes corresponding to different resolution levels $l \in \mathbf{N}_0$, where the mesh size decreases with increasing refinement level. Each grid Ω_l builds a partition of the computational domain $\Omega \subset \mathbf{R}^d$, i.e., $\Omega = \bigcup_{\lambda\in I_l} V_\lambda$. The cell identifier $\lambda \in I_l$ typically contains a level information and some some information on the local position, i.e., $\lambda = (level, position)$. For instance, in case of a Cartesian grid hierarchy it usually is given by the level l and a multiindex $\boldsymbol{k} = (k_1, \ldots, k_d) \in \mathbf{Z}^d$, i.e., $\lambda = (l, \boldsymbol{k})$. The grid hierarchy is assumed to be *nested*. This implies that each cell $\lambda \in I_l$ on level l is the union of cells $\mu \in \mathcal{M}_\lambda^0 \subset I_{l+1}$ on the next higher refinement level $l + 1$, i.e.,

$$V_\lambda = \bigcup_{\mu\in\mathcal{M}_\lambda^0\subset I_{l+1}} V_\mu, \quad \lambda \in I_l, \tag{1}$$

We call $\mathcal{M}_\lambda^0$ the refinement set that contains the indices of the child cells μ of the parent cell λ. Examples are shown in Figure 1 for a nested grid hierarchy of Cartesian (A) and triangular (C) meshes, respectively, and corresponding tree (B). Note that the framework presented here is not restricted to these configurations but can also be applied to *unstructured* grids and *irregular* grid refinements, cf. [37].

Multiscale Decomposition. On the grids Ω_l we introduce the sequences of cell averages $\hat{\boldsymbol{u}}_l := \{\hat{u}_\lambda\}_{\lambda\in I_l}$ corresponding to a scalar, integrable function $u \in L^1(\Omega, \mathbf{R})$. These are defined as L^2-inner product

$$\hat{u}_\lambda := \langle u, \phi_\lambda\rangle = \int_\Omega u(\boldsymbol{x})\, \phi_\lambda(\boldsymbol{x})\, d\boldsymbol{x}$$

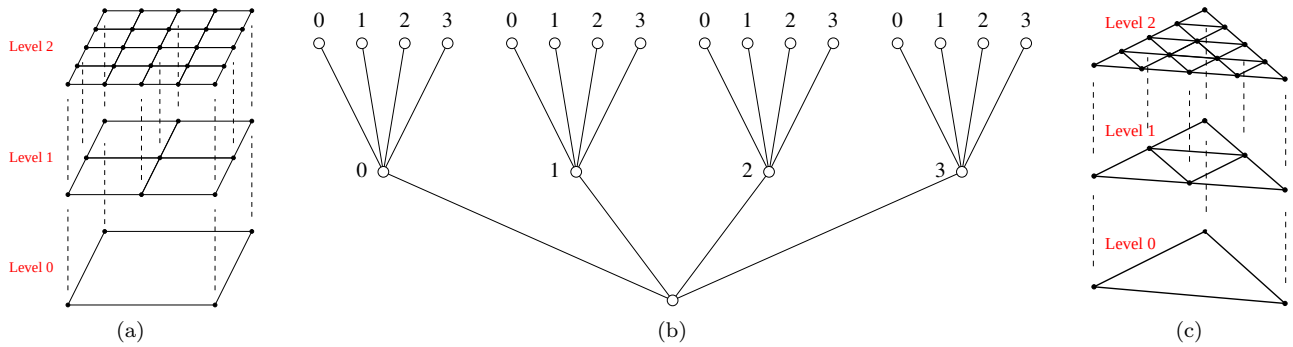


Figure 1: Hierarchy of nested Cartesian grids (A) and triangulations (C) as well as corresponding tree (B).

of u with the L^1–normalized box function

$$\phi_\lambda(\boldsymbol{x}) := \frac{1}{|V_\lambda|}\chi_{V_\lambda}(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \tag{2}$$

Here $|V_\lambda| := \int_{V_\lambda} 1\, d\boldsymbol{x}$ denotes the cell volume and χ_{V_λ} the characteristic function on V_λ, i.e., $\chi_{V_\lambda}(\boldsymbol{x}) = 1$ if $\boldsymbol{x} \in V_\lambda$ and zero elsewhere. Then the nestedness of the grids as well as the linearity of the integration operator imply the two–scale relation

$$\hat{u}_\lambda = \sum_{\mu \in \mathcal{M}_\lambda^0} m_{\mu,\lambda}^0\, \hat{u}_\mu, \quad m_{\mu,\lambda}^0 := \frac{|V_\mu|}{|V_\lambda|}, \quad \lambda \in I_l, \tag{3}$$

i.e., the coarse–grid average can be represented by a linear combination of the corresponding fine–grid averages. Consequently, the averages can be successively computed on coarser levels, starting on the finest level. Since information is lost by the averaging process, it is not possible to reverse (3). For this purpose, we have to store the update between two successive refinement levels by additional coefficients, so-called details. From the nestedness of the grid hierarchy we infer that the linear spaces $S_l := \operatorname{span}\{\phi_\lambda \ ; \ \lambda \in I_l\}$ are nested, i.e., $S_l \subset S_{l+1}$. Hence there exist complement spaces W_l such that $S_{l+1} = S_l \oplus W_l$. These are spanned by some basis, i.e., $W_l := \operatorname{span}\{\psi_\lambda \ ; \ \lambda \in J_l\}$ where $\#J_l = \#I_{l+1} - \#I_l$. Typically there are $M_\lambda := \#\mathcal{M}_\lambda^0 - 1$ different wavelets for each cell $\lambda \in I_l$. The different wavelet types are enumerated by an index $e \in E_\lambda^* := \{1, \ldots, M_\lambda\}$. Therefore the wavelet index is assumed to contain also the wavelet type in addition to the level and position information, i.e., $\mu = (\lambda, e) \in J_l$ with $\lambda \in I_l$. For our purposes, *biorthogonal wavelets* are preferable. Details on the construction of an appropriate wavelet basis will be given in Section 3.1. The basic idea is to complete the basis $\Phi_l := (\phi_\lambda)_{\lambda \in I_l}$ by a system of wavelet functions $\Psi_l := (\psi_\lambda)_{\lambda \in J_l}$ such that (i) they are locally supported, (ii) provide vanishing moments of a certain order and (iii) there exists a dual system $\tilde{\Phi}_l := (\tilde{\phi}_\lambda)_{\lambda \in I_l}$ and $\tilde{\Psi}_l := (\tilde{\psi}_\lambda)_{\lambda \in J_l}$ of dual functions satisfying the biorthogonality relations

$$\langle \phi_\lambda, \tilde{\phi}_\mu \rangle = \delta_{\lambda,\mu}, \quad \langle \phi_\lambda, \tilde{\psi}_\mu \rangle = 0, \quad \langle \psi_\lambda, \tilde{\psi}_\mu \rangle = \delta_{\lambda,\mu}, \quad \langle \psi_\lambda, \tilde{\phi}_\mu \rangle = 0. \tag{4}$$

The last requirement is typically the hardest to satisfy. It is closely related to the Riesz basis property of the infinite collection $\tilde{\Phi}_0 \cup \bigcup_{l=0}^\infty \tilde{\Psi}_l$ of $L_2(\Omega)$. For details we refer to the *concept of stable completions*, see [13].

In analogy to the cell averages, the details can be introduced as inner products of the function u with the *wavelet* ψ_λ. Since the box functions and the wavelets are linearly independent, there exists a two–scale relation

for the details, i.e.,

$$d_\lambda := \langle u, \psi_\lambda \rangle = \sum_{\mu \in \mathcal{M}_\lambda^1 \subset I_{l+1}} m_{\mu,\lambda}^1 \, \hat{u}_\mu, \quad \lambda \in J_l. \tag{5}$$

On the other hand, we deduce from the change of basis the existence of an inverse two–scale relation

$$\hat{u}_{l+1,\mu} = \sum_{\lambda \in \mathcal{G}_\mu^0 \subset I_l} g_{\lambda,\mu}^0 \, \hat{u}_\lambda + \sum_{\lambda \in \mathcal{G}_\mu^1 \subset J_l} g_{\lambda,\mu}^1 \, d_\lambda. \tag{6}$$

Note that the mask coefficients $m_{\mu,\lambda}^0$, $m_{\mu,\lambda}^1$ and $g_{\lambda,\mu}^0$, $g_{\lambda,\mu}^1$ in (3), (5) and (6) do not depend on the data but on geometric information only. In particular, by the biorthogonality relations (4) they can be determined as inner products

$$g_{\lambda,\mu}^0 = \langle \tilde{\phi}_\lambda, \phi_\mu \rangle, \quad \mu \in I_l, \ \lambda \in I_{l+1}, \qquad m_{\mu,\lambda}^0 = \langle \tilde{\phi}_\lambda, \phi_\mu \rangle, \quad \mu \in I_{l+1}, \ \lambda \in I_l,$$
$$g_{\lambda,\mu}^1 = \langle \tilde{\phi}_\lambda, \psi_\mu \rangle, \quad \mu \in J_l, \ \lambda \in I_{l+1}, \qquad m_{\mu,\lambda}^1 = \langle \tilde{\psi}_\lambda, \phi_\mu \rangle, \quad \mu \in I_{l+1}, \ \lambda \in J_l.$$

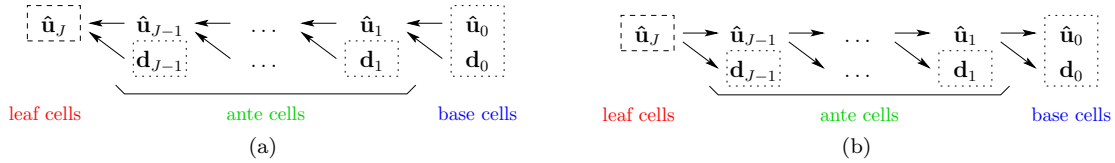


Figure 2: Pyramid scheme of multiscale (A) and inverse multiscale (B) transformation.

Cancellation Property. It can be shown that the details become small with increasing refinement level when the underlying function is smooth in the support of the wavelet ψ_λ, $\lambda \in J_l$, i.e.,

$$|d_\lambda| \leq C \, 2^{-l\,M} \, \|u^{(M)}\|_{L^\infty(V_\lambda)}. \tag{7}$$

More precisely, the details decay at a rate of at least $2^{-l\,M}$ provided that the function u is sufficiently differentiable and the wavelets have vanishing moments of order M, i.e.,

$$\langle p, \psi_\lambda \rangle = 0 \tag{8}$$

for all polynomials p of degree less than M. Here we assume that the grid hierarchy is quasi-uniform in the sense that the diameters of the cells on each level l are proportional to 2^{-l}. If coefficient and function norms behave essentially the same, as asserted by the Riesz basis property, (7) suggests to neglect all sufficiently small details in order to compress the original data. In fact, the higher M the more details may be discarded in smooth regions. The construction of appropriate wavelets is addressed below in Section 3.1.

Grid Adaptation. By means of the multiscale analysis a locally refined grid can be constructed. For this purpose, we first perform the multiscale decomposition successively applying (3) and (5), see Figure 2 (left). Since the details may become small, if the underlying function u is locally smooth, the basic idea is to perform data compression on the vector of details using hard thresholding, i.e., we discard all detail coefficients d_λ whose absolute values fall below a level-dependent threshold value $\varepsilon_l = 2^{(l-L)d}\varepsilon$ and keep only the *significant details* corresponding to the index set

$$\mathcal{D}_{L,\varepsilon} := \{\lambda \in J_l \ ; \ |d_\lambda| > \varepsilon_l, \ l \in \{0, \ldots, L-1\}\}.$$

Note that in case of a vector of conserved quantities it is reasonable to use different threshold values for each of these quantities, because they may differ by several orders of magnitude. In order to account for this jump in scales, we typically scale the details of a conserved quantity by the maximal absolute value of its cell averages in the entire flow domain. Then the threshold procedure is applied to the scaled details using a uniform threshold value ε.

In order to account for the dynamics of a flow field due to the time evolution and to appropriately resolve all physical effects on the new time level, this set is to be inflated such that the prediction set $\tilde{\mathcal{D}}_{L,\varepsilon} \supset \mathcal{D}_{L,\varepsilon}$ contains all significant details of the old *and* the new time level. The prediction strategy depends on the underlying system of evolution equations to be approximated. We will not discuss this issue here but refer to previous publications [15, 31, 37, 44].

In a last step, we construct the locally refined grid and corresponding cell averages. For this purpose, we proceed levelwise from coarse to fine, see Figure 2 (left), and we check for all cells of a level whether there exists a significant detail. If there is one, then we refine the respective cell, i.e., we replace the average of this cell by the averages of its children by locally applying the inverse multiscale transformation (6), see Figure 2 (right). The final grid is then determined by the index set $\tilde{\mathcal{G}}_{L,\varepsilon} \subset \bigcup_{l=0}^{L} I_l$ such that $\bigcup_{\lambda \in \tilde{\mathcal{G}}_{L,\varepsilon}} \overline{V}_\lambda = \Omega$ and $V_\lambda \cap V_\mu = \emptyset$ for all $\lambda \neq \mu$ with $\lambda, \mu \in \tilde{\mathcal{G}}_{L,\varepsilon}$. The grid adaptation procedure together with the resulting graded tree is illustrated in Figure 3.

Note that from a practical point of view it is useful to inflate the prediction set such that it corresponds to a tree. This allows to proceed levelwise in the multiscale transformation In particular, if the tree is graded then it is also guaranteed that there is at most one hanging node at a cell edge, see [37]. Details on performing the grading on arbitrary grid hierarchies are given in the next section.

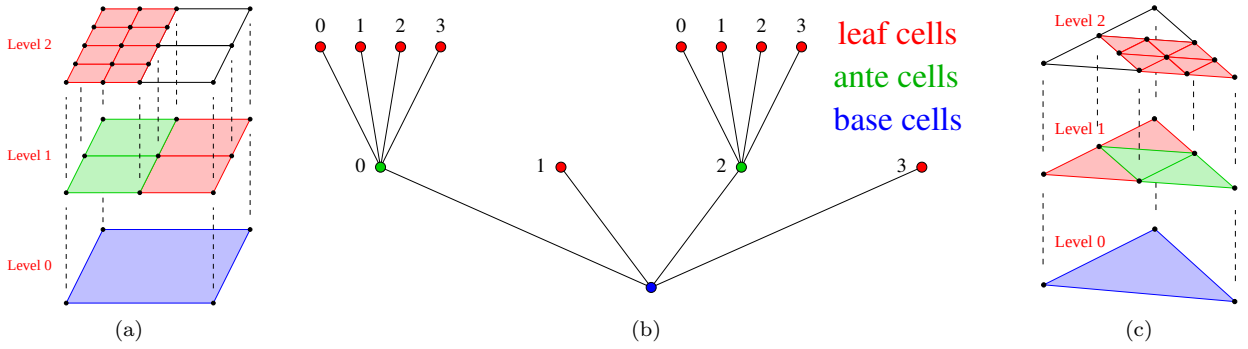


Figure 3: Grid adaptation procedure in case of Cartesian grids (A) and triangulations (C) as well as the corresponding graded tree (B).

2.2. **Algorithms**

In order to benefit from the reduced complexity corresponding to the cardinality of the set of significant details and the locally refined grid, respectively, all transformations have to be performed locally. In particular, we are not allowed to operate on the full arrays corresponding to the uniformly refined grids, i.e., the summation in the transformations (3), (5) and (6) have to be restricted to those indices which correspond to non–vanishing entries of the mask coefficients. Introducing the mask matrices $\boldsymbol{M}_{l,0} = (m^0_{\mu,\lambda})_{\mu \in I_{l+1}, \lambda \in I_l}$, $\boldsymbol{M}_{l,1} = (m^1_{\mu,\lambda})_{\mu \in I_{l+1}, \lambda \in J_l}$ and $\boldsymbol{G}_{l,0} = (g^0_{\mu,\lambda})_{\lambda \in I_l, \mu \in I_{l+1}}$, $\boldsymbol{G}_{l,1} = (g^1_{\mu,\lambda})_{\lambda \in J_l, \mu \in I_{l+1}}$, these two-scale relations may be rewritten in terms of matrix-vector products as $\boldsymbol{y}^T = \boldsymbol{x}^T \boldsymbol{A}$. Note that the mask matrices are sparse due to an appropriate choice of the wavelet basis. In order to perform the summation in the matrix-vector product only for the non-vanishing

entries of the matrices, the notion of the *support* of matrix columns and rows is helpful, i.e.,

$$\mathcal{A}_\lambda := \mathrm{supp}(\boldsymbol{A},\lambda) := \{\mu\ ;\ a_{\mu,\lambda}\neq 0\} = \text{ support of } \lambda\text{th column of } \boldsymbol{A},$$

$$\mathcal{A}^*_\lambda := \mathrm{supp}(\boldsymbol{A}^T,\lambda) := \{\mu\ ;\ a_{\lambda,\mu}\neq 0\} = \text{ support of } \lambda\text{th row of } \boldsymbol{A}.$$

The support $\mathcal{A}_\lambda$ of a column collects all non–vanishing matrix elements that might yield a non–trivial contribution to the λth component of the matrix–vector product, i.e., y_λ. Therefore $\mathcal{A}_\lambda$ can be interpreted as the *domain of dependence* for y_λ, i.e., the components x_μ which contribute to y_λ. The support $\mathcal{A}^*_\lambda$ of a row collects all non–vanishing matrix entries of the λth row that might yield a non–trivial contribution to the vector $\boldsymbol{y}$ of the matrix–vector product. Therefore $\mathcal{A}^*_\lambda$ can be interpreted as the *range of influence*, i.e., the components y_μ which are influenced by the component x_λ.

With this notation in mind, we now can give efficient algorithms for locally performing the decoding and encoding processes as they have been realized in the multiscale library:

Algorithm 1. *(Encoding) Proceed levelwise from $l = L-1$ downto 0:*

I. **Computation of cell averages on level** l*:*
   1. *For each active cell on level $l+1$ determine its parent cell on level l:*
      $U_l^0 := \bigcup_{\mu\in I_{l+1,\varepsilon}} \mathcal{M}^{*,0}_\mu$ *where* $I_{l+1,\varepsilon} := I_{l+1}\cap\tilde{\mathcal{G}}_{L,\varepsilon}$
   2. *Compute cell averages for parents on level l:*
      $\hat{u}_\lambda = \sum_{\mu\in\mathcal{M}^0_\lambda} m^0_{\mu,\lambda}\,\hat{u}_\mu,\ \lambda\in U_l^0$

II. **Computation of details on level** l*:*
   1. *For each active cell on level $l+1$ determine all cells on level l influencing their corresponding details:*
      $U_l^1 := \bigcup_{\mu\in I_{l+1,\varepsilon}} \mathcal{M}^{*,1}_\mu$
   2. *For each detail on level l determine the cell averages on level $l+1$ that are needed to compute the detail:*
      $P_{l+1} := \bigcup_{\lambda\in U_l^1} \mathcal{M}^1_\lambda\backslash I_{l+1,\varepsilon}$
   3. *Compute a prediction value for the cell averages on level $l+1$ not available in adaptive grid:*
      $\hat{u}_\lambda = \sum_{\mu\in\mathcal{G}^0_\lambda} g^0_{\mu,\lambda}\,\hat{u}_\mu,\ \lambda\in P_{l+1}$
   4. *Compute the details on level l:*
      $d_\lambda := \sum_{\mu\in\mathcal{M}^1_\lambda} m^1_{\mu,\lambda}\hat{u}_\mu,\ \lambda\in U_l^1$

Algorithm 2. *(Decoding) Proceed levelwise from $l=0$ to $L-1$:*

I. **Computation of cell averages on level** $l+1$*:*
   1. *Determine all cells on level $l+1$ that are influenced by a detail on level l:*
      $I^+_{l+1} := \bigcup_{\mu\in J_{l,\varepsilon}} \mathcal{G}^{*,1}_\mu$ *where* $J_{l,\varepsilon} := J_l\cap\tilde{\mathcal{D}}_{L,\varepsilon}$
   2. *Compute cell averages for cells on level $l+1$:*
      $$\hat{u}_\lambda = \sum_{\mu\in\mathcal{G}^0_\lambda} g^0_{\mu,\lambda}\,\hat{u}_\mu + \sum_{\mu\in\mathcal{G}^1_\lambda} g^1_{\mu,\lambda}\,d_\mu,\quad \lambda\in I^+_{l+1}$$

II. **Remove refined cells on level** l*:*
   1. *For each cell on level $l+1$ determine its parent cell on level l:*
      $I_l^- := \bigcup_{\lambda\in I^+_{l+1}} \mathcal{M}^{*,0}_\lambda$
   2. *Remove the parent cells on level l from the adaptive grid:*
      delete $\hat{u}_\lambda,\ \lambda\in I_l^-$*, i.e.,* $I_{l,\varepsilon} := I_l^+/I_l^-$ *(Note:* $I_0^+ = I_0$*)*

Grading. In order to realize the local multiscale transformation in *one* sweep through the refinement levels and in view of its feasibility, i.e., the data $\hat{u}_\lambda$, d_λ on the right-hand sides of the calculations in Algorithm 1 and 2 are accessible, we have to inflate the set of significant details $\mathcal{D}_{L,\varepsilon}$ by a grading procedure. For this purpose, we first agglomerate all wavelet indices corresponding to a cell $V_\lambda,\ \lambda\in I_l$, i.e., $T_\lambda := \{(\lambda,e)\ ;\ e\in E^*_\lambda\}$, and

introduce the stencil of neighbors $\mathcal{N}_\lambda \subset I_l$ corresponding to V_λ by the recursive definition

$$\mathcal{N}_\lambda^0 := \{\lambda\},\ \mathcal{N}_\lambda^i := \{\mu \in I_l\ :\ \exists\, \nu \in \mathcal{N}_\lambda^{i-1} \text{ s.t. } \overline{V}_\mu \cap \overline{V}_\nu \neq \emptyset\},\ i = 1, \ldots, q. \tag{9}$$

Then $\mathcal{N}_\lambda^q$ is referred to as the *neighborhood of degree* $q \in \mathbf{N}_0$. This set is the union of neighbor cells corresponding to V_λ which are attached to each other by at least one point. Hence the support $\bigcup_{\nu \in \mathcal{N}_\lambda^q} \overline{V}_\nu$ is a simply connected domain with no holes. In Figure 4 the sets $\mathcal{N}_\lambda^q$ are illustrated for two–dimensional grids composed of quadrilaterals and triangles, respectively. Here the cell V_λ is indicated by 0. The neighborhood of degree q is determined by all cells labeled with the numbers $q, q-1, \ldots, 0$.

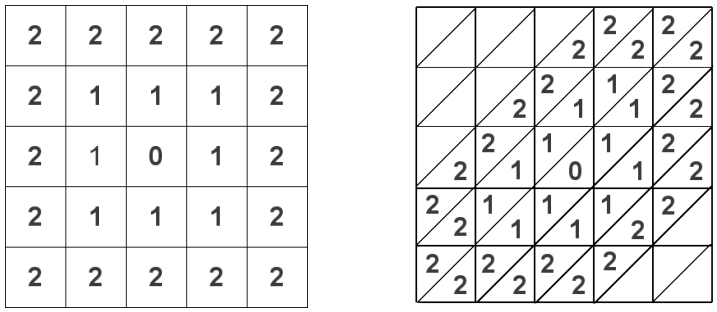


Figure 4: Illustration of the sets $\mathcal{N}_\lambda^q$ for $q = 0, 1, 2$: Cartesian grid (A) and triangulation (B).

The sets $\mathcal{M}_\lambda^0 \subset I_{l+1}$ characterize the indices of the child cells generated by refining the cell V_λ, $\lambda \in I_l$. The inverse relation, i.e., we are interested in the index $\lambda \in I_l$ of the parent cell corresponding to a cell $\mu \in I_{l+1}$ is provided by the operators $\pi_{l+1} : I_{l+1} \to I_l,\ l = 0, \ldots, L-1$, defined by $\pi_{l+1}(\mu) = \lambda$ for $\mu \in \mathcal{M}_\lambda^0$. Due to the nestedness of the grid hierarchy we conclude that for any $\mu \in I_{l+1}$ there is exactly one index $\lambda \in I_l$ such that $\mu \in \mathcal{M}_\lambda^0$.

By means of the above settings we can now introduce the notion of a graded tree: the set of significant details $\mathcal{D}_{L,\varepsilon}$ is called a *graded tree of degree* q, if for any $l \in \{1, \ldots, L\}$ the relation

$$\nu = (\lambda, e) \in \mathcal{D}_{L,\varepsilon},\ \lambda \in I_l,\ e \in E_\lambda^* \Rightarrow T_\mu \subset \mathcal{D}_{L,\varepsilon}, \quad \forall\, \mu \in \mathcal{N}_{\pi_l(\lambda)}^q \subset I_{l-1} \tag{10}$$

holds. The grading of the truncated details results in a tree structure of the details. Generally speaking, a graded tree means that for any significant detail on level l there are significant details in the neighborhood but on the next lower level $l-1$. Thus, gradedness simplifies the local multiscale transformation, since there are no isolated details on higher levels. In addition, locally switching between finer and coarser levels becomes easier, because only two levels are involved if the grading parameter q is chosen sufficiently large, i.e., the support of the wavelet functions corresponding to a cell V_λ, $\lambda \in I_l$, must be contained in the set $\mathcal{N}_\lambda^q$. In particular, the level difference between two neighbor cells is at most 1. Hence, we have to look for all neighbors only on the same level, the next coarser level or the next finer level, i.e., we avoid recursive searching in the tree. We emphasize that for $q = 0$ the details still correspond to a tree but not necessarily a graded tree.

Finally, we present the algorithm that realizes the grading of the set of significant details $\mathcal{D}_{L,\varepsilon}$:

Algorithm 3. *(Grading) Proceed levelwise from $l = L-1$ downto 0:*

1. *Initialize the grading set* $J_{l-1,\varepsilon}^+ := \emptyset$
2. *Collect the neighborhoods* $\mathcal{N}_{\pi_l(\lambda)}^q \subset I_{l-1}$ *for all significant details* $(\lambda, e) \in \mathcal{D}_{L,\varepsilon}$, $\lambda \in I_l$, $e \in E_\lambda^*$*:*
   $J_{l-1,\varepsilon}^+ := J_{l-1,\varepsilon}^+ \cup (\mathcal{N}_{\pi_l(\lambda)}^q \times E_{\pi_l(\lambda)}^*)$

3. *Set the corresponding details to zero if not yet available:*
   $d_\mu := 0$ *for* $\mu \in J^+_{l-1,\varepsilon} \backslash \mathcal{D}_{L,\varepsilon}$
4. *Add the grading set to the original set* $\mathcal{D}_{L,\varepsilon}$ *on the lower level* $l-1$*:*
   $\mathcal{D}_{L,\varepsilon} := \mathcal{D}_{L,\varepsilon} \cup J^+_{l-1,\varepsilon}$

3. Realization of a Multiresolution Analysis

In Section 2 we have briefly summarized the general framework of multiscale-based grid adaptation using (biorthogonal) wavelets. We now address its realization and implementation. There are basically two issues: (i) the construction of an appropriate MRA on a given hierarchy of nested grids and (ii) the design of suited cell identifiers and data structures that allow for an optimal performance of the Algorithms 1 and 2, i.e., the computational effort both in terms of CPU time and memory is proportional to the cardinality of the locally refined grid.

3.1. Construction of wavelets

For the construction of an appropriate MRA we proceed in two steps. First of all, we construct so-called box wavelets that can be considered generalized Haar wavelets on arbitrary grid hierarchies. Since these wavelets provide only one vanishing moment, we modify these functions in a second step.

Box wavelets. For the construction the so-called box wavelets are introduced as a linear combination of the fine–scale box functions $\phi_\mu,\ \mu \in \mathcal{M}^0_\lambda \subset I_{l+1}$, related to the refinement of the cell V_λ, $\lambda \in I_l$,

$$\psi^B_{\lambda,e} := \sum_{\mu \in \mathcal{M}^0_\lambda} a^\lambda_{0,\mu} a^\lambda_{e,\mu}\, \phi_\mu, \quad e \in E^*_\lambda. \tag{11}$$

Here the coefficients are determined such that the vectors $\boldsymbol{a}^\lambda_e := (a^\lambda_{e,\mu})_{\mu\in\mathcal{M}^0_\lambda}$, $e \in E^*_\lambda$, form an orthonormal system to the vector $\boldsymbol{a}^\lambda_0 := \left(\sqrt{|V_\mu|/|V_\lambda|}\right)_{\mu\in\mathcal{M}^0_\lambda}$. Then the system $\{\phi_\lambda, \psi^B_{\lambda,1}, \ldots, \psi^B_{\lambda,M_\lambda}\}$ of L^1-normalized functions is orthogonal to the system of L^∞-scaled functions $\{\tilde\phi_\lambda, \tilde\psi^B_{\lambda,1}, \ldots, \tilde\psi^B_{\lambda,M_\lambda}\}$ defined by $\tilde\phi_\lambda := |V_\lambda|\,\phi_\lambda$ and $\tilde\psi^B_{\lambda,e} := |V_\lambda|\,\psi^B_{\lambda,e}$. Obviously, the above design principles hold by construction, where the support of the box wavelets is V_λ and the order of vanishing moments is $M = 1$. In particular, we deduce from the orthogonality of the parameter vectors

$$\phi_\mu = \phi_\lambda + \sum_{e\in E^*} \frac{a^\lambda_{e,\mu}}{a^\lambda_{0,\mu}}\, \psi^B_{\lambda,e}, \quad \mu \in \mathcal{M}^0_\lambda. \tag{12}$$

Then the mask coefficients $m^1_{\mu,\lambda}$, $m^1_{\mu,\lambda}$ and $g^0_{\lambda,\mu}$, $g^1_{\lambda,\mu}$ can be deduced from the two-scale relations (3), (5) and (6), respectively. For this purpose we employ that the cell averages and the details are inner products of the box function and the box wavelet with some function, i.e., $\hat u_\lambda = \langle u, \phi_\lambda\rangle$ and $d_{\lambda,e} = \langle u, \psi^B_{\lambda,e}\rangle$. Finally we obtain

$$\check m^0_{\mu,\lambda} = a^\lambda_{0,\mu} a^\lambda_{0,\mu},\ \lambda \in I_l,\ \mu \in I_{l+1} \qquad \check m^1_{\mu,(\lambda,e)} = a^\lambda_{e,\mu} a^\lambda_{0,\mu},\ (\lambda,e) \in J_l,\ \mu \in I_{l+1}, \tag{13}$$

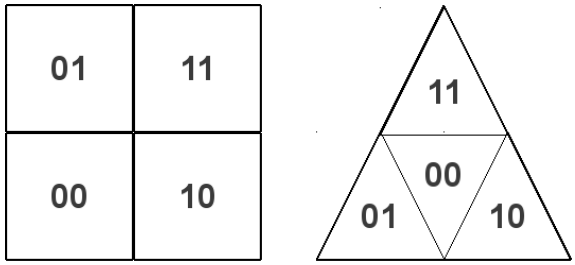
$$\check g^0_{\lambda,\mu} = a^\lambda_{0,\mu}/a^\lambda_{0,\mu},\ \lambda \in I_l,\ \mu \in I_{l+1} \qquad \check g^1_{(\lambda,e),\mu} = a^\lambda_{e,\mu}/a^\lambda_{0,\mu},\ (\lambda,e) \in J_l,\ \mu \in I_{l+1}. \tag{14}$$

Here we use the check to distinguish the mask coefficients with respect to the box wavelets from its modification introduced below.

Example: As an example we consider the Cartesian grid hierarchy. In this case the cell identifiers are determined by multiindices, i.e., $\lambda = (l, \boldsymbol{k})$ and $\mu = (l+1, 2\boldsymbol{k} + \boldsymbol{i})$ with $\boldsymbol{i} \in E := \{0,1\}^d$. In this case, the coefficients $a^\lambda_{\boldsymbol{e},\mu}$ can be explicitly determined by

$$a^\lambda_{\boldsymbol{0},\mu} = \sqrt{|V_\mu|/|V_\lambda|}, \quad a^\lambda_{\boldsymbol{e},\mu} = (-1)^{\boldsymbol{e}\cdot\boldsymbol{i}} a^\lambda_{\boldsymbol{0},\mu}, \quad \boldsymbol{e} \in E^* := E \setminus \{\boldsymbol{0}\}.$$

A proof can be found in [24], Lemma 3.2, p. 77. In the twodimensional case ($d = 2$) the box function and the box wavelets are shown in Figure 6. Note that we can use the same coefficients in case of regular 2D-triangulations, because there are also four children and the volume is the same for all of them as illustrated in Figure 5.



(a) (b)

Figure 5: Enumeration of wavelet types and correlation to subcells in case of a square (A) and a regular triangle (B).

Modified box wavelets with higher vanishing moments. Since we want to have sparse grids, we need wavelets with higher vanishing moments that allow for higher compression rates. For this purpose, the box wavelets are not convenient. By means of a change of stable completion, cf. [13], we may deduce another MRA, where we introduce additional parameters of freedom by adding coarse-scale box functions to the box wavelets, i.e.,

$$\psi^M_{\lambda,e} = \psi^B_{\lambda,e} + \sum_{\nu \in \mathcal{L}_\lambda} l^\nu_{\lambda,e}\, \phi_\nu,\ e \in E^*. \tag{15}$$

Here the index set $\mathcal{L}_\lambda \subset I_l$ denotes a neighborhood of the cell V_λ, $\lambda \in I_l$. Then the parameters $l^\nu_{\lambda,e}$ can be determined by solving the linear system of equations that can be derived from the cancellation property (8) putting $p = \psi^M_{\lambda,e}$. Note that this system in general will be under-determined in the multivariate case. The details of the construction can be found in [37]. Here we only want to remark that typically the dual wavelets ψ^M_λ are piecewise constant functions with vanishing moments of order M, whereas the primal wavelets $\tilde{\psi}^M_\lambda$ in general are not known explicitly. However, they have some regularity, i.e., they are in some Hölder space C^s with $s \le M$, cf. [14]. Moreover, we note that because of (2) the basis functions of the primal and dual system are normalized with respect to the L^∞- and L^1-metric, respectively.

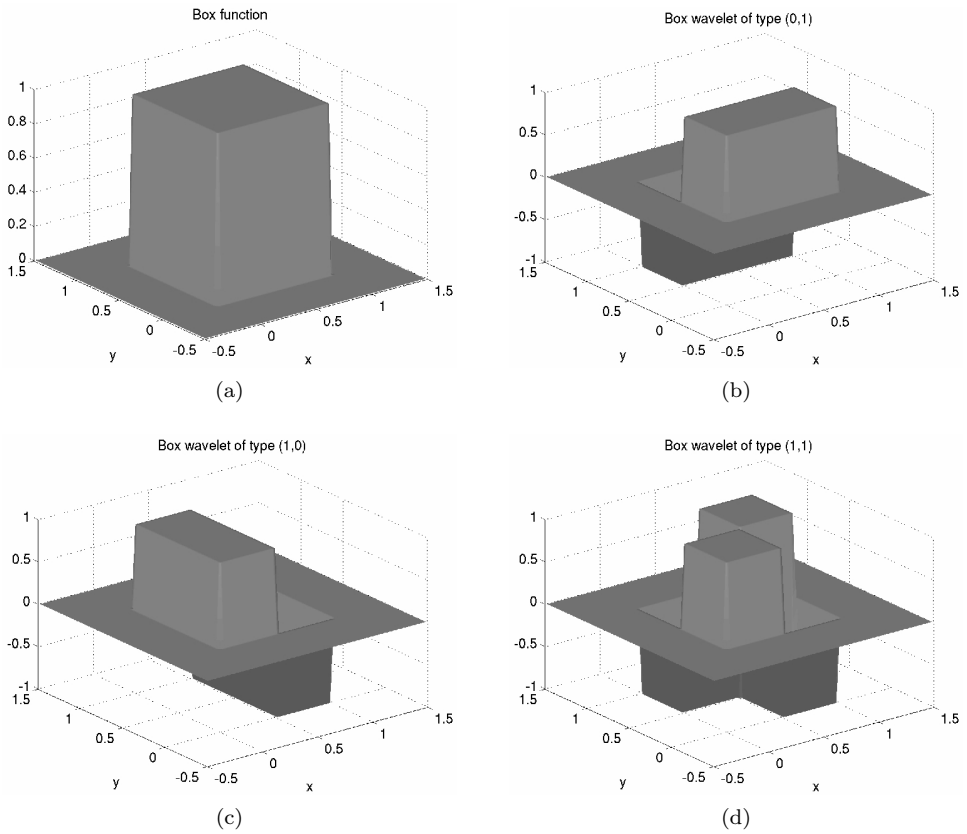
Again we can compute the corresponding mask matrices. For this purpose, we follow the concept of changing a stable completion according to [13]. By this concept the mask coefficients turn out to be modifications of the coefficients (13) and (14) corresponding to the box wavelets. In particular, we obtain

$$m^0_{\mu,\lambda} = \check{m}^0_{\mu,\lambda},\ \lambda \in I_l,\ \mu \in I_{l+1} \qquad m^1_{\mu,(\lambda,e)} = \check{m}^1_{\mu,(\lambda,e)} + l^\nu_{\lambda,e}\check{m}^0_{\mu,\nu},\ (\lambda,e) \in J_l,\ \mu \in I_{l+1}, \tag{16}$$

$$g^0_{\lambda,\mu} = \check{g}^0_{\lambda,\mu} - \sum_{e \in E^*} l^\lambda_{\nu,e}\check{g}^1_{(\nu,e),\mu},\ \lambda \in I_l,\ \mu \in I_{l+1} \qquad g^1_{(\lambda,e),\mu} = \check{g}^1_{(\lambda,e),\mu},\ (\lambda,e) \in J_l,\ \mu \in I_{l+1}. \tag{17}$$

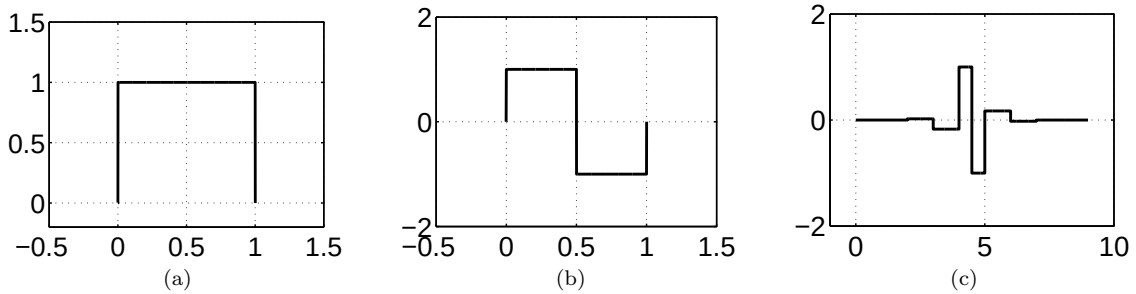
Here the index $\nu \in I_l$ is uniquely determined by $\nu = \pi_{l+1}(\mu)$, i.e., $\mu \in \mathcal{M}^0_\nu$, see Section 2.2. Details on the computation of the mask coefficients can be found in [37], Sections 2.3.2 and 2.5.1.

Example: For an example we first consider the case of a 1D dyadic grid hierarchy on the real line, i.e., $\Omega = \mathbf{R}$. Let $\mathcal{L}_{(l,k)} = \{(l, k-s), \dots (l, k+s)\}$ be the extended support of the modified box wavelets. Here $s \in \mathbf{N}$



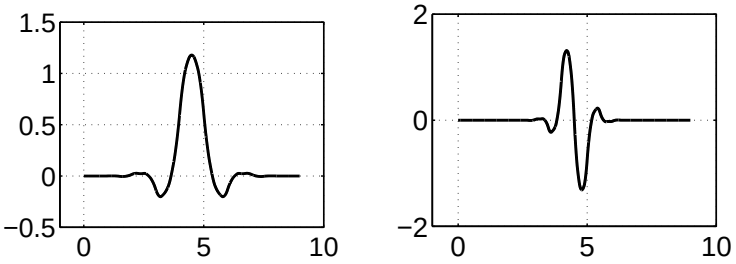
(a) (b) (c) (d)

Figure 6: Box function ϕ (subfigure A) and box wavelets $\psi^B_{\boldsymbol{e}}$ (subfigures B-D) where all functions are defined on a reference element $[0,1]^2$.



(a) (b) (c)

Figure 7: Box function ϕ (subfigure A), Haar wavelet ψ^B (subfigure B), and modified Haar wavelet ψ^M with $M = 5$ vanishing moments (subfigure C), where all functions are defined on a reference element.



(a) (b)

Figure 8: Primal scaling function $\tilde{\phi}^M$ (subfigure A) and primal wavelet $\tilde{\psi}^M$ (subfigure B) corresponding to the modified Haar wavelet ψ^M with $M = 5$ vanishing moments, where all functions are defined on a reference element.

is some fixed positive number. The coefficients $l^{\nu}_{\lambda,e}$ can be uniquely determined solving the linear system

$$\sum_{r=-s}^{s} 2^{i+r} \left((r+0.5)^{i+1} - (r-0.5)^{i+1} \right) l_r = 1 - (-1)^i, \quad i = 0, \ldots, M-1,$$

resulting from the vanishing moments conditions (8) with $M = 2s + 1$. Note that these coefficients do not dependent on the level and the position, i.e., $l^{\nu}_{\lambda,e} = l^{(l,r)}_{(l,k),1} = l_{r-k}$, $r \in \{k - s, \ldots, k + s\}$. In particular, we obtain for $s = 1$: $l_{-1} = -1/8$, $l_0 = 0$, $l_1 = 1/8$ and for $s = 2$: $l_{-2} = 3/128$, $l_{-1} = -11/64$, $l_0 = 0$, $l_1 = 11/64$, $l_2 = -3/128$. In Figures 7 and 8 we present the primal and dual functions of the resulting MRA in case of $s = 2$. Note that in case of $s = 0$ the box wavelet coincides with the well-known Haar wavelet. In order to deal with bounded domains $\Omega = [a, b]$, we can apply the same strategy using one-sided stencils $\mathcal{L}_\lambda$. For details we refer to [37].

For multidimensional Cartesian grid hierarchies then the MRA can be constructed using tensor products of these functions instead of applying (15). For this purpose, we introduce the convention $\psi_{l,k,0} := \phi_{l,k}$ and $\psi_{l,k,1} = \psi_{l,k}$ where $\phi_{l,k}$ and $\psi_{l,k}$ are the univariate box function and the univariate box wavelet (Haar wavelet), respectively. Then the multivariate analogue is determined by

$$\phi_{l,\boldsymbol{k}}(\boldsymbol{x}) := \prod_{i=1}^{d} \psi_{l,k_i,0}(x_i), \qquad \psi_{l,\boldsymbol{k},\boldsymbol{e}}(\boldsymbol{x}) := \prod_{i=1}^{d} \psi_{l,k_i,e_i}(x_i), \tag{18}$$

where $\boldsymbol{e} \in E^* := \{0, 1\}^d \backslash \{\boldsymbol{0}\}$ denotes the different wavelet types corresponding to the cell $V_{l,\boldsymbol{k}}$.

For the Algorithms 1 and 2 realizing the local multiscale setting, see Section 2.2, we introduced the supports of the columns and rows corresponding to the mask matrices. In case of Cartesian and curvilinear grid hierarchies these can be determined easily in terms of multiindices. In particular, for the box function and the modified

box wavelets the supports are given by

$$\begin{aligned}
\mathcal{M}^0_{l,\boldsymbol{k}} &= \{(l+1, 2\boldsymbol{k}+\boldsymbol{i})\ ;\ \boldsymbol{i}\in E\} \subset I_{l+1},\ (l,\boldsymbol{k})\in I_l,\\
\mathcal{M}^1_{l,\boldsymbol{k},\boldsymbol{e}} &= \{(l+1, 2\boldsymbol{k}+\boldsymbol{r})\ ;\ \boldsymbol{r}\in\{-2s,\dots,2s+1\}^d\} \subset I_{l+1},\ (l,\boldsymbol{k},\boldsymbol{e})\in J_l,\\
\mathcal{G}^0_{l+1,\boldsymbol{k}} &= \{(l, \lfloor\boldsymbol{k}/2\rfloor+\boldsymbol{r})\ ;\ \boldsymbol{r}\in\{-s,\dots,s\}^d\} \subset I_l,\ (l+1,\boldsymbol{k})\in I_{l+1},\\
\mathcal{G}^1_{l+1,\boldsymbol{k}} &= \{(l, \lfloor\boldsymbol{k}/2\rfloor, \boldsymbol{e})\ ;\ \boldsymbol{e}\in E^*\} \subset J_l,\ (l+1,\boldsymbol{k})\in I_{l+1},\\
\mathcal{M}^{*,0}_{l+1,\boldsymbol{k}} &= \{(l, \lfloor\boldsymbol{k}/2\rfloor)\} \subset I_l,\ (l+1,\boldsymbol{k})\in I_{l+1},\\
\mathcal{M}^{*,1}_{l+1,\boldsymbol{k}} &= \{(l, \lfloor\boldsymbol{k}/2\rfloor+\boldsymbol{r}, \boldsymbol{e})\ ;\ \boldsymbol{r}\in\{-s,\dots,s\}^d,\ \boldsymbol{e}\in E^*\} \subset J_l,\ (l+1,\boldsymbol{k})\in I_{l+1},\\
\mathcal{G}^{*,0}_{l,\boldsymbol{k}} &= \{(l+1, 2\boldsymbol{k}+\boldsymbol{r})\ ;\ \boldsymbol{r}\in\{-2s,\dots,2s+1\}^d\} \subset I_{l+1},\ (l,\boldsymbol{k})\in I_l,\\
\mathcal{G}^{*,1}_{l,\boldsymbol{k},\boldsymbol{e}} &= \{(l+1, 2\boldsymbol{k}+\boldsymbol{i})\ ;\ \boldsymbol{i}\in E\} \subset I_{l+1},\ (l,\boldsymbol{k},\boldsymbol{e})\in J_l.
\end{aligned}$$

Here $\lfloor\boldsymbol{k}/2\rfloor$ denotes integer devision by 2 in each of the components of the multiindex $\boldsymbol{k}$. Note that we assume an infinite domain $\Omega = \mathbf{R}^d$. In case of bounded domains, we have to use one-sided stencils near to the boundaries accounting for the supports of the wavelets fully contained inside the domain. Details can be found [37], Section 3.8. In order to perform the Algorithms 1 and 2 we need to choose the grading parameter $q \geq s$ in Algorithm 3.

In case of triangulations, the supports cannot be described in such a compact form. Instead we have to apply the neighboring algorithms provided in [10] to compute the sets during runtime.

3.2. **Cell identifiers and data structures**

In order to realize the Algorithms 1 and 2 the cells in the adaptive grid have to be addressed. There are typically two approaches for managing grids with face-centered connectivity used in standard flow solver codes.
Lists. In some classical codes, multilevel grids are treated as fully unstructured and the multilevel structure is neglected. This leads to a maximum flexibility of the code as a very wide range of grid types can be processed. The cells and therefore the degrees of freedom are numbered in an arbitrary way, e.g., in the order of creation of the corresponding cells. So there is no global a-priori ordering of the cells. The connectivities of the cells is maintained in lists that state which cells share a common face.

In these approaches usually long vectors are used as underlying data structure to store the corresponding cell data, e.g., cell averages or coefficients of ansatz functions. This is a very rigid data structure with respect to adaptation, because the data in the vector has to be stored in a contiguous way. When the grid is modified, the number of coefficients and also the vector length changes. As some cells are removed, there are segments in the vector that are no longer needed and abandoned. At the same time new segments are required in the vector as some cells are refinement. The management of the cell data therefore induces an extra overhead, e.g., for the administration of a list of free segments in the long vector.
Pointers. A different approach that is also very common exploits the structure of the multilevel grid. It uses pointers to represent the cell connectivity: each cell contains a pointer to the spatially adjacent cells. In a multilevel code, each cell carries furthermore a pointer to its parent and to each of its child cells (hierarchical connectivity). As usually a cell is divided into two, four or eight subcells, this leads to a binary tree, quadtree or octree data structure.

This approach offers very fast access to the neighbor cell via the pointer, but a lot of effort is necessary to keep all pointers updated when the grid is adapted. Another drawback of this data structure is the relatively big storage overhead for the pointers representing the connectivity. On current computers with 64-bit architecture, a pointer and a double precision floating point number according to the IEEE 754 standard both need 64 bits of memory. In a 2D implementation, a quadrilateral cell, that can be refined into four subcells, has to store at least 5 pointers for hierarchical (one to parent, 4 to child cells) and 4 pointers for spatial connectivity. For a tetrahedral cell in 3D that is subdivided in 8 subtetrahedra, in the same situation at least 13 pointers are needed.

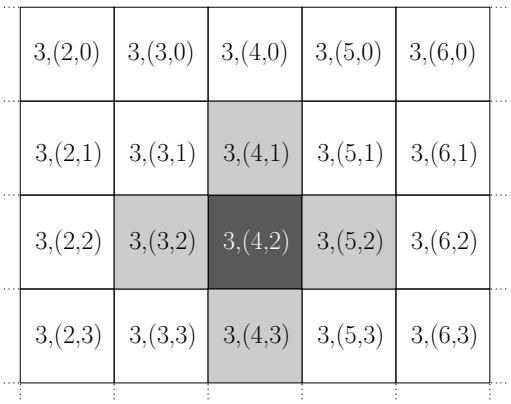
Cell identifiers. In both approaches we need additional memory and there is no direct access to the data of these cells. Alternatively we suggest to use hash maps instead, see Section 3.2. For this purpose we need some global identifiers of the cells in the grid hierarchy that do not change during the computation. Therefore we introduce for each cell in the grid hierarchy a unique *cell identifier*. However, the identifiers must not be stored for all cells of the grid hierarchy but only for the active cells in the adaptive grid. Since these will be changing with each grid adaptation, the identifiers must be easy to construct and, in addition, must provide easy access to the computation of identifiers corresponding to the neighbors, parents and children of a cell. In the following we give two examples for Cartesian and triangular grid hierarchies.

Example: Grid hierarchy of Cartesian grids A nested grid hierarchy is defined by means of a sequence of nested uniform partitions of the domain $\Omega = [0,1]^d$, see Figure 1. To this end, we introduce the sets of multiindices $I_l := \prod_{i=1}^d \{0,\dots,N_{l,i}-1\} \subset \mathbf{N}_0^d$, $l = 0,\dots,L$, with $N_{l,i} = 2\,N_{l-1,i}$ initialized by some $N_{0,i}$. Here l represents the refinement level where the coarsest partition is indicated by 0 and the finest by L. The product denotes the Cartesian product, i.e., $\prod_{i=1}^d A_i := A_1 \times \cdots \times A_d$. Then a nested sequence of grids $\mathcal{G}_l := \{V_{l,k}\}_{\boldsymbol{k}\in I_l}$, $l = 0,\dots,L$, is determined by $V_{l,\boldsymbol{k}} := \prod_{i=1}^d [k_i\, h_{l,i}, (k_i+1)\, h_{l,i}]$, with $h_{l,i} := 1/N_{l,i} = h_{l-1,i}/2$, see Figure 1.

Obviously, each grid $\mathcal{G}_l$ builds a partition of Ω, i.e., $\Omega = \bigcup_{(l,\boldsymbol{k})\in I_l} V_{l,\boldsymbol{k}}$, and the cells of two neighboring levels are nested, i.e., $V_{l,\boldsymbol{k}} = \bigcup_{(l+1,\boldsymbol{r})\in\mathcal{M}^0_{l,\boldsymbol{k}}} V_{l+1,\boldsymbol{r}}$, $(l,\boldsymbol{k}) \in I_l$. Because of the *dyadic* refinement, the refinement set is determined by $\mathcal{M}^0_{l,\boldsymbol{k}} = \{(l+1, 2\boldsymbol{k}+\boldsymbol{i})\ ;\ \boldsymbol{i} \in E := \{0,1\}^d\} \subset I_{l+1}$ of 2^d cells on level $l+1$ resulting from the subdivision of the cell $V_{l,\boldsymbol{k}}$.

Note that by the nested Cartesian grid hierarchy it is straightforward to construct a hierarchy of curvilinear grids by means of a grid mapping $\boldsymbol{x} : R := [0,1]^d \to \Omega$. See [34], [37] for more details.

A cut of a Cartesian grid and its cell identifiers, together with the hierarchical and neighborhood relations can be seen in Figure 9.



- parent: $ID_p = (\ell-1, (\lfloor \frac{k_x}{2} \rfloor, \lfloor \frac{k_y}{2} \rfloor))$
- children:
  $ID_{c_0} = (\ell+1, (2k_x, 2k_y))$,
  $ID_{c_1} = (\ell+1, (2k_x, 2k_y+1))$,
  $ID_{c_2} = (\ell+1, (2k_x+1, 2k_y))$,
  $ID_{c_3} = (\ell+1, (2k_x+1, 2k_y+1))$
- neighbors of same level:
  $ID_{n_N} = (\ell, (k_x, k_y-1))$,
  $ID_{n_S} = (\ell, (k_x, k_y+1))$,
  $ID_{n_E} = (\ell, (k_x+1, k_y))$,
  $ID_{n_W} = (\ell, (k_x-1, k_y))$

Figure 9: Cell identifiers for a Cartesian mesh.

Example: Grid hierarchy of triangulations The construction of a grid hierarchy can be extended to non-Cartesian grids by providing an appropriate refinement rule that is recursively applied. The refinement rule defines how to divide a given cell into its child cells, cf. [10]. Thus, each cell that may occur within the grid hierarchy is characterized by a cell identifier, which will also provide the hierarchical connectivity. Each cell is associated with the *path* of child numbers we have to pass through from level to level to reach that cell

$$path = c_l \cdots c_3 c_2 c_1, \quad c_i \in \{0,1,2,3\}, \quad i = 1,\dots,L.$$

Reading from right to left, the path gives the children passed through from coarser to finer levels. Hence, each cell within one base cell is uniquely identified by its level l and a sequence of path digits. As detailed in [10], from this information the connectivity can be established and this concept can be extended to hybrid grids of base cells. Figure 10 shows the child numbering and the paths associated with triangles for three levels of refinement. For more details we refer to [10].

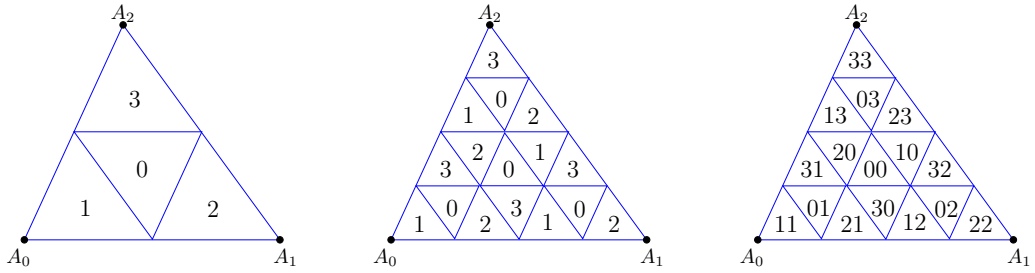


Figure 10: Child numbering on two refinement levels (A and B) and the corresponding paths on level two (C) (Copyright SIAM J. Sci. Comput. [10]).

Data structures. In order to realize the reduced algorithmical complexity, the data structures have to be designed such that the computational complexity (memory and CPU time) is proportional to the cardinality of the adaptive grid. In view of such an optimal memory management and a fast data access we use the well-known concept of *hash maps*, cf. [20], that is composed of two parts, namely, a vector of pointers, a so–called *hash table*, and a memory heap, see Figure 11. The hash table is connected to a *hash function* $f : \mathcal{U} \to \mathcal{T}$, which maps a *key*, here λ, to a row in the hash table of length $\#\mathcal{T}$, i.e., a number between 0 and $\#\mathcal{T} - 1$. Here the set $\mathcal{U}$ can be identified with all possible cells in the nested grid hierarchy (universe of keys), i.e., $\mathcal{U} = \{\lambda : \lambda \in I_l,\ l = 0, \dots, L\}$, and $\mathcal{T}$ corresponds to the keys of the dynamically changing adaptive grid, i.e., $\lambda \in \tilde{\mathcal{G}}_{L,\varepsilon}$.

The set of all possible keys is much larger than the length of the hash table, i.e., $\#\,\mathcal{T} \ll \#\mathcal{U}$. Hence, the hash function cannot be injective. This leads to collisions in the hash table, i.e., different keys might be mapped to the same position by the hash function. As collision resolution we choose chaining: the corresponding values of these keys are linked to the list that starts at position $f(key)$. Each element in the hash table is a pointer to a linked list whose elements are stored in the heap. Here each element of the list can be a complex data structure itself. It contains the key and usually additional data, the so–called *value*. In general, the value consists of the data corresponding to a cell.

The performance of the hash map crucially depends on the number of collisions. In order to optimize the number of collisions, the length of the hash table $\#\mathcal{U}$ and the number of collisions $\#\{key \in \mathcal{U} : f(\{key\}) = c\}$ have to be well-balanced. Several strategies have been developed for the design of a hash function, see [20, 47]. For our purpose, choosing the modulo function and appropriate table lengths turned out to be sufficient, see [37].

Since the local multiscale transformations are performed level by level, see Algorithms 1 and 2, the hash map has to maintain the level information. For this purpose, the standard hash map is extended by a vector of length $L+1$. The idea is to have a linked list of all cells on level $l \in \{0, \dots, L\}$: the lth component of the vector contains a pointer that points to the first element of level l put into the memory heap. Additionally, the value has to be internally extended by a pointer that points to the next element of level l. This is sketched in Figure 11. Then we can access all elements of level l by traversing the resulting singly linked list. For more details on the construction of the specific hash map we refer the reader to [9].

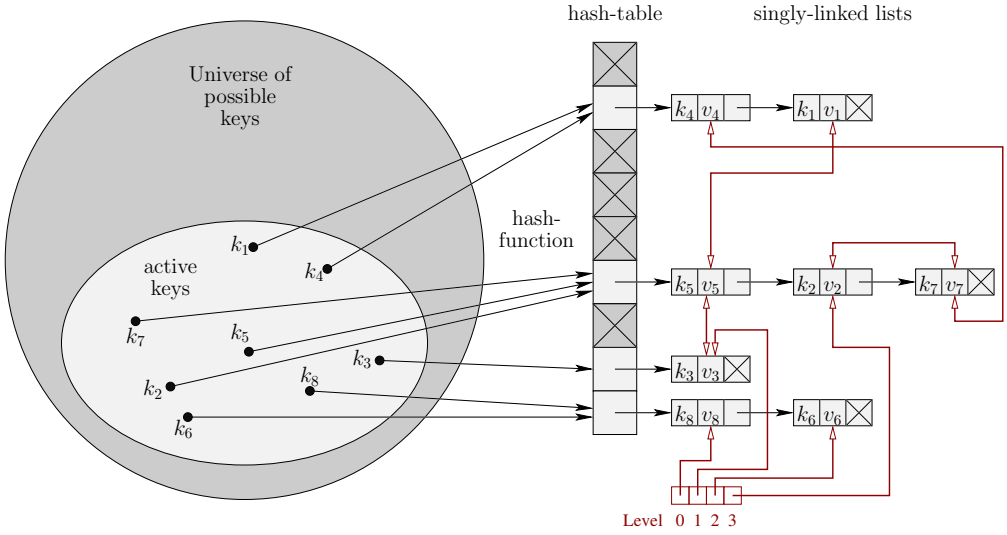


Figure 11: Linked hash map (Copyright SIAM Sci. Comput. [10]).

The above concept has been realized by the C++-template class library `igpm_t_lib` [40, 48]. This library provides data structures that are tailored to the algorithmic requirements, see Algorithms 1 and 2, from which the fundamental design criteria are deduced, namely, (i) *dynamic memory operations* and (ii) *fast data access* with respect to inserting, deleting and finding elements. In particular, it enables fast memory operations by allocating a sufficiently large memory block and by managing the algorithm's memory requirements with a specific data structure. In addition, since the overall memory demand can only be estimated, the data structure provides dynamic extension of the memory.

4. Parallelization

For 3D computations the adaptive multiresolution concept has to be realized on parallel architectures to make the computational load feasible. For this purpose we employ the concept of space-filling curves for providing an optimal load-balancing and embed this into the Algorithms 1 and 2 by which we perform the local multiscale analysis.

4.1. **Load-Balancing**

When it comes to parallelisation, we have to take into account the mesh partitioning or the load-balancing problem. As the starting point is a hierarchy of nested grids, we do not need to partition a single uniform refined mesh, but a locally refined grid where not all cells on all levels of refinement are active. Hence, the efficiency of the parallel algorithm depends on an equal distribution of the data among the processors. On the other hand, the interprocessor boundary surface is another factor that influences the overall performance, since it has a direct effect on the amount of interprocessor communication. There are mainly two distinct approaches applied: advanced partitioning tools based on sometimes complicated heuristics and more simplistic methods based on geometric approaches.

A first approach for achieving a good load-balancing is based on graph partitioning. In mathematics, the graph partitioning problem consists of dividing a graph into pieces, such that the pieces are of about the same size and there are few connections between the pieces. This problem is known to be NP-complete, hence, a number of heuristics providing good solutions have been developed and implemented in libraries like Metis. The second approach to partition a mesh is based on space-filling curves, which is relatively easy to implement

and much cheaper from point of view of memory and time consumption. In [46] the quality of partitioning computed with different types of space-filling curves is compared to those generated with the graph partitioning package Metis. It turned out that Metis computes partitions with lower edge-cuts than space-filling curves do. However, space-filling curves save both, a lot of time and a lot of memory. This is essential for time-dependent applications, because we frequently need to rebalance the load. Due to the dynamics of the flow field, the grid has to be often updated in order to track the waves in the flow field appropriately.

Space-Filling Curves. In our case, a natural representation of a multilevel partition of a mesh is a global enumeration of the active cells. We need a method to do this at runtime, as the adaptive mesh is also created at runtime using the multiscale representation techniques. Such an enumeration is provided by space-filling curves (SFC) by mapping a higher-dimensional domain to a onedimensional curve, i.e., the unit square or the unit cube is mapped to the unit interval. Using space-filling curves, each of the cells of the adaptive grid has a corresponding unique number on the curve. So, instead of having to split the geometrical domain to different processors, we only have to split the interval of numbers on the curve into parts that contain approximately the same numbers of cells. Each of these parts is mapped to a different processor, so that we obtain a well-balanced amount of data, but we also have to pay the cost of interprocessor communications, while neighbors from the geometrical domain may belong to different processors.

Space-filling curves have been created for purely mathematical purposes first, cf. [45]. Nowadays, these curves have several applications, one of them being the load-balancing for numerical simulations on parallel computer architectures. They can be used for data partitioning and, due to self-similarity features, multilevel partitions can also be constructed.

In the mathematical definition, a space-filling curve is a surjective, continuous mapping of the unit interval $[0, 1]$ to a compact d-dimensional domain Ω with positive measure. In our context, we restrict our attention to Ω being the unit square or the unit cube. In fact, as our grids have finite resolution, the iterates — so-called discrete space-filling curves — are applied, instead of the continuous space-filling curve. Construction of these curves is extremely inexpensive, as the SFC index for any cell in the grid can be computed using only local information, making it suitable for parallel computations.

Peano Space-Filling Curve. Depending on the type of the mesh discretization, several types of space-filling curves can be constructed. Thus, for a subdivision scheme of a square Q into nine subsquares, the Peano curve can be used. As in the case of the other space-filling curves, the construction is based on a generator template that defines the order in which the quadrants are visited. When passing to the next higher refinement level, the template is reduced to 1/3 and applied, with a possibly different orientation, to the new quadrants, and this procedure can then be applied recursively for each new refinement level. Of course, many different 3-dimensional Peano curves can be constructed, by a subdivision into 3^3 cubes. The construction template and a first refinement step for the Peano discrete space-filling curve can be seen in Figure 12. Since a detailed discussion on the construction on the Peano space-filling curve is not subject of this paper, we refer the reader to [45, 49].

Sierpiński Space-Filling Curve. When we have to deal with unstructured grids, besides the problems associated to the data management on parallel architectures, another issue is the partitioning of the grids. To solve the load-balancing problem for unstructured triangular meshes, the Sierpiński space-filling curve can be used. The construction of the Sierpiński curve can be explained more easily by a triangulation than by a template like the Hilbert or Peano curves. The starting point are four numbered triangles connected by a curve through the centers of gravity. The triangulation is then refined by successive bisection of the longest edge and the new numbering is obtained by substituting the single parent cell triangle with the child triangles. The Sierpiński curve was originally defined on a single triangle, but it can be generalized to arbitrary unstructured triangulations. Starting with an arbitrary initial curve, a finite set of refinement rules is used to subsequently construct finer meshes and the mesh is refined accordingly. Whenever an element is substituted by a set of finer elements covering the same area, the curve is substituted by a new curve that cycles through these elements exactly where the original element was removed. This process defines *generalized* Sierpiński curves on a sequence of meshes, see [49] for more details.

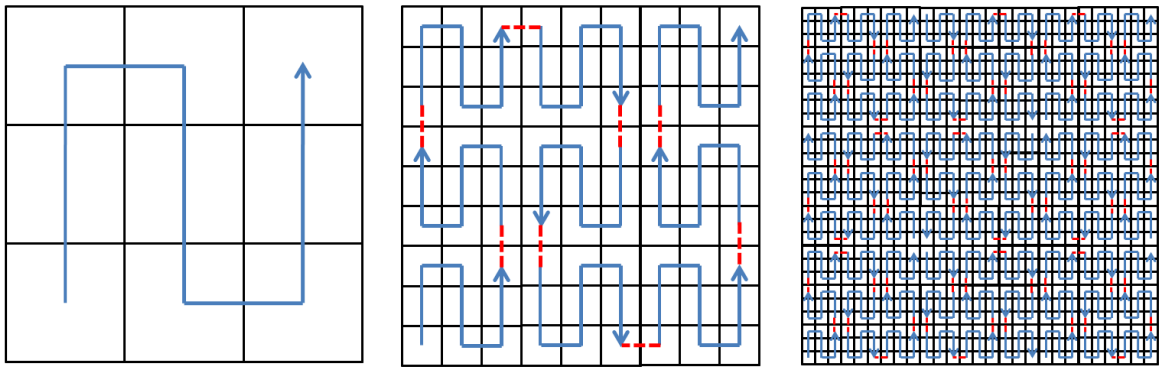


Figure 12: Template and first three refinement steps of the Peano SFC.

Another approach is proposed also by [6]. Here, the algorithm for finding an index for a triangle of the finest grid depends on three basic properties of the grid. First, the procedure mimics the bisection of triangles, then the algorithm uses the center coordinates of the triangles as nodes for the space-filling curve. Finally, a tabulated indexing scheme is used.

In our case, we consider adaptively refined grids with at most one hanging node per edge, with the refinement rules and the cell identifiers proposed by Brix et al. in [10], where a triangle on a coarser level is replaced by four triangles on the next level and each cell is uniquely identified by the *path* of children numbers. The encoding of the Sierpiński order for this type of refinement will be described below, in analogy to the encoding of the Hilbert order corresponding to dyadic grid refinement. Figure 13 shows the first three refinement steps of the Sierpiński ordering used in our approach.

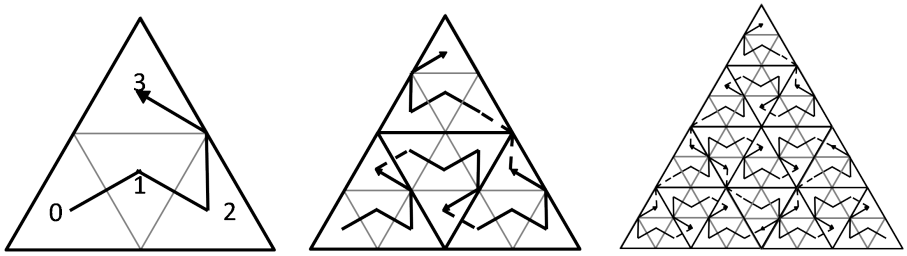


Figure 13: First three refinement steps of a Sierpiński SFC.

Hilbert Space-Filling Curve. One of the oldest space-filling curves, the Hilbert curve, can be defined geometrically, cf. [49]. The mapping is defined by the recursive subdivision of the interval I and the square Q. In 3D, the Hilbert curve is based on a subdivision into eight octants. The construction begins with a generator template, which defines the order in which the quadrants are visited. Then the template (identical, mirrored or rotated) is applied to each quadrant and, by connecting the loose ends of the curve, the next iterate of the space-filling curve is obtained. Actually, the mapping between the cells of the adaptive grid and the space-filling curve is realized using the finest iterate of the curve, which is constructed by recursively applying the template to the subquadrants (2D) and suboctants (3D) until the number of refinement levels is reached. Figures 14 and 15 show the first iterates of a 2D and 3D Hilbert SFC, respectively. Here, we only summarize the procedure of the Hilbert curve construction and focus our attention on how the curve can efficiently contribute to the parallelisation of the multiscale-based grid adaptation scheme.

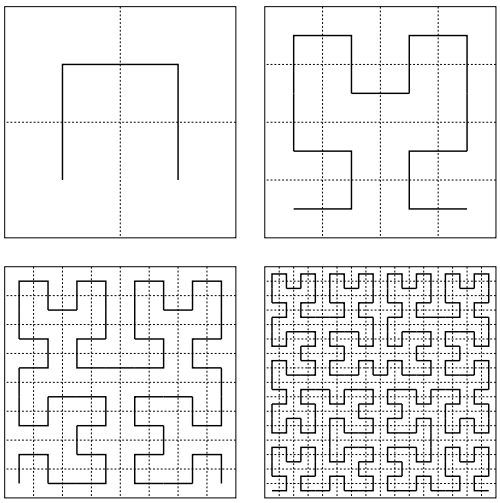


Figure 14: First 4 iterates of 2D Hilbert SFC.

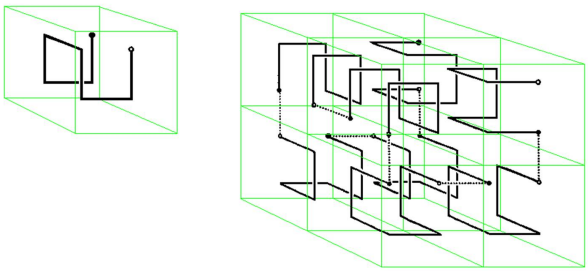


Figure 15: 1st and 2nd iterate of 3D Hilbert SFC (Courtesy of Gilbert [23]).

```
const int templates[8][4]=
{
    {0, 1, 3, 2}, // template 0
    {3, 0, 2, 1}, // template 1
    {2, 3, 1, 0}, // template 2
    {1, 2, 0, 3}, // template 3
    {3, 2, 0, 1}, // template 4
    {0, 3, 1, 2}, // template 5
    {1, 0, 2, 3}, // template 6
    {2, 1, 3, 0}, // template 7
}
```

```
const int states[8][4]=
{
    {5, 0, 7, 0}, // state 0
    {4, 6, 1, 1}, // state 1
    {2, 5, 2, 7}, // state 2
    {3, 3, 4, 6}, // state 3
    {1, 4, 3, 4}, // state 4
    {0, 2, 5, 5}, // state 5
    {6, 1, 6, 3}, // state 6
    {7, 7, 0, 2}, // state 7
}
```

Table 1: Table of templates (left) and states (right) of the 2D Hilbert curve.

Construction of the Transformation Tables. In practice, we do not actually have to construct from scratch the discrete space-filling curves on every refinement level, for every cell of the uniformly refined grid. The implementation is based on a finite state machine, so we have to determine its tables once and then use them recursively to determine the index on the curve, for any given active cell index of the adaptive mesh. The procedure for the construction of the transformation tables is described in the following.

First, the templates of the Hilbert space-filling curves for the first refinement level are determined. As mentioned above, these templates give the possible trajectories of the curve through the four subquadrants of the square Q, with the corresponding numbering. In 2D, there are eight possible templates shown in Figure 16. Starting from each of the subquadrants (00, 01, 10 or 11) with the space-filling curve number 0, we can determine four templates going clockwise and four others going counterclockwise. Each subquadrant is actually the index of a cell from the dyadic grid with one refinement level. Thus, for each cell of the 2D grid, we get the possible numbers on the curve. Each constructed template determines another entry in the *templates table*, where each column of the table $ij \in \{00, 01, 10, 11\} = \{0, 1, 2, 3\}$, i.e., the number of the quadrant, contains the possible number $\{0, 1, 2, 3\}$ of the cell on the curve for the corresponding template, see Table 1.

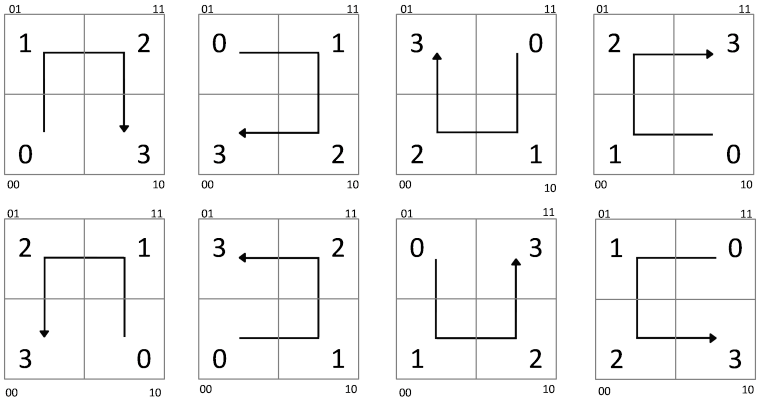


Figure 16: 2D templates of the Hilbert SFC corresponding to the templates table, see Table 1, corresponding to template 0-7 from top left to bottom right.

Starting from each of these templates, a curve on the second refinement level can be constructed (see Figure 17). The curve on the second refinement level provides information on the behavior of the curve when going from one refinement level to the next one. More specifically, if template $i, i = 0, \ldots, 7$, has to be refined, we need to know which template should be applied in each subquadrant $ij, i, j \in \{0, 1\}$. Having the new template, the procedure can then be applied recursively until the required refinement level is reached. This way we obtain a second table, the so-called *states table*, where each column $ij, i, j \in \{0, 1\}$ contains the template that is applied in each quadrant ij when refining the template from which the state was constructed, see Table 1.

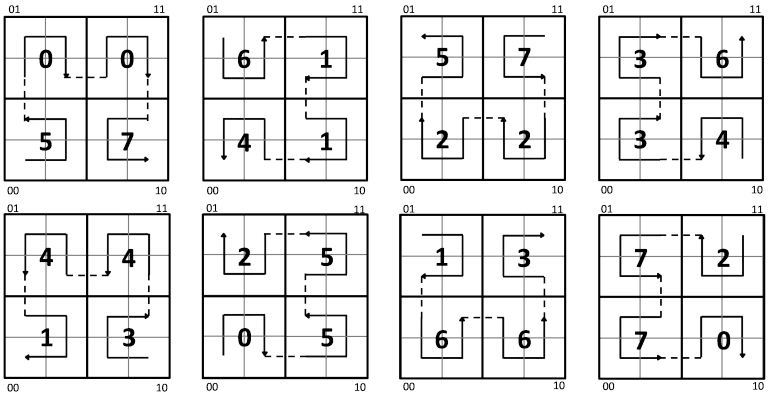


Figure 17: 2D states of the Hilbert SFC, see Table 1, corresponding to template 0-7 from top left to bottom right.

In 3D, the procedure for constructing the transformation tables is similar to the 2D case, based on the subdivision of the templates on the first refinement level into suboctants, getting then in each of the octants on the second refinement level $ijk, i, j, k \in \{0, 1\}$ the template that should be applied. Thus, the following 3D transformation tables could be obtained, see Table 2.

```
const int templates[24][8]=
{
    {0, 7, 1, 6, 3, 4, 2, 5}, // template  0
    {7, 0, 6, 1, 4, 3, 5, 2}, // template  1
    {3, 4, 0, 7, 2, 5, 1, 6}, // template  2
    {4, 3, 7, 0, 5, 2, 6, 1}, // template  3
    {1, 6, 2, 5, 0, 7, 3, 4}, // template  4
    {6, 1, 5, 2, 7, 0, 4, 3}, // template  5
    {2, 5, 3, 4, 1, 6, 0, 7}, // template  6
    {5, 2, 4, 3, 6, 1, 7, 0}, // template  7
    {0, 1, 3, 2, 7, 6, 4, 5}, // template  8
    {7, 6, 4, 5, 0, 1, 3, 2}, // template  9
    {3, 0, 2, 1, 4, 7, 5, 6}, // template 10
    {4, 7, 5, 6, 3, 0, 2, 1}, // template 11
    {1, 2, 0, 3, 6, 5, 7, 4}, // template 12
    {6, 5, 7, 4, 1, 2, 0, 3}, // template 13
    {2, 3, 1, 0, 5, 4, 6, 7}, // template 14
    {5, 4, 6, 7, 2, 3, 1, 0}, // template 15
    {0, 1, 7, 6, 3, 2, 4, 5}, // template 16
    {7, 6, 0, 1, 4, 5, 3, 2}, // template 17
    {3, 0, 4, 7, 2, 1, 5, 6}, // template 18
    {4, 7, 3, 0, 5, 6, 2, 1}, // template 19
    {1, 2, 6, 5, 0, 3, 7, 4}, // template 20
    {6, 5, 1, 2, 7, 4, 0, 3}, // template 21
    {2, 3, 5, 4, 1, 0, 6, 7}, // template 22
    {5, 4, 2, 3, 6, 7, 1, 0}, // template 23
}
```

```
const int states[24][8]=
{
  {16, 19, 8,  11, 6,  6,  8,  11}, // state  0
  {17, 18, 9,  10, 7,  7,  9,  10}, // state  1
  {4,  4,  12, 15, 17, 18, 17, 18}, // state  2
  {5,  5,  13, 14, 16, 19, 16, 19}, // state  3
  {20, 23, 20, 23, 9,  10, 2,  2 }, // state  4
  {21, 22, 21, 22, 8,  11, 3,  3 }, // state  5
  {13, 14, 0,  0,  13, 14, 21, 22}, // state  6
  {12, 15, 1,  1,  12, 15, 20, 23}, // state  7
  {0,  16, 14, 16, 5,  21, 14, 21}, // state  8
  {1,  17, 15, 17, 4,  20, 15, 20}, // state  9
  {12, 18, 1,  1,  12, 23, 4,  4 }, // state 10
  {13, 19, 0,  0,  13, 22, 5,  5 }, // state 11
  {2,  2,  17, 10, 7,  7,  20, 10}, // state 12
  {3,  3,  16, 11, 6,  6,  21, 11}, // state 13
  {19, 8,  19, 3,  22, 8,  22, 6 }, // state 14
  {18, 9,  18, 2,  23, 9,  23, 7 }, // state 15
  {0,  8,  3,  13, 22, 8,  22, 13}, // state 16
  {1,  9,  2,  12, 23, 9,  23, 12}, // state 17
  {20, 10, 20, 15, 1,  1,  2,  2 }, // state 18
  {21, 11, 21, 14, 0,  0,  3,  3 }, // state 19
  {4,  4,  7,  7,  9,  18, 12, 18}, // state 20
  {5,  5,  6,  6,  8,  19, 13, 19}, // state 21
  {11, 16, 14, 16, 11, 5,  14, 6 }, // state 22
  {10, 17, 15, 17, 10, 4,  15, 7 }, // state 23
}
```

Table 2: Table of templates (left) and states (right) of the 3D Hilbert curve.

Encoding the Hilbert SFC order. By the inverse of a discrete space-filling curve, multi-dimensional data can be mapped to a onedimensional interval. The basic idea is to map each cell of our adaptive grid to points on the space-filling curve, so that we obtain a global enumeration of the grid cells. In the data structures, we use the key composed by the cell's refinement level and the cell's multiindex on that level to identify each cell, i.e., $\lambda = (l, \boldsymbol{k})$. As each cell of the adaptive grid is uniquely identified by this key, the aim is to use it in order to determine for each cell a corresponding number on the space-filling curve. Also, due to locality properties of the curves, each cell visited is directly connected to two face-neighboring cells which remain face neighbors in the onedimensional space spanned by the curve. This way, the cell's children are sorted according to the SFC numbers and they will be nearest-neighbors on a contiguous segment of the SFC. As we look at multilevel adaptive rectangular grids, we restrict ourselves to recursively defined, self-similar SFC with rectangular recursive decomposition of the domain.

Encoding and decoding the Hilbert SFC order requires only local information, i.e., a cell's 1D index can be constructed using only that cell's integer coordinates in the d-dimensional space and the maximum number of refinement levels L that exists in the mesh. In a 2D space, consider a $2^L \times 2^L$ square ($0 \leq X \leq 2^{L-1},\ 0 \leq Y \leq 2^{L-1}$) in a Cartesian coordinate system. Note that the variable L used for determining the Hilbert SFC order might be larger than the one introduced in Section 2.1 for the number of refinement levels, since $N_{0,i} > 1$ in general and for the space-filling curve construction we should have a coarsest mesh with $N_{0,i} = 1$. Any point can be expressed by its integer coordinates, (X, Y), where X, Y are two sequences of L-bit binary numbers, as follows:

$$X = x_1 x_2 \dots x_r \dots x_L, \quad Y = y_1 y_2 \dots y_r \dots y_L.$$

Each sequence of two interleaved bits $\{x_l, y_l\}_{l=1,\dots,L}$ recursively determines on each level l one of the four quadrants the cell belongs to. Then, by using the tables of the finite state machine described previously, the state to which the curve passes when going to the next refinement level can be determined. Algorithm 4 describes how the number on the space-filling curve can be determined for a cell with a given index on a specific

refinement level. Thus by simply inspecting the cell's integer coordinates and using a finite state machine, the cell's location on a curve can easily be computed.

Algorithm 4. *(Hilbert Order Encoding)*

1. $sfc_{no} = 0$
2. $state = 0$
3. *For* $i = 1, \ldots,$ *level* L
   a) *extract the pair of bits* $x_i y_i$ *from* X, Y
   b) $quadrant = x_i y_i$
   c) $sfc_{no} = templates[state][quadrant] + 4 \cdot sfc_{no}$
   d) $state = states[state][quadrant]$

In the 3D case we proceed similarly: one cells integer coordinates (X, Y, Z) are represented as sequences of L-bit binary numbers,

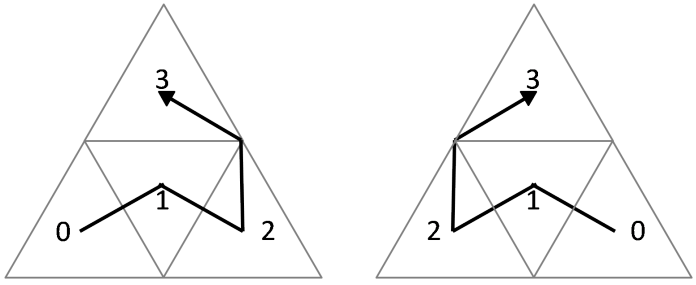
$$X = x_1 x_2 \ldots x_r \ldots x_L, \quad Y = y_1 y_2 \ldots y_r \ldots y_L, \quad Z = z_1 z_2 \ldots z_r \ldots z_L.$$

In analogy to the 2D case, the octant the cell belongs to on each level l is determined by the sequence of interleaved bits $\{x_l, y_l, z_l\}_{l=1,\ldots,L}$ and the above algorithm can be adjusted for the three dimensional case using the corresponding transformation tales.

An important aspect that should be mentioned is that the construction of the space-filling curve on an adaptive mesh ensures that a parent cell λ has exactly the same number on the SFC as *one* of its children $\mu = (l+1, 2\boldsymbol{k} + \boldsymbol{e})$, $\boldsymbol{e} \in \{0,1\}^d$, cf. [49], see also Figure 20. This leads to the minimization of the interprocessor communication in the case of a parallel MST using the Hilbert space-filling curve as partitioning scheme, as in most of the cases the parent cell should be computed on the same processor as its children.

Encoding the Sierpiński SFC order. The encoding of the Sierpiński space-filling curve order is based on the same strategy as the encoding of the Hilbert ordering. The same idea of using a finite state machine for determining how the curve passes from one state to another when increasing the refinement level is applied. Unlike the Cartesian meshes, in the case of unstructured triangular meshes we can no longer consider the bit representation of the integer indices. Instead, we use the cell numbering illustrated in Figure 10 (A) proposed by Brix et al. [10], where each cell is uniquely identified by the level l of refinement and the *path* of child numbers we have to pass through to reach that cell, see Section 3.2.

Considering this numbering, we proceed as for the Hilbert curve and first determine the templates of the space-filling curve, mainly to assign to each child number $c_i, i \in \{0,1,2,3\}$ the position on the initial iterate of the curve. Thus we obtain two different templates and two different entries in the transformation table, see Figure 18.



(a) Template 0. (b) Template 1.

Figure 18: Templates of the Sierpiński SFC corresponding to the templates table, see Table 3.

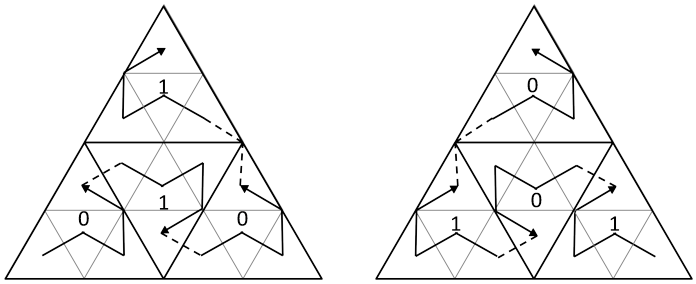
```
const int templates[2][4]=
{
  {1, 0, 2, 3}, // template 0
  {1, 2, 0, 3}, // template 1
}
```

```
const int states[2][4]=
{
  {1, 0, 0, 1}, // state 0
  {0, 1, 1, 0}, // state 1
}
```

Table 3: Table of templates (left) and states (right) of the Sierpiński curve.

The two templates lead to the construction of the curve on the next refinement level, as shown in Figures 19, and implicitly the two corresponding entries in the states table can be determined. Thus, for the triangular unstructured mesh, the following transformation tables for the finite state machine are used in Algorithm 5 to determine for any given cell of the mesh the index on the space-filling curve, see Table 3.

Here, $templates[i][j]$ represents the number of the triangle j on the curve template i, and $states[i][j]$ represents the template that should be applied in triangle j when going to a higher refinement level.



(a) State 0. (b) State 1.

Figure 19: States of the Sierpiński SFC corresponding to the templates table, see Table 3.

Algorithm 5. *(Sierpiński Order Encoding)*

1. $sfc_{no} = 0$
2. $state = 0$
3. $triangle = 0$
4. *For* $l = 1, \ldots,$ *level* L
   a) *extract* c_l*, the cell number on level* l *from path*
   b) $triangle = c_l$
   c) $sfc_{no} = templates[state][triangle] + 4 \cdot sfc_{no}$
   d) $state = states[state][triangle]$

Load-Balancing. After computing the SFC indices for all the cells in the mesh, these indices are taken as sort keys and the mesh may be ordered along the curve using standard sorting routines, such as introsort in our case. Having all the cells sorted along the curve, the partition can be easily determined, just by choosing the number of cells that each processor should get. So the mesh cells are distributed to the different processors according to their index on the curve. Since the position of each cell on the curve can be computed very inexpensively at any time in the computation, there is no need to store all the keys. Instead, it is sufficient to store the separators between the elements of the partition of the interval, i.e., the first index on a processor, in order to determine for any cell's multidimensional index the corresponding processor number. For a more specific example, Figure 20 shows a locally refined grid with three levels of refinement where each cell is mapped to a position on the

Hilbert SFC and then a partition to three processors can be determined only by ordering the numbers along the curve and then distributing it to the desired number of processors. The left side of the figure shows the Hilbert curve and numbering corresponding to the three-level uniform refined grid, while for the locally refined grid on the right it can be seen that only the active cells have corresponding indices on the Hilbert curve. In this specific example, after ordering the active SFC keys, the set of separators for a load-balancing on three processors is determined to be {40, 51, 63}.

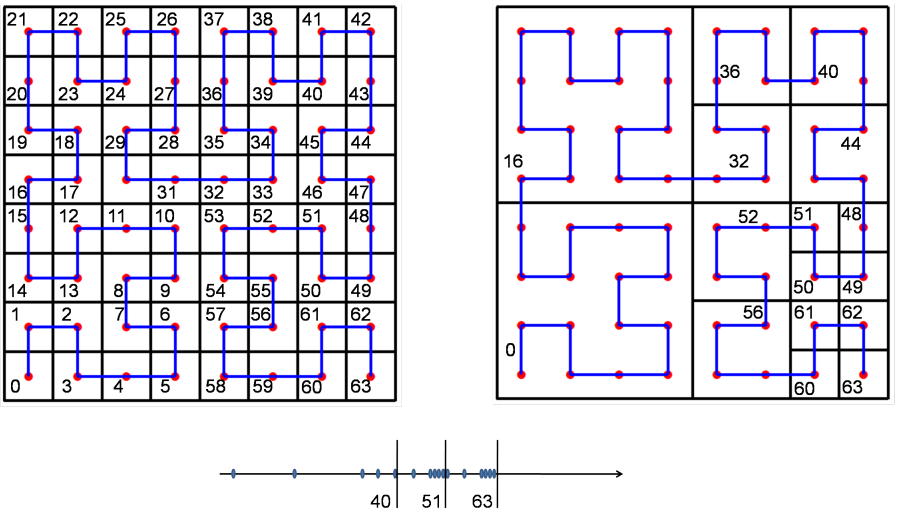


Figure 20: Encoding of the Hilbert order for a three-level adaptive grid with the corresponding split of the unit interval to three processors.

The same procedure can be applied in case of a locally adapted triangular mesh, as illustrated in Figure 21.

There are two possible choices to achieve the data partitioning and the load-balancing problem in the beginning of the computation, namely, (i) master-based partitioning and (ii) symmetric multiprocessing. In case of master-based partitioning, as its name says, the entire adaptive mesh is initialized on a master processor, according to the input file. Once the grid is initialized, the same master processor is also responsible for the entire partitioning procedure already described: the mapping of the cells to the SFC, the sorting of the keys, the load-balancing and separators' determination. After having performed these steps once, the cells can be distributed to the corresponding processors. This approach is straight forward if the starting point is a running serial algorithm, as no data transfer and no barrier points are needed before the distribution of the data to processors actually begins. On the other hand, this implies that there is only one processor active during all the initialization and sorting of the SFC procedures, while the others are idle, waiting to receive the data from the master for initializing their own data structures.

The second possibility is symmetric multiprocessing: this implies no master processor, i.e., all processors should work in parallel, executing the same code and initializing only their corresponding part of the grid. For this, a set of initial separators on the space-filling curve has to be assumed, without knowing in advance anything about the structure of the adaptive grid. So the worst case has to be taken into account, when the grid would be uniformly refined, which is equivalent to the fact that, for each number on the discrete space-filling curve, there exists an active cell in the grid. In the case of a fully refined grid, the number of cells in the grid ($2^L \times 2^L$ and $2^L \times 2^L \times 2^L$, in 2D and 3D, respectively) corresponds to the last number on the SFC and the guess of the initial separators is straight forward. The main drawback of this second approach is that this initial guess might and is very probable to be far from the optimal choice. Hence, the possibility of not having a remarkable

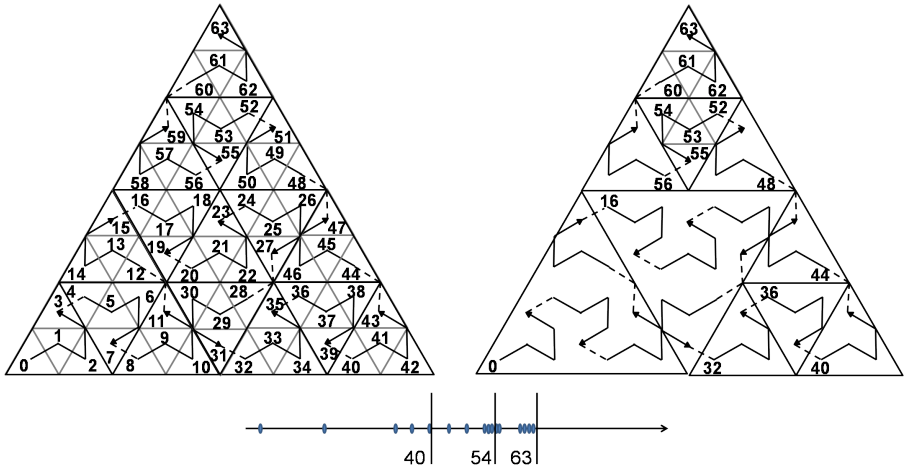


Figure 21: Encoding of the Sierpiński order for a three-level adaptive grid with the corresponding split of the unit interval to three processors.

performance improvement in the initialization part is very high, also due to the interprocessor communication costs that arise. A rebalancing of the initial data is then required in order to obtain a well-balanced distribution of data among processors and a new set of separators is computed for the new partition.

Applying either of these two strategies leads to a well-balanced data distribution as shown in Figure 22.

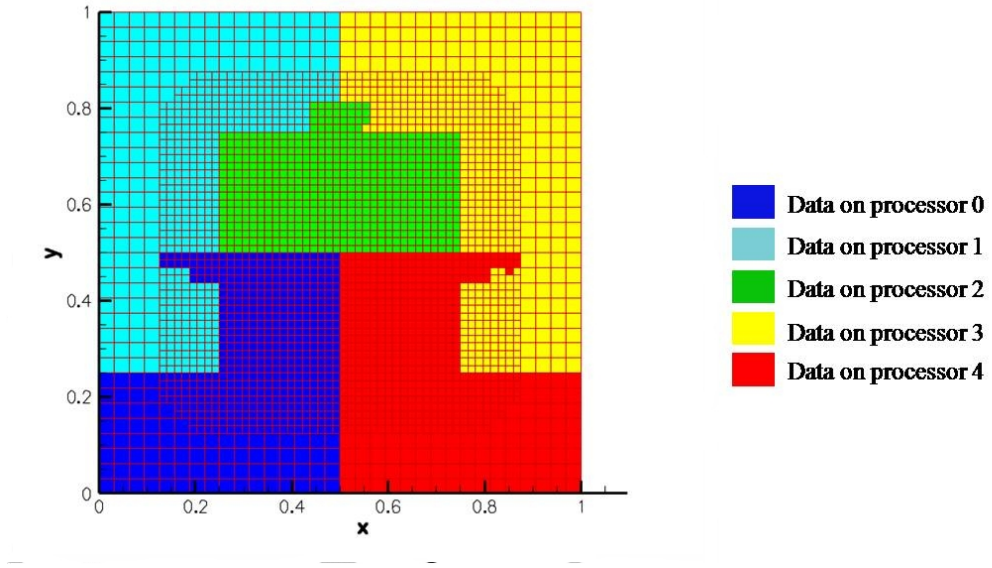


Figure 22: Well-balanced distribution of a locally refined grid to 5 processors.

Parallel rebalancing. A reordering of the cells along the curve is also needed whenever the load-balancing is significantly spoiled due to the grid adaptation process. When this occurs, a new set of separators — that determines a new well-balanced partition — has to be computed. There is no need to gather all cells on a master processor for the reordering, since all the cells on a processor p have smaller numbers on the SFC than the cells on processor $p+1$ for all $p = 0, \ldots, n_{\text{proc}} - 2$. The parallel rebalancing is described in Algorithm 6.

Algorithm 6. *(Parallel Rebalancing)*

1. *Sort local cells along the SFC according to the old list of separators* $sep_{\text{old}}[i]$ *for* $i = 0, \ldots, n_{\text{proc}} - 1$.
2. *Compute the total workload from all processors:*
   $\text{total\_workload} = \sum_{p=0}^{n_{\text{proc}}-1} \#\text{cells}(p)$
3. *Compute the positions of the new separators on the SFC:*
   a) $\text{new\_positions}[0] = 0$
   b) *For* $i = 1, \ldots, n_{\text{proc}} - 1$ *do*
   $\text{new\_positions}[i] = \text{new\_positions}[i-1] + i \cdot (\text{total\_workload}/n_{\text{proc}})$
4. *For* $pos = 1, \ldots, n_{\text{proc}} - 1$ *do*
   If $\text{new\_positions}[pos]$ *belongs to the local processor, then determine the new separator* $\text{sep}_{\text{new}}[\text{new\_positions}[pos]]$ *at position* $\text{new\_positions}[pos]$.
5. *Distribute new separators to all processors.*
6. *Redistribute data according to the new separators.*

4.2. Parallel Grid Adaptation and Data Transfer

Once load-balancing is achieved, each processor should perform the grid adaptation, see Section 2.1, on the local data. Special attention must be paid to the cells located at the processor's boundary, i.e., the cells that have at least one neighbor belonging to another processor. As these are the only ones that make the difference between serial and parallel algorithms and as they are similarly handled in all the steps of the grid adaptation, in the sequel we will go into details only in the parallelisation of the encoding step, by mainly discussing the special treatment applied to the boundary cells. As described in Section 2.2, the encoding step consists of computing the cell averages on level l starting from data on level $l+1$ and the computation of the details on level l. Since the approach for parallelizing the coarsening is different to the one for the details computation, they will be discussed separately.

Parallel coarsening. To compute the parent's cell average λ on the processor indicated by the parent's position on the SFC and the separators between the elements of the partition, all its children μ, $\mu \in \mathcal{M}_\lambda^0$, should already be available on the same processor. Due to the locality properties of the SFC and the compactness of each element of the partition, i.e., the ratio of an element's volume and its surface is large, ensured by its construction, the children are nearest neighbors in the onedimensional space, leading to the fact that only some few cells at the partitions' boundaries have to be transferred between processors before computing the cell averages on level l, see Figure 23. When running through the active data on level $l+1$ for constructing the set of parent cells that need to be computed, the processor the parent belongs to is also determined and so a buffer is set up, containing the child cells that need to be transferred to a neighbor processor. The buffer is transferred to the corresponding processor before the computation of the averages on level l actually begins. Once the cell averages of all cells' parents have been computed, the ghost cells transferred from other processors can be deleted from the local hash map.

Algorithm 7. *(Parallel Coarsening) Proceed levelwise from* $l = L - 1$ *downto 0: (cf. Algorithm 1, Step I)*

I. **Computation of cell averages on level** l*:*
   1. *For each active cell on level* $(l+1)$ *determine*
      a) *the parent cell on level* l*;*
      b) *the processor* p *to which the parent belongs to.*
   If p *is the current processor then* $U_l^0 := \bigcup_{\mu \in I_{l+1,\varepsilon}} \mathcal{M}_\mu^{*,0}$ *where* $I_{l+1,\varepsilon} := I_{l+1} \cap \tilde{\mathcal{G}}_{L,\varepsilon}$
   else transfer cell $\mu \in I_{l+1,\varepsilon}$ *to processor* p *and there add it to the local hash map.*

2. *Compute cell averages for parents on level l:*
   $\hat{u}_\lambda = \sum_{\mu \in \mathcal{M}_\lambda^0} m_{\mu,\lambda}^0 \hat{u}_\mu,\ \lambda \in U_l^0$
3. *Delete the cells received from other processors.*

Parallel details computation. In the case of the details computation, more data from the neighbor processors have to be transferred, see Figure 24. For the parallel coarsening, a single step data transfer is made, since each processor can determine the cells on level $l+1$ that are necessary for the neighbors to compute the averages on level l by themselves. In case of the details computation a two step transfer has to be performed. In a first step, each processor has to send a request to the others for the cells that influence the details computation of the local cells, so we obtain a first pair of MPI_Isend and MPI_Recv calls. A new pair of such calls is needed to fulfill the requests, when actually all the data located at the geometrical boundary of the partitions has to be transferred to the neighboring processors.

Algorithm 8. *(Parallel Details Computation) Proceed levelwise from $l = L-1$ downto 0: (cf. Algorithm 1, Step II)*

I. **Computation of details on level** l*:*

0. *On each processor initialize the index sets $U_{l,p}^1 = \emptyset$, $p = 0, \ldots, n_{proc} - 1$*
1. *For each active cells on level $\mu \in I_{l+1,\varepsilon}$ do*
   a) *determine all cells on level l influencing their corresponding details: $\lambda \in \mathcal{M}_\mu^{*,1}$*
   b) *for each $\lambda \in \mathcal{M}_\mu^{*,1}$ determine the processor p where the details of λ should be computed:*
   if $p = p_{loc}$ (current processor) then $U_{l,p}^1 := U_{l,p}^1 \cup \{\lambda\}$;
   else transfer λ to processor p and there add it to the index set $U_{l,p}^1 := U_{l,p}^1 \cup \{\lambda\}$ and transfer cell μ to processor p and there add it to the local hash map.
2. *For each detail on level l determine the cell averages on level $l+1$ that are needed to compute the detail:*
   $P_{l+1} := \bigcup_{\lambda \in U_{l,p}^1} \mathcal{M}_\lambda^1 \backslash I_{l+1,\varepsilon}$
3. *For all indices $\mu \in P_{l+1}$ that belong to other processors $p \neq p_{loc}$ (current processor), $p = 0, \ldots, n_{proc}-1$ do*
   a) *send requests to processor p to transfer the unavailable data;*
   b) *receive needed data from processor p.*
4. *Accept requests from other processors and send back the data to the other processors requested from the current processor.*
5. *Compute a prediction value for the cell averages on level $l+1$ not available in the adaptive grid:*
   $\hat{u}_\mu = \sum_{\lambda \in \mathcal{G}_\mu^0} g_{\lambda,\mu}^0 \hat{u}_\lambda,\ \mu \in P_{l+1}$
6. *Compute the details on level l:*
   $d_\lambda := \sum_{\mu \in \mathcal{M}_\lambda^1} m_{\mu,\lambda}^1 \hat{u}_\mu,\ \lambda \in U_l^1$
7. *Delete data received from other processors from the local hash map.*

5. Numerical Results

As a test case we consider the two-dimensional inviscid problem of wave interaction, which has been investigated in [35]. The configuration consists of four regions, separated by membranes, bursting at t_0. Each sector is initialized with different constant states, so that the jump conditions are fulfilled at the interfaces. The initial conditions for the four sectors are summarized in Figure 25.

The computational domain is $\Omega = [0,1]^2$ and the coarsest grid is composed of 10×10 cells. The threshold value for the adaptation is $\varepsilon = 10^{-3}$. The time discretization on the finest discretization level is fixed by

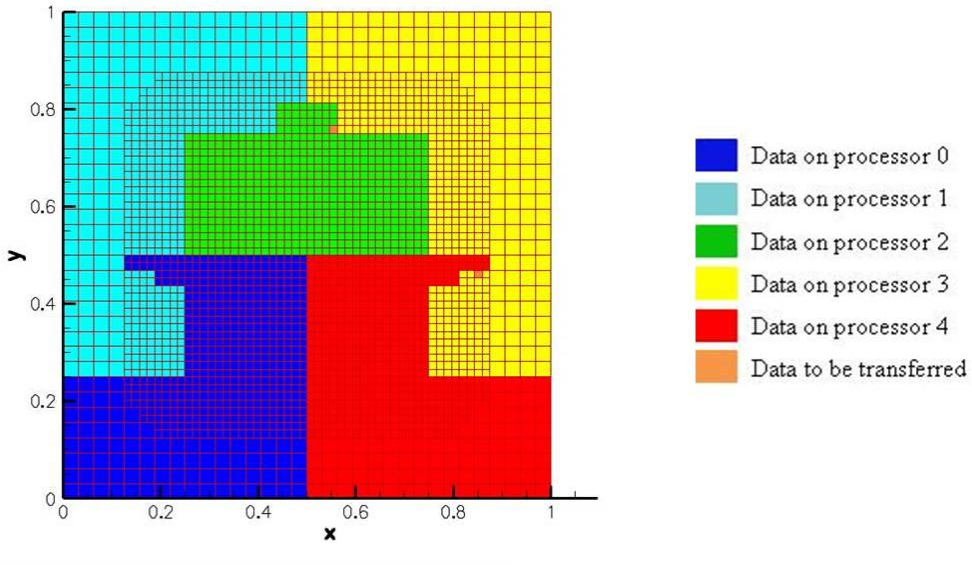


Figure 23: Cells to be transferred for parallel coarsening.

$\tau_L = 0.0064 \times 2^{-L}$. The computations are performed with an essentially non-oscillatory (ENO) scheme. This ENO scheme is determined by a one-dimensional second order reconstruction technique via primitive variables, cf. [29]. Here the primitive variables are reconstructed by means of piecewise linear polynomials. To avoid oscillations at discontinuities the minmod limiter is applied. At the cell interfaces Riemann problems are solved approximatively by the Roe solver. For time discretization we perform an explicit Euler step. By means of a Cauchy-Kowalevski procedure we formally obtain second order accuracy also in time. For details on the solver we refer to [36].

We present the results of the computation with $L = 8$ refinement levels on increasing number of processors. In Figure 26 (A-D) the density, the pressure, the temperature as well as the Mach number distribution are depicted. Characteristic for this configuration are the shear layer and the jet in the center of the flow field. Figure 26 (E) shows the corresponding locally refined grid verifying that all flow features such as shock waves, shear layers and the jet are detected and well-resolved by the multiresolution-based grid adaptation.

To investigate the efficiency of the parallelization, we performed several computations with increasing number of processors on a locally refined grid that eventually consists of 335.176 cells out of 6.553.600 cells of the uniformly refined reference mesh. After the partitioning is done, each processor is getting a number of cells equal to the total number of cells in the adaptive grid over the number of processors. Note that the processor with the highest rank also takes the few cells that remain if the total number of entries cannot be divided by the number of processors. This ensures a very good initial load-balancing. Due to the adaptation process, rebalancing is performed after 1000 time steps. Figure 27 illustrates the results of these experiments and a good scaling with respect to the different numbers of processors and the different modules of the application, namely the flow solver and the multiscale transformation modules realizing the grid adaptation process. Since we plot the logarithm of the cpu time versus the logarithm of the number of processors, the optimal scaling would correspond to a slope of 1 as indicated by the reference triangle. For our computations we realize a slope of approximately 0.9. The corresponding speed up is quantified in Table 4.

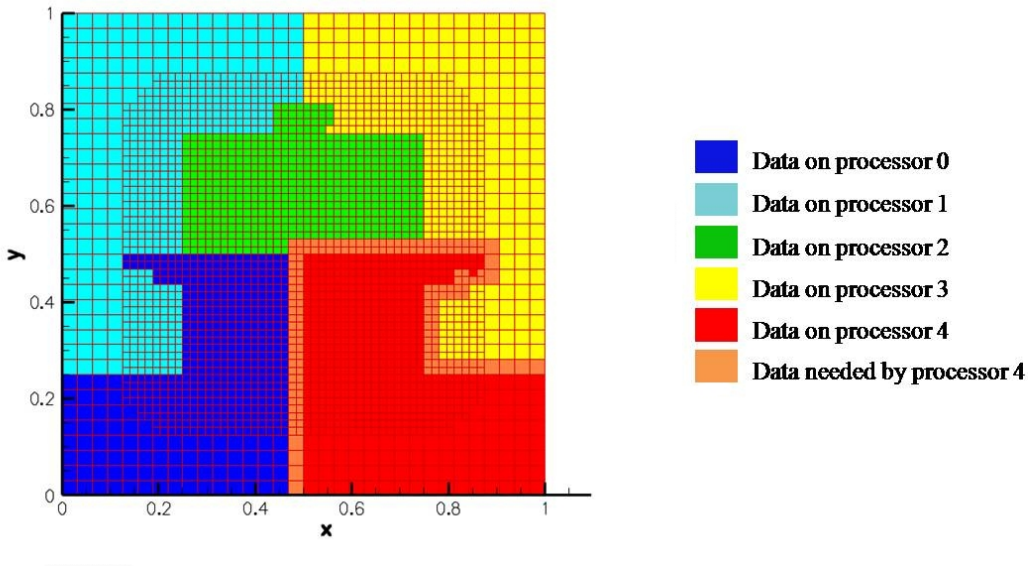


Figure 24: Cells to be transferred for parallel details computations.

	Sector 1	Sector 2	Sector 3	Sector 4
p	0.138000	0.532250	0.532250	1.500000
ρ	0.166428	0.000000	0.641894	0.000000
u	0.166428	0.641894	0.000000	0.000000
v	0.273212	1.137062	1.137062	3.750000

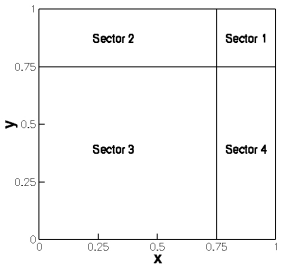


Figure 25: Initial conditions for two-dimensional wave interaction.

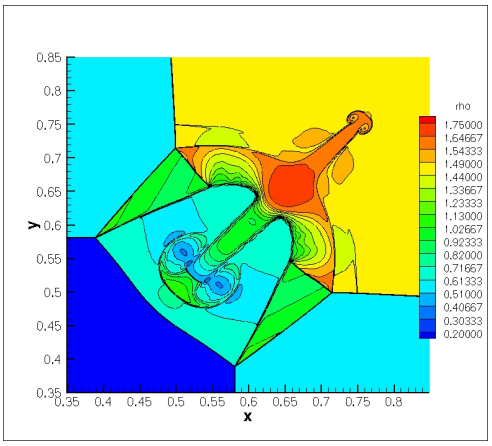
References

[1] S. Andreae. *Wave Interactions with Material Interfaces*. PhD thesis, RWTH Aachen, 2008.

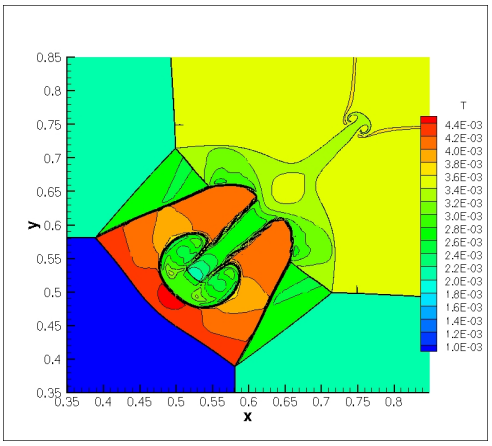
[2] S. Andreae, J. Ballmann, and S. Müller. Wave processes at interfaces. In G. Warnecke, editor, *Analysis and numerics for conservation laws*, pages 1–25. Springer, Berlin, 2005.

[3] S. Balay, K. Buschelmann, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc web page, `http://www.mcs.anl.gov/petsc`. Technical report, 2001.

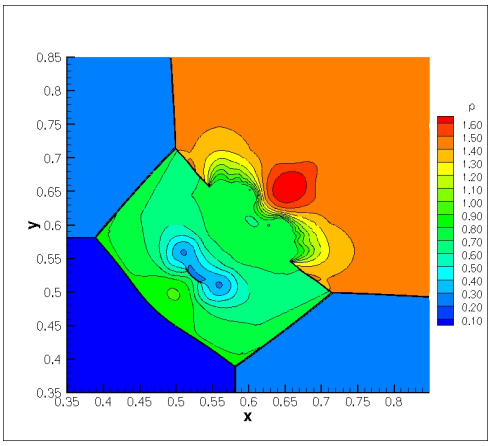
[4] S. Balay, K. Buschelmann, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc users manual. Technical report anl-95/11 - revision 2.1.5, Argonne National Laboratory, 2004.



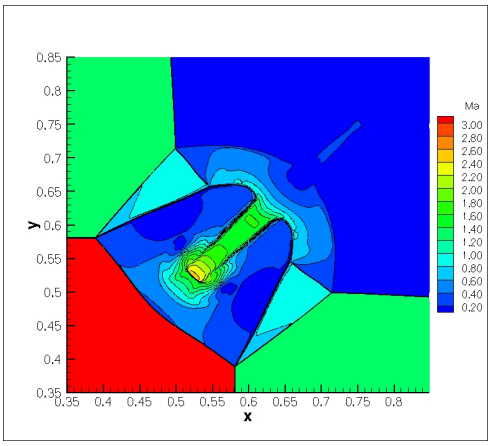
(a) Density.



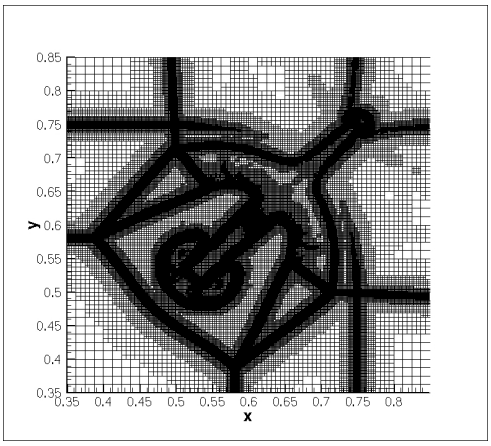
(b) Temperature



(c) Pressure.



(d) Mach number.



(e) Adaptive grid.

Figure 26: Two-dimensional wave interaction.

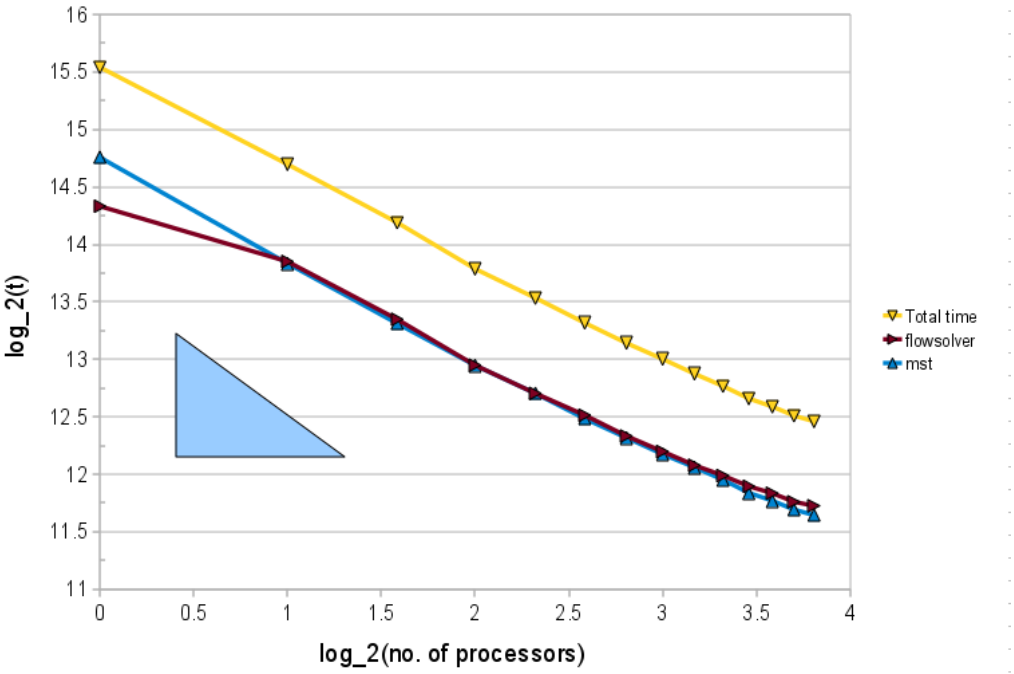


Figure 27: CPU times for performing the two-dimensional wave interaction experiments on increasing number of processors.

# proc.	flow solver	MST	total	speed up
1	20623	27686	47687	1.00
2	14772	14599	26559	1.80
3	10427	10165	18688	2.55
4	7906	7896	14136	3.37
5	6670	6667	11857	4.02
6	5847	5727	10204	4.67
7	5157	5102	9032	5.28
8	4697	4621	8204	5.81
9	4323	4269	7506	6.35
10	4063	3961	6970	6.84
11	3808	3660	6466	7.38
12	3651	3491	6158	7.74
13	3467	3309	5830	8.18
14	3385	3201	5632	8.47

Table 4: CPU times [s] and speed up for performing the two-dimensional wave interaction experiments on increasing number of processors.

[5] S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, Basel, 1997.

[6] J. Behrens and J. Zimmermann. Parallelizing an unstructured grid generator with a space-filling curve approach. In *EURO-PAR 2000, NUMBER 1900 IN LNCS*, pages 815–823. Springer, Berlin, 2000.
[7] F. Bramkamp, B. Gottschlich-Müller, M. Hesse, Ph. Lamby, S. Müller, J. Ballmann, K.-H. Brakhage, and W. Dahmen. *H*-adaptive Multiscale Schemes for the Compressible Navier–Stokes Equations — Polyhedral Discretization, Data Compression and Mesh Generation. In J. Ballmann, editor, *Flow Modulation and Fluid-Structure-Interaction at Airplane Wings*, volume 84 of *Numerical Notes on Fluid Mechanics*, pages 125–204. Springer, Berlin, 2003.
[8] F. Bramkamp, Ph. Lamby, and S. Müller. An adaptive multiscale finite volume solver for unsteady and steady state flow computations. *J. Comp. Phys.*, 197(2):460–490, 2004.
[9] K. Brix, R. Massjung, and A. Voss. A hash data structure for adaptive PDE–solvers based on Discontinuous Galerkin discretizations. IGPM–Report 302, RWTH Aachen, 2009.
[10] K. Brix, R. Massjung, and A. Voss. Refinement and connectivity algorithms for adaptive discontinuous Galerkin methods. *SIAM J. Sci. Comput.*, 33(1):66–101, 2011.
[11] R. Bürger, R. Ruiz, and K. Schneider. Fully adaptive multiresolution schemes for strongly degenerate parabolic equations with discontinuous flux. *J. Eng. Math.*, 60(3–4):365–385, 2008.
[12] R. Bürger, R. Ruiz, K. Schneider, and M.A. Sepulveda. Fully adaptive multiresolution schemes for strongly degenerate parabolic equations in one space dimension. *ESAIM Math. Model. Numer. Anal.*, 42(4):535–563, 2008.
[13] J.M. Carnicer, W. Dahmen, and J.M. Peña. Local decomposition of refinable spaces and wavelets. *Appl. Comput. Harmon. Anal.*, 3:127–153, 1996.
[14] A. Cohen, I. Daubechies, and J. Feauveau. Bi–orthogonal bases of compactly supported wavelets. *Comm. Pure Appl. Math.*, 45:485–560, 1992.
[15] A. Cohen, S.M. Kaber, S. Müller, and M. Postel. Fully Adaptive Multiresolution Finite Volume Schemes for Conservation Laws. *Math. Comp.*, 72(241):183–225, 2003.
[16] A. Cohen, S.M. Kaber, and M. Postel. Multiresolution Analysis on Triangles: Application to Gas Dynamics. In G. Warnecke and H. Freistühler, editors, *Hyperbolic Problems: Theory, Numerics, Applications*, pages 257–266. Birkhäuser, Basel, 2002.
[17] F. Coquel, Y. Maday, S. Müller, M. Postel, and Q.H. Tran. Multiresolution and adaptive methods for convection-dominated problems. ESAIM Proc., 29, 2009.
[18] F. Coquel, Q.L. Nguyen, M. Postel, and Q.H. Tran. Local time stepping applied to implicit-explicit methods for hyperbolic systems. *Multiscale Model. Simul.*, 8(2):540–570, 2009.
[19] F. Coquel, M. Postel, N. Poussineau, and Q.H. Tran. Multiresolution technique and explicit-implicit scheme for multicomponent flows. *J. Numer. Math.*, 14:187–216, 2006.
[20] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2nd edition, 2001.
[21] M. Domingues, O. Roussel, and K. Schneider. On space-time adaptive schemes for the numerical solution of partial differential equations. *ESAIM Proc.*, 16:181–194, 2007.
[22] M. Duarte, M. Massot, S. Descombes, C. Tenaud, T. Dumont, V. Louvet, and F. Laurent. New resolution strategy for multiscale reaction waves using time operator splitting, space adaptive multiresolution and dedicated high order implicit/explicit time integrators. HAL-00457731, version 1, 2010. `http://hal.archives-ouvertes.fr/hal-00457731/en/`.
[23] E. N. Gilbert. Gray codes and the paths on the n-cube. *Bell System Tech. J.*, 37:815–826, 1958. `http://www.math.uwaterloo.ca/~wgilbert/Research/HilbertCurve/HilbertCurve.html`.
[24] B. Gottschlich–Müller. *Multiscale Schemes for Conservation Laws*. PhD thesis, RWTH Aachen, 1998.
[25] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, MA, 2nd edition, 1999.
[26] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI-2 - Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, MA, 2nd edition, 1999.
[27] A. Harten. Adaptive multiresolution schemes for shock computations. *J. Comp. Phys.*, 115:319–338, 1994.
[28] A. Harten. Multiresolution algorithms for the numerical solution of hyperbolic conservation laws. *Comm. Pure Appl. Math.*, 48(12):1305–1342, 1995.
[29] A. Harten, B. Engquist, S. Osher, and S.R. Chakravarthy. Uniformly high order accurate essentially non–oscillatory schemes III. *J. Comp. Phys.*, 71:231–303, 1987.
[30] B. Hejazialhosseini, D. Rossinelli, M. Bergdorf, and P. Koumoutsakos. High order finite volume methods on wavelet-adapted grids with local time-stepping on multicore architectures for the simulation of shock-bubble interactions. *J. Comp. Physics*, 229:8364–8383, 2010.
[31] N. Hovhannisyan and S. Müller. On the stability of fully adaptive multiscale schemes for conservation laws using approximate flux and source reconstruction strategies. *IMA J. Numer. Anal.*, 30:1256–1295, 2010.
[32] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Supercomputing*, 1998.
[33] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48:71–85, 1998.
[34] Ph. Lamby. *Parametric Multi-Block Grid Generation and Application to Adaptive Flow Simulations*. PhD thesis, RWTH Aachen, 2007. `http://darwin.bth.rwth-aachen.de/opus3/volltexte/2007/1999/pdf/Lamby_Philipp.pdf`.

[35] P.D. Lax and X.D. Liu. Solution of two-dimensional Riemann problems of gas dynamics by positive schemes. *SIAM J. Sci. Comput.*, 19(2):319–340, 1998.
[36] S. Müller. *Erweiterung von ENO–Verfahren auf zwei Raumdimensionen und Anwendung auf hypersonische Stauspunktprobleme*. PhD thesis, RWTH Aachen, 1993.
[37] S. Müller. *Adaptive Multiscale Schemes for Conservation Laws*, volume 27 of *Lecture Notes on Computational Science and Engineering*. Springer, Berlin, 2003.
[38] S. Müller. Multiresolution schemes for conservation laws. DeVore, Ronald (ed.) et al., Multiscale, nonlinear and adaptive approximation. Dedicated to Wolfgang Dahmen on the occasion of his 60th birthday, pages 379–408, Springer, Berlin, 2009., 2009.
[39] S. Müller, Ph. Helluy, and J. Ballmann. Numerical simulation of a single bubble by compressible two-phase fluids. *Int. J. Numer. Methods Fluids*, 62(6):591–631, 2010.
[40] S. Müller and A. Voss. A Manual for the Template Class Library `igpm_t_lib`. IGPM–Report 197, RWTH Aachen, 2000.
[41] O. Roussel and K. Schneider. A fully adaptive multiresolution scheme for 3D reaction–diffusion equations. In B. Herbin, editor, *Finite Volumes for Complex Applications*. Hermes Science, Paris, 2002.
[42] O. Roussel and K. Schneider. Adaptive multiresolution method for combustion problems: Application to flame ball-vortex interaction. *Comput. & Fluids*, 34(7):817–831, 2005.
[43] O. Roussel and K. Schneider. Numerical studies of spherical flame structures interacting with adiabatic walls using an adaptive multiresolution scheme. *Combust. Theory Modelling*, 10(2):273–288, 2006.
[44] O. Roussel, K. Schneider, A. Tsigulin, and H. Bockhorn. A conservative fully adaptive multiresolution algorithm for parabolic PDEs. *J. Comp. Phys.*, 188(2):493–523, 2003.
[45] H. Sagan. *Space-Filling Curves*. Springer, Berlin, 1st edition, 1994.
[46] S. Schamberger and J.-M. Wierum. Partitioning finite element meshes using space-filling curves. *Future Gener. Comput. Syst.*, 21(5):759–766, 2005.
[47] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, 1997.
[48] A. Voss. Notes on adaptive grids in 2d and 3d, Part I: Navigating through cell hierarchies using cell identifiers. IGPM–Report 268, RWTH Aachen, 2006.
[49] G. Zumbusch. *Parallel multilevel methods. Adaptive mesh refinement and loadbalancing*. Advances in Numerical Mathematics. Teubner, Wiesbaden, 2003.