ESAIM: PROCEEDINGS, December 2009, Vol. 29, p. 108-129
F. Coquel, Y. Maday, S. Müller, M. Postel and Q. H. Tran, Editors

PARALLELISATION OF MULTISCALE-BASED GRID ADAPTATION USING SPACE-FILLING CURVES *

Kolja Brix[1], Silvia Sorana Melian[1], Siegfried Müller[1] and Gero Schieffer[2]

Résumé. Le concept des schémas de volumes finis multi-échelles et adaptatifs a été développé et étudié pendant les dix dernières années. Jusqu'à maintenant il a été utilisé avec succès dans de multiples applications provenant de l'ingéniérie. Dans le but de réaliser des simulations en 3D avec des géométries complexes en un temps de calcul raisonnable, la stratégie de maillage adaptatif multi-échelles a du être parallélisée via MPI pour des architectures à mémoires partagées. Pour de bonnes performances du point de vue du temps de calcul et de la gestion de la mémoire, la quantité de données doit être bien répartie et la communication entre les processeurs doit être minimisée. Ceci a été réalisé à l'aide des courbes remplissantes.

Abstract. The concept of fully adaptive multiscale finite volume schemes has been developed and investigated during the past decade. By now it has been successfully employed in numerous applications arising in engineering. In order to perform 3D computations for complex geometries in reasonable CPU time, the underlying multiscale-based grid adaptation strategy has to be parallelised via MPI for distributed memory architectures. In view of a proper scaling of the computational performance with respect to CPU time and memory, the load of data has to be well-balanced and communication between processors has to be minimised. This has been realised using space-filling curves.

Introduction

The numerical simulation of (compressible) fluid flow requires highly efficient numerical algorithms which allow for a high resolution of all physical waves occurring in the flow field and their dynamical behaviour. In order to use the computational resources (CPU time and memory) in an efficient way, adaptive schemes are well-suited . By these schemes, the discretisation is locally adapted to the variation of the flow field. The crucial point in the adaptation process is the design of a criterion by which to decide whether to refine or to coarsen the grid locally.

In recent years, a new adaptive concept for finite volume schemes, frequently applied to the discretisation 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 [24, 25]. The basic idea is to transform the arrays of cell averages associated with any given finite volume discretisation into a different format that reveals insight into the local smoothness behaviour of the solution. The cell averages on a given highest level of

* *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.*

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

[2] Lehrstuhl für Computergestützte Analyse Technischer Systeme, RWTH Aachen University, Steinbachstr. 53B, 52074 Aachen, Germany, email: schieffer@cats.rwth-aachen.de

© EDP Sciences, SMAI 2009

Article published by EDP Sciences and available at http://www.edpsciences.org/proc or http://dx.doi.org/10.1051/proc/2009058

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, threshold techniques are applied to the multiresolution decomposition in [13,30], 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 modelled by the Euler and Navier–Stokes equations, respectively, on block–structured curvilinear grid patches [9], backward–facing step on 2D triangulations [14] and simulation of a flame ball modelled by reaction–diffusion equations on 3D Cartesian grids [34,37]. 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 [2, 3, 31]. By now, there are several groups working on this subject: Coquel et al. [15, 16], Roussel et al. [35, 36], Burger et al. [11, 12] and Domingues et al. [18].

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` [32] has been developed. It has been recently extended with respect to unstructured grid hierarchies, see [10, 41]. In view of an optimal memory management and a fast data access the well-known concept of *hash maps*, cf. [17], is used. Since the multiscale-based grid adaptation requires sweeping through the different refinement levels, it turned out that hash maps are better suited for this purpose than tree structures. The latter need some overhead to access children, parents and neighbours in the tree, cf. [37]. 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. [17]. Using hash maps, the multiscale library, cf. [30], has been realised and incorporated successfully into the multi-block finite volume solver Quadflow [8, 9].

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 parallelised 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 frequently employed using the Metis software [27, 28] together with PETSc [4–6]. Opposite to this approach, we use space-filling curves, cf. [42]. Here the basic idea is to map level-dependent multi-indices identifying the cells in a dyadic 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 the case of the hash maps.

In [39] 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 appropriately track the waves in the flow field.

The aim of the present work is to give an overview on the parallelisation of the multiscale-based grid adaptation via MPI [21, 22] using space-filling curves. For this purpose, we first summarise in Section 1 the basic

ingredients of the multiscale library: (i) the multiscale analysis of the discrete cell averages and grid adaptation, (ii) algorithms and (iii) data structures. When we consider parallelisation, we have to care for load-balancing and interprocessor communication. This issue is addressed in Section 2. An optimal balancing of the load can be realised using space-filling curves. By means of the local multiscale transformation we discuss the data transfer at processor boundaries. In Section 3, we explain the work to be done to embed the parallelised multiscale-library into the flow solver Quadflow. Finally, in Section 4, we present some performance studies for the parallel multiscale transformation and show first adaptive, parallel 3D computations of a Lamb-Oseen vortex using the parallelised Quadflow solver.

1. Basic Ingredients

In this section we summarise the ingredients 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 discretisations 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, can be determined. In principle, the concept can be applied to any hierarchy of nested grids, no matter whether these grids are structured or unstructured. However, here we will confine ourselves to structured grids and uniform dyadic refinements where on each refinement level the grids can be determined by evaluation of a grid mapping. This has been successfully realised in the multiscale library, see [30] for details, and incorporated into the Quadflow solver [8, 9].

1.1. Multiscale Analysis and Grid Adaptation

Grid mapping. The starting point is a smooth function $\boldsymbol{x} : R := [0,1]^d \to \Omega$, which maps the parameter domain R onto the computational domain Ω. The Jacobian is assumed to be regular, i.e., $\det(\partial \boldsymbol{x}(\boldsymbol{\xi})/\partial \boldsymbol{\xi}) \neq 0$, $\boldsymbol{\xi} \in R$. In our applications we represent the grid function by B–splines, see [29]. This admits control of good local grid properties, e.g., orthogonality and smoothness of the grid, and a consistent boundary representation by a small number of control points depending on the configuration at hand.

Nested Grid Hierarchy. A nested grid hierarchy is defined from the grid mapping by means of a sequence of nested uniform partitions of the parameter domain. To this end, we introduce the sets of multi–indices $I_l := \prod_{i=1}^d \{0, \ldots, N_{l,i} - 1\} \subset \mathbf{N}_0^d$, $l = 0, \ldots, L$, with $N_{l,i} = 2\, N_{l-1,i}$ initialised 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 the nested sequence of parameter partitions $\mathcal{R}_l := \{R_{l,\boldsymbol{k}}\}_{\boldsymbol{k} \in I_l}$, $l = 0, \ldots, L$, is determined by $R_{l,\boldsymbol{k}} := \prod_{i=1}^d [k_i\, h_{l,i}, (k_i + 1)\, h_{l,i}]$, with $h_{l,i} := 1/N_{l,i} = h_{l-1,i}/2$, see Figure 1. Finally, a sequence of *nested grids* $\mathcal{G}_l := \{V_{l,k}\}_{\boldsymbol{k} \in I_l}$, $l = 0, \ldots, L$, of the computational domain Ω is obtained by $V_{l,\boldsymbol{k}} := \boldsymbol{x}(R_{l,\boldsymbol{k}})$, see Figure 2 for an illustration. Each grid $\mathcal{G}_l$ builds a partition of Ω, i.e.,

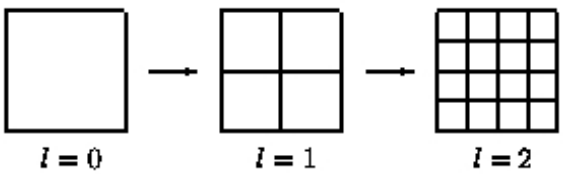

Figure 1. Dyadic grid hierarchy in parameter space.

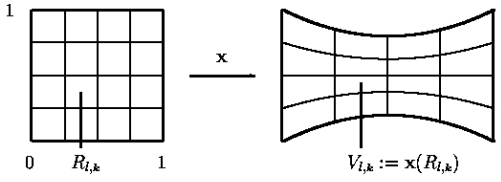

Figure 2. Transformation from parameter to computational domain.

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

Multiscale decomposition. By means of the grids $\mathcal{G}_l$ we introduce the sequences of averages $\hat{\boldsymbol{u}}_l := \{\hat{u}_{l,\boldsymbol{k}}\}_{\boldsymbol{k}\in I_l}$ corresponding to a scalar, integrable function $u \in L^1(\Omega, \mathbf{R})$ as the inner product $u_{l,\boldsymbol{k}} := \langle u, \phi_{l,\boldsymbol{k}}\rangle_{L^2(\Omega)}$ of u with the L^1–normalised box function $\phi_{l,\boldsymbol{k}}(\boldsymbol{x}) := |V_{l,\boldsymbol{k}}|^{-1}\chi_{V_{l,\boldsymbol{k}}}(\boldsymbol{x})$, $\boldsymbol{x} \in \Omega$, where $|V_{l,\boldsymbol{k}}| := \int_{V_{l,\boldsymbol{k}}} 1\, d\boldsymbol{x}$ denotes the cell volume and $\chi_{V_{l,\boldsymbol{k}}}$ the characteristic function on $V_{l,\boldsymbol{k}}$. Then the nestedness of the grids as well as the linearity of the integration operator imply the two–scale relation

$$\hat{u}_{l,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{M}^0_{l,\boldsymbol{k}}} m^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l+1,\boldsymbol{r}}, \quad m^{l,0}_{\boldsymbol{r},\boldsymbol{k}} := \frac{|V_{l+1,\boldsymbol{r}}|}{|V_{l,\boldsymbol{k}}|}, \tag{1}$$

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 (1). 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 $S_l := \mathrm{span}\{\phi_{l,\boldsymbol{k}}\ ;\ \boldsymbol{k} \in I_l\}$ 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., $W_l := \mathrm{span}\{\psi_{l,\boldsymbol{k},\boldsymbol{e}}\ ;\ \boldsymbol{k} \in I_l,\ \boldsymbol{e} \in E^* := E\backslash\{\boldsymbol{0}\}\}$. For the construction of an appropriate wavelet basis we refer to [30]. In analogy to the cell averages, the details can be introduced as inner products $d_{l,\boldsymbol{k},\boldsymbol{e}} := \langle u, \psi_{l,\boldsymbol{k},\boldsymbol{e}}\rangle_{L^2(\Omega)}$ of the function u with the *wavelet* $\psi_{l,\boldsymbol{k},\boldsymbol{e}}$. Since the box functions and the wavelets are linearly independent, there exists a two–scale relation for the details, i.e.,

$$d_{l,\boldsymbol{k},\boldsymbol{e}} = \sum_{\boldsymbol{r}\in\mathcal{M}^{\boldsymbol{e}}_{l,\boldsymbol{k}}\subset I_{l+1}} m^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l+1,\boldsymbol{r}}. \tag{2}$$

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

$$\hat{u}_{l+1,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{G}^0_{l,\boldsymbol{k}}\subset I_l} g^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l,\boldsymbol{r}} + \sum_{\boldsymbol{e}\in E^*}\sum_{\boldsymbol{r}\in\mathcal{G}^{\boldsymbol{e}}_{l,\boldsymbol{k}}\subset I_l} g^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}\, d_{l,\boldsymbol{r},\boldsymbol{e}}. \tag{3}$$

Note that the non-vanishing mask coefficients $m^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$ and $g^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$ in (1), (2) and (3), corresponding to the index sets $\mathcal{M}^0_{l,\boldsymbol{k}}$, $\mathcal{M}^{\boldsymbol{e}}_{l,\boldsymbol{k}}$, $\boldsymbol{e} \in E^*$, and $\mathcal{G}^0_{l,\boldsymbol{k}}$, $\mathcal{G}^{\boldsymbol{e}}_{l,\boldsymbol{k}}$, $\boldsymbol{e} \in E^*$, respectively, do not depend on the data but on geometric information only. For specific examples we refer to [30].

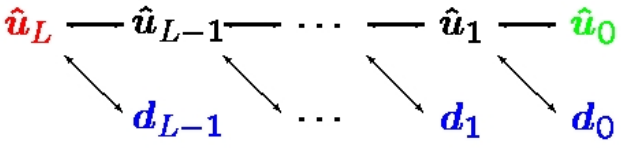

FIGURE 3. Pyramid scheme of multiscale transformation.

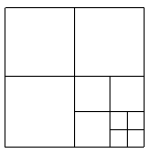

FIGURE 4. Locally refined grid.

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 (1) and (2), as illustrated in Figure 3. 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 $d_{l,\boldsymbol{k},\boldsymbol{e}}$ whose absolute values fall below a level-dependent threshold value $\epsilon_l = 2^{(l-L)d}\epsilon$ and keep only the *significant details* corresponding to the index set $\mathcal{D}_{L,\epsilon} := \{(l,\boldsymbol{k})\ ;\ |d_{l,\boldsymbol{k},\boldsymbol{e}}| > \varepsilon_l,\ \boldsymbol{k} \in I_l,\ \boldsymbol{e} \in E^*,\ l \in \{0,\ldots,L-1\}\}$. 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 $\tilde{\mathcal{D}}_{L,\epsilon} \supset \mathcal{D}_{L,\epsilon}$ contains all significant details of the old and the new time level. In a last step, we construct the locally refined grid, see Figure 4, and corresponding cell averages. For this purpose, we proceed levelwise from coarse to fine, see Figure 3, 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 (3). The final grid is then determined by the index set $\tilde{\mathcal{G}}_{L,\epsilon} \subset \bigcup_{l=0}^{L} \{l\} \times I_l$ such that $\bigcup_{(l,\boldsymbol{k})\in\tilde{\mathcal{G}}_{L,\epsilon}} V_{l,\boldsymbol{k}} = \Omega$. In order to proceed levelwise, we have to inflate the prediction set such that it corresponds to a graded tree. This also guarantees that there is at most one hanging node at a cell edge, see [30].

1.2. 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 (1), (2) and (3) have to be restricted to those indices which correspond to non–vanishing entries of the mask coefficients. Introducing the mask matrices $\boldsymbol{M}_{l,\boldsymbol{e}} = (m^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}})_{\boldsymbol{r}\in I_{l+1},\boldsymbol{k}\in I_l}$ and $\boldsymbol{G}_{l,\boldsymbol{e}} = (g^{l,\boldsymbol{e}}_{\boldsymbol{k},\boldsymbol{r}})_{\boldsymbol{r}\in I_l,\boldsymbol{k}\in I_{l+1}}$, and the vectors $\boldsymbol{u}_l = \{u_{l,\boldsymbol{k}}\}_{\boldsymbol{k}\in I_l}$ and $\boldsymbol{d}_{l,\boldsymbol{e}} = \{d_{l,\boldsymbol{k},\boldsymbol{e}}\}_{\boldsymbol{k}\in I_l}$, $\boldsymbol{e} \in E^*$, these two-scale relations may be rewritten in terms of matrix-vector products, $\boldsymbol{u}_l = \boldsymbol{M}^T_{l,0}\ \boldsymbol{u}_{l+1}$, $\boldsymbol{d}_{l,\boldsymbol{e}} = \boldsymbol{M}^T_{l,\boldsymbol{e}}\ \boldsymbol{u}_{l+1}$, $\boldsymbol{e} \in E^*$, and $\boldsymbol{u}_{l+1} = \boldsymbol{G}^T_{l,0}\ \boldsymbol{u}_l + \sum_{\boldsymbol{e}\in E^*} \boldsymbol{G}^T_{l,\boldsymbol{e}}\ \boldsymbol{d}_{l,\boldsymbol{e}}$. 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.,

$$\mathcal{A}_k := \operatorname{supp}(\boldsymbol{A}, k) := \{r\ ;\ a_{r,k} \neq 0\} = \text{ support of } k\text{th column of } \boldsymbol{A},$$

$$\mathcal{A}^*_k := \operatorname{supp}(\boldsymbol{A}^T, k) := \{r\ ;\ a_{k,r} \neq 0\} = \text{ support of } k\text{th row of } \boldsymbol{A},$$

where $\boldsymbol{A}$ denotes one of the above mask matrices. The support $\mathcal{A}_k$ of a column collects all non–vanishing matrix elements that might yield a non–trivial contribution to the kth component of the matrix–vector product, i.e., y_k. Therefore $\mathcal{A}_k$ can be interpreted as the *domain of dependence* for y_k, i.e., the components x_r which contribute to y_k. The support $\mathcal{A}^*_k$ of a row collects all non–vanishing matrix entries of the kth row that might yield a non–trivial contribution to the vector $\boldsymbol{y}$ of the matrix–vector product. Therefore $\mathcal{A}^*_k$ can be interpreted as the *range of influence*, i.e., the components y_r which are influenced by the component x_k.

With this notation in mind, we now can give efficient algorithms for locally performing the decoding and encoding processes as they have been realised 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:*
      $U^0_l := \bigcup_{\boldsymbol{r}\in I_{l+1,\varepsilon}} \mathcal{M}^{*,0}_{l,\boldsymbol{r}}$ *where* $I_{l+1,\varepsilon} := \{\boldsymbol{k} \in I_{l+1} : (l+1,\boldsymbol{k}) \in \tilde{\mathcal{G}}_{L,\epsilon}\}$
   2. *Compute cell averages for parents on level l:*
      $\hat{u}_{l,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{M}^0_{l,\boldsymbol{k}}} m^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l+1,\boldsymbol{r}},\ \boldsymbol{k} \in U^0_l$

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:*
      $U^{\boldsymbol{e}}_l := \bigcup_{\boldsymbol{r}\in I_{l+1,\varepsilon}} \mathcal{M}^{*,\boldsymbol{e}}_{l,\boldsymbol{r}},\ \boldsymbol{e} \in E^*$
   2. *For each detail on level l determine the cell averages on level $l+1$ that are needed to compute the detail:*
      $P_{l+1} := \bigcup_{\boldsymbol{e}\in E^*} \bigcup_{\boldsymbol{k}\in U^{\boldsymbol{e}}_l} \mathcal{M}^{\boldsymbol{e}}_{l,\boldsymbol{k}} \backslash I_{l+1,\boldsymbol{\varepsilon}}$
   3. *Compute a prediction value for the cell averages on level $l+1$ not available in adaptive grid:*
      $\hat{u}_{l+1,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{G}^0_{l,\boldsymbol{k}}} g^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l,\boldsymbol{r}},\ \boldsymbol{k} \in P_{l+1}$

4. *Compute the details on level l:*
   $d_{l,\boldsymbol{k},\boldsymbol{e}} := \sum_{\boldsymbol{r}\in\mathcal{M}^{\boldsymbol{e}}_{l,k}} m^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}} \hat{u}_{l+1,\boldsymbol{r}},\ \boldsymbol{k}\in U^{\boldsymbol{e}}_l,\ \boldsymbol{e}\in E^*$

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:*
      $I^+_{l+1} := \bigcup_{\boldsymbol{e}\in E^*}\bigcup_{\boldsymbol{l}\in J_{l,\varepsilon}} \mathcal{G}^{*,\boldsymbol{e}}_{l,\boldsymbol{l}}$ *where* $J_{l,\varepsilon} := \{\boldsymbol{k}\in I_l\ :\ (l,\boldsymbol{k})\in\tilde{\mathcal{D}}_{L,\epsilon}\}$
   2. *Compute cell averages for cells on level $l+1$:*
      $\hat{u}_{l+1,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{G}^0_{l,\boldsymbol{k}}} g^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\,\hat{u}_{l,\boldsymbol{r}} + \sum_{\boldsymbol{e}\in E^*}\sum_{\boldsymbol{r}\in\mathcal{G}^{\boldsymbol{e}}_{l,\boldsymbol{k}}} g^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}\,d_{l,\boldsymbol{e},\boldsymbol{r}},\quad \boldsymbol{k}\in I^+_{l+1}$

II. **Remove refined cells on level l:**
   1. *For each cell on level $l+1$ determine its parent cell on level l:*
      $I^-_l := \bigcup_{\boldsymbol{k}\in I^+_{l+1}} \mathcal{M}^{*,0}_{l,\boldsymbol{k}}$
   2. *Remove the parent cells on level l from the adaptive grid:*
      delete $\hat{u}_{l,\boldsymbol{k}},\ \boldsymbol{k}\in I^-_l$, *i.e.,* $I_{l,\varepsilon} := I^+_l / I^-_l$ *(Note:* $I^+_0 = I_0$*)*

1.3. Data Structures

In order to realise the reduced algorithmical complexity, we need appropriate data structures. These have to be designed such that the computational complexity (storage and CPU time) is proportional to the cardinality of the adaptive grid and the significant details, respectively. For this purpose, the C++-template class library `igpm_t_lib` [10, 32, 41] has been developed. 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.

Due to refinement and coarsening operations in the algorithm, memory operations are frequently performed and therefore should be very fast. This can be realised more efficiently 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 should provide dynamic extension of the memory.

In view of an optimal memory management and a fast data access we use the well-known concept of *hash maps*, cf. [17], that is composed of two parts, namely, a vector of pointers, a so–called *hash table*, and a memory heap, see Figure 5. The hash table is connected to a *hash function* $f : \mathcal{U} \to \mathcal{T}$, which maps a *key*, here $(l,\boldsymbol{k})$, 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., $\mathcal{U} = \{(l,\boldsymbol{k})\ :\ \boldsymbol{k}\in I_l,\ l=0,\dots,L\}$, and $\mathcal{T}$ corresponds to the keys of the dynamically changing adaptive grid, i.e., $(l,\boldsymbol{k})\in\tilde{\mathcal{G}}_{L,\epsilon}$.

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 optimise 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 [17, 40]. For our purpose, choosing the modulo function and appropriate table lengths turned out to be sufficient, see [30].

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. The idea is to have a linked list of all cells on level 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 6. Then we can access all elements of level l by traversing the resulting singly linked list.

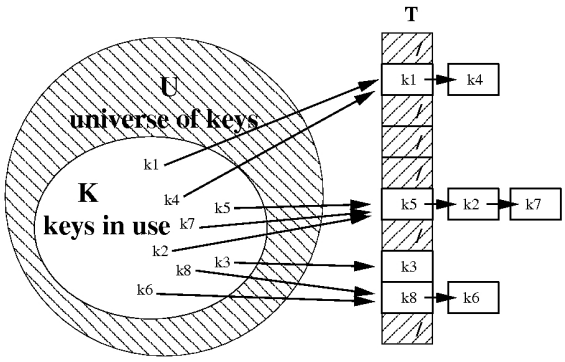

FIGURE 5. Hashing (Courtesy of [42]).

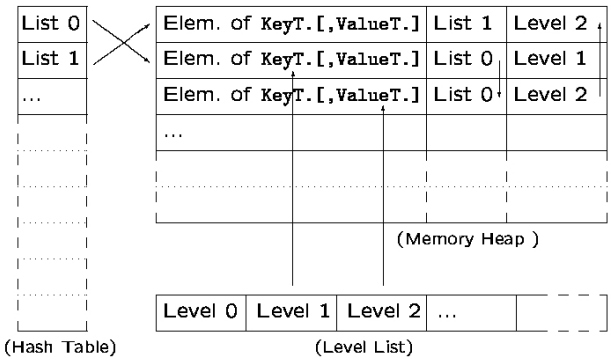

FIGURE 6. Linked hash map.

2. Parallelisation

In the following we will present how we have parallelised the multiscale library by which we perform local grid adaptation in one block using multiscale techniques. For this purpose, we first outline the strategy of load-balancing using space-filling curves, see Section 2.1. In a second step, see Section 2.2, we explain how to perform the multiscale transformation in parallel. Here the crucial point is the handling of the cells on the partition boundary and interprocessor communication. Note that the multiscale-based grid adaptation consists of additional steps such as thresholding, prediction, grading and decoding. The parallelisation of these steps is in complete analogy and therefore not detailed here.

2.1. Load-Balancing via Space-Filling Curves

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. 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 one-dimensional 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 neighbours from the geometrical domain may belong to different processors.

Space-Filling Curve. Space-filling curves have first been created for purely mathematical purposes, cf. [38]. 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.

Hilbert Space-Filling Curve. One of the oldest space-filling curves, the Hilbert curve, can be defined geometrically, cf. [42]. 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 realised 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 7 and 8 show the first iterates of a 2D and 3D Hilbert SFC, respectively. A detailed discussion on the construction on the Hilbert space-filling curve is not the subject of this paper, for this we refer the reader to [38, 42]. Here, we only summarise 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.

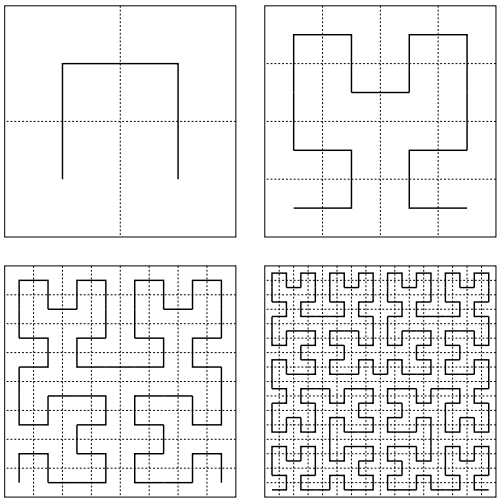

FIGURE 7. First 4 iterates of 2D Hilbert SFC.

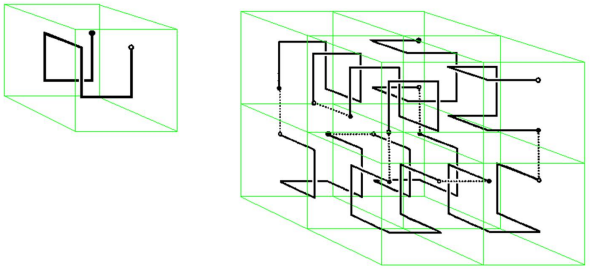

FIGURE 8. 1st and 2nd iterate of 3D Hilbert SFC (Courtesy of Gilbert [20]).

Encoding the Hilbert SFC order. By the inverse of a discrete space-filling curve, multi-dimensional data can be mapped to a one-dimensional 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 multidimensional index on that level $(l, \boldsymbol{k})$ to identify each cell. 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-neighbouring cells which remain face neighbours in the one-dimensional space spanned by the curve. This way, the cell's children are sorted according to the SFC numbers and they will be nearest-neighbours 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 1.1 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 $\{x_l, y_l\}_{l=1,\dots,L}$ determines on each level l which one of the four quadrants the cell belongs to, recursively. 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, cf. [42]. In the 3D case we proceed similarly.

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 $(l, \boldsymbol{k})$ has exactly the same number on the SFC as *one* of its children $(l+1, 2\boldsymbol{k}+\boldsymbol{e})$, $\boldsymbol{e} \in \{0,1\}^d$, cf. [42]. This leads to the minimisation 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.

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.

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 initialised on a master processor, according to the input file. Once the grid is initialised, 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 straightforward 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 initialisation and sorting of the SFC procedures, while the others are idle, waiting to receive the data from the master for initialising 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 initialising 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, so the possibility of not having a remarkable performance improvement in the initialisation 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 9.

Parallel rebalancing. A reordering of the cells along the curve is also needed whenever the load-balancing is significantly spoiled due to the adaptivity of the grid. When this occurs, a new set of separators — that determine 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 $p = 0, \dots, n_{\text{proc}} - 2$. The parallel rebalancing is described in Algorithm 3.

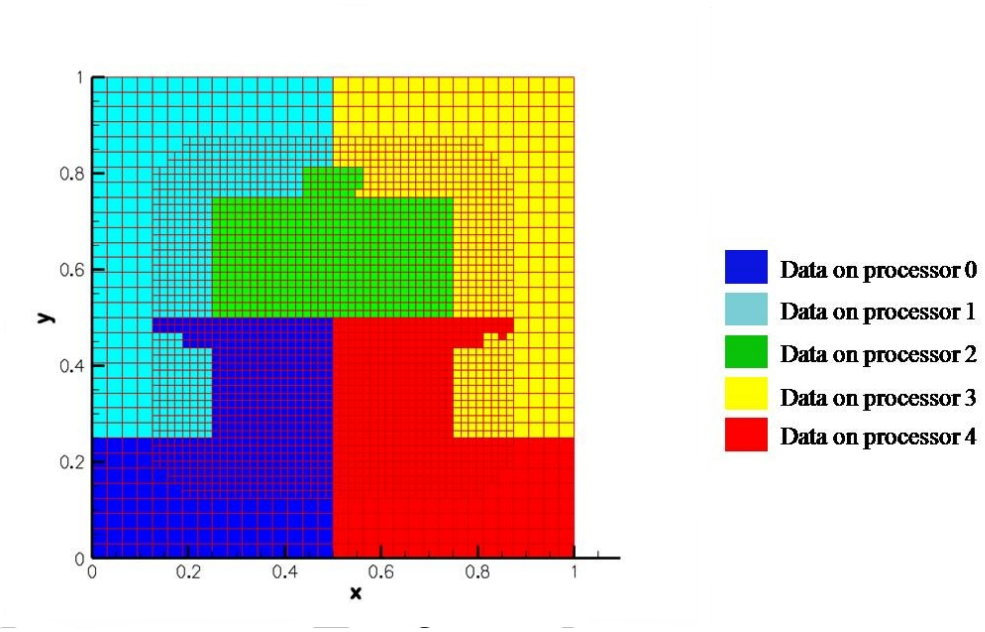

FIGURE 9. Well-balanced distribution of a locally refined grid to 5 processors.

Algorithm 3. *(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* $\text{sep}_{\text{new}}[pos]$ *at position* $\text{new\_positions}[pos]$.
5. *Distribute new separators to all processors.*
6. *Redistribute data according to the new separators.*

2.2. Parallel Grid Adaptation and Data Transfer

Once load-balancing is achieved, each processor should perform the grid adaptation, see Section 1.1, 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 neighbour 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 1.2, the encoding step consists of computing both the cell averages and the details on level l starting from data on level $l + 1$. Since the approach for parallelising the coarsening (or averaging) step (see Algorithm 1, Step I) is different from the one for computing the details (see Algorithm 1, Step II), they will be discussed separately.

Parallel coarsening. To compute the parent's cell average $\hat{u}_{l,\boldsymbol{k}}$ on the processor indicated by the parent's position on the SFC and the separators between the elements of the partition, all its children $\hat{u}_{l+1,\boldsymbol{r}}$, $\boldsymbol{r} \in \mathcal{M}^0_{l,\boldsymbol{k}}$, 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 neighbours in the one-dimensional 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 10. 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 children cells that need to be transferred to a neighbour 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 4. *(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 $(l+1, \boldsymbol{r})$, $\boldsymbol{r} \in I_{l+1,\varepsilon}$, determine*
   a) *the parent cell on level l;*
   b) *the processor p which the parent belongs to.*

   If p is the current processor then $U^0_l = U^0_l \cup \mathcal{M}^{*,0}_{l,\boldsymbol{r}}$;
   else transfer cell $(l+1, \boldsymbol{r})$ to processor p and there add it to the local hash map.
2. *Compute cell averages for parents on level l:*
   $\hat{u}_{l,\boldsymbol{k}} = \sum_{\boldsymbol{r} \in \mathcal{M}^0_{l,\boldsymbol{k}}} m^{l,0}_{\boldsymbol{r},\boldsymbol{k}}\, \hat{u}_{l+1,\boldsymbol{r}},\ \boldsymbol{k} \in U^0_l$
3. *Delete the cells received from other processors.*

Parallel details computation. In the case of the details computation, more data from the neighbour processors have to be transferred, see Figure 11. 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 neighbours 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 fulfil the requests, when actually all the data located at the geometrical boundary of the partitions has to be transferred to the neighbouring processors.

Algorithm 5. *(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 initialise the index sets $U^{\boldsymbol{e}}_{l,p} = \emptyset$, $p = 0, \ldots, n_{proc}-1$*
1. *For each active cells on level $(l+1, \boldsymbol{r})$, $\boldsymbol{r} \in I_{l+1,\varepsilon}$ do*
   a) *determine all cells on level l influencing their corresponding details:*
   $(l, \boldsymbol{k}) \in \mathcal{M}^{*,\boldsymbol{e}}_{l,\boldsymbol{r}}$, $\boldsymbol{e} \in E^*$;
   b) *for each $(l, \boldsymbol{k}) \in \mathcal{M}^{*,\boldsymbol{e}}_{l,\boldsymbol{r}}$ determine the processor p where the details of cell $(l, \boldsymbol{k})$ should be computed:*
   if $p = p_{loc}$ (current processor) then $U^{\boldsymbol{e}}_{l,p} := U^{\boldsymbol{e}}_{l,p} \cup \{\boldsymbol{k}\}$;
   else transfer index $(l, \boldsymbol{k})$ to processor p and there add it to the index set $U^{\boldsymbol{e}}_{l,p} := U^{\boldsymbol{e}}_{l,p} \cup \{\boldsymbol{k}\}$ and transfer cell $(l+1, \boldsymbol{r})$ 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:*
   $P_{l+1} := \bigcup_{\boldsymbol{e} \in E^*} \bigcup_{\boldsymbol{k} \in U^{\boldsymbol{e}}_l} \mathcal{M}^{\boldsymbol{e}}_{l,\boldsymbol{k}} \backslash I_{l+1,\boldsymbol{\varepsilon}}$

3. *For all indices* $(l+1, \boldsymbol{r}) \in P_{l+1}$ *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:* $\hat{u}_{l+1,\boldsymbol{k}} = \sum_{\boldsymbol{r} \in \mathcal{G}^0_{l,\boldsymbol{k}}} g^{l,0}_{\boldsymbol{r},\boldsymbol{k}} \hat{u}_{l,\boldsymbol{r}},\ \boldsymbol{k} \in P_{l+1}$
6. *Compute the details on level* l*:* $d_{l,\boldsymbol{k},\boldsymbol{e}} := \sum_{\boldsymbol{r} \in \mathcal{M}^{\boldsymbol{e}}_{l,k}} m^{l,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}} \hat{u}_{l+1,\boldsymbol{r}},\ \boldsymbol{k} \in U^{\boldsymbol{e}}_l,\ \boldsymbol{e} \in E^*$
7. *Delete data received from other processors from the local hash map.*

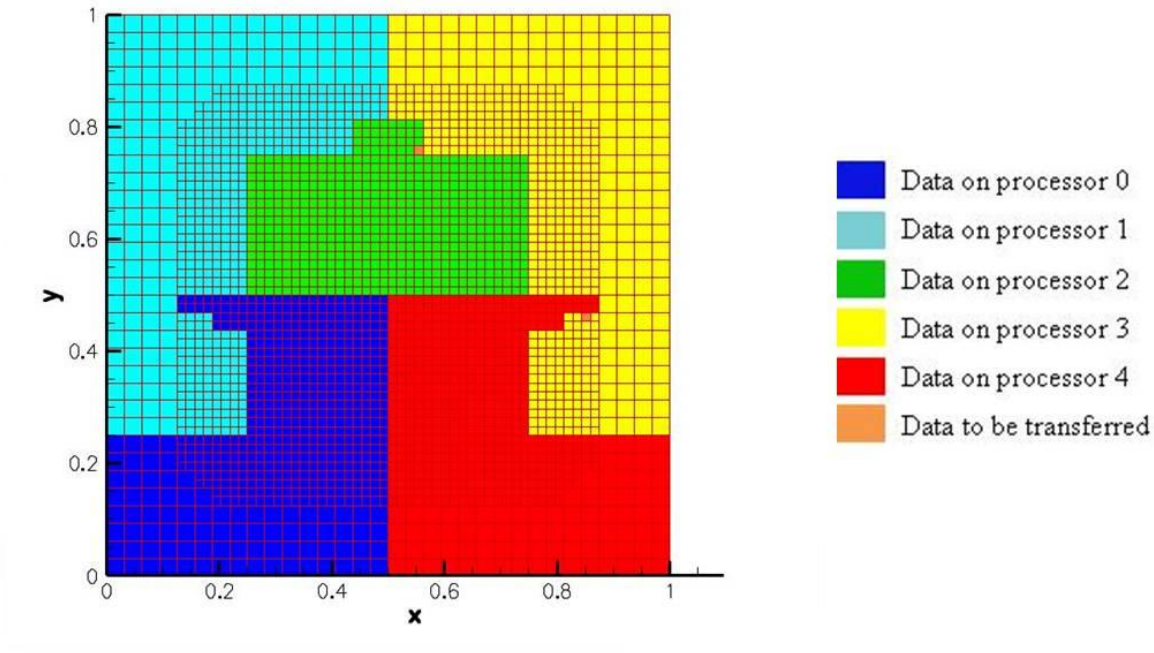

FIGURE 10. Cells to be transferred for parallel coarsening.

3. Embedding of Parallel Multiscale Library into the Quadflow Solver

The finite volume solver Quadflow [8, 9] has been designed to handle (i) unstructured grids composed of polygonal(2D)/polyhedral(3D) elements [7] and (ii) block-structured grids where in each block the grid is determined by local evaluation of B-Spline mappings [29]. Therefore the solver can handle grids provided by an external grid generator. However, grid adaptation is only available for block-structured grids, where in each block the grid is locally refined using the concept of multiscale-based grid adaptation [30].

Note that the flow solver and the multiscale-based grid adaptation have totally different algorithmic requirements: on one hand, there is a finite volume scheme working on arbitrary, unstructured discretisations. On the other hand, there is the multiscale algorithm assuming the existence of hierarchies of structured meshes. The flow solver module is face-centred, since the central item is the computation of the fluxes at the cell faces, while

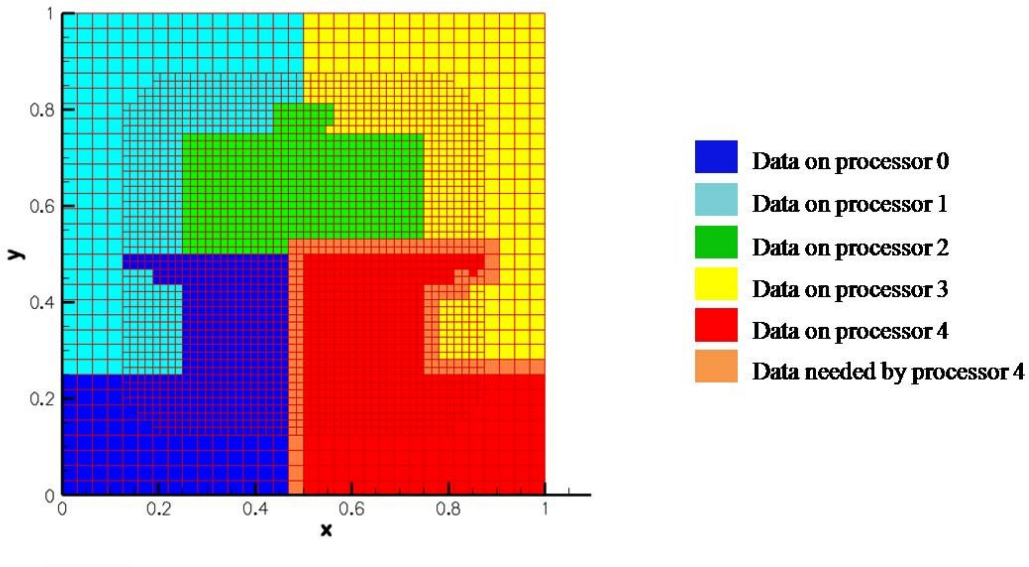

FIGURE 11. Cells to be transferred for parallel details computations.

the adaptation module is cell-centred, analysing and manipulating cell averages. Moreover, the data structures used in the two parts are also different: while for the adaptation part a special implementation of hash maps is used, see Section 1.3, for the flow solver module the FORTRAN style data structures remained optimal. The link between these two modules is done by a data conversion algorithm, which organises all the data communication — the transfer of the conservative variables, volumes, cell centres, the registration of the knots, the construction of the faces and determination of their neighbouring cells and nodes — between the two modules in a connectivity list.

In order to embed the parallelised version of the multiscale library into Quadflow, the transfer had to be adjusted. In the serial case, a special treatment is required for some cells located at the physical boundary or at the far field boundary, cf. [29]. In parallel, special attention is also needed for the cells at the partition's boundary. Since the adaptive mesh is determined at runtime, as well as the partition, and since the adaptive mesh and implicitly the shape of the elements of the partition could change at any time step, there is no way of knowing in advance which cells are located on a processor's boundary. This implies that the partition boundary on each processor should be reconstructed each time the connectivity list is built, in order to transfer the cells on the boundary to the neighbour processors, see Figure 12. This way, the partitions' boundary faces and the flux at these faces can be properly computed on each processor.

4. Numerical Results

First of all, we investigate in Section 4.1 the performance of the parallelised multiscale library. For this purpose, we focus on the multiscale transformation (MST), i.e., the encoding of the data. Note that a complete cycle of grid adaptation also includes thresholding, prediction, grading and decoding. In Section 4.2 we then present numerical simulations of the behaviour of a Lamb-Oseen vortex using the parallelised Quadflow solver.

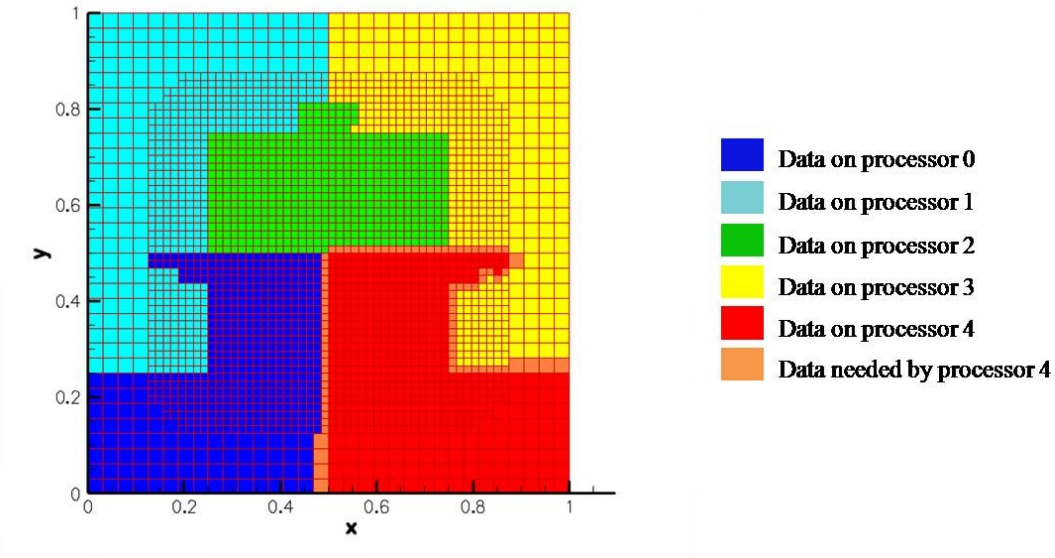

FIGURE 12. Parallel Transfer and Conversion.

No. of procs.	Initial workload	MST time [CPU s]	Transfer time [CPU s]	No. of cells sent	No. of cells received
1	437236	10.21580	0	0	0
2	218618	5.55433	0.066706	2020	2020
3	145746	3.45852	0.170083	5985	5975
4	109309	2.81636	0.125800	2048	2048
5	87448	2.13640	0.127011	10826	10706
6	72876	1.78998	0.126228	7240	7190
7	62464	1.55737	0.133888	9097	9078
8	54658	1.47147	0.103048	6899	6954
9	48588	1.33229	0.105228	7112	7068
10	43729	1.23401	0.135434	8856	8791

TABLE 1. Performance study for parallel multiscale transformation.

4.1. **Performance Study for Multiscale Transformation**

The performance of the multiscale based grid adaptation has been investigated by means of a data set corresponding to a locally refined grid that consists of 437236 cells. The underlying grid hierarchy is determined by $L = 10$ refinement levels and a coarse grid discretisation of 8×8 cells. For this configuration, we performed the adaptation process using an increasing number of processors.

The experiments were performed on a Sun Fire X4600 system, with 2 AMD Opteron 885 nodes having 8 sockets per node (a total of 16 processors), 32 GB memory and a high speed low latency network (InfiniBand) for parallel MPI and hybrid parallelised programs.

Tableau 1 shows the results of these experiments with respect to the different number of processors mentioned in the first column. The second column contains the values for the initial workload, i.e. the number of cells on each processor. Columns 3 and 4 show the times measured when running one MST step and the time spent in the MPI_Isend and MPI_Recv routines, respectively. The number of cells sent by one processor during this MST step is shown in the fifth column, while the sixth column contains the number of cells received by the same processor. These data are visualised in Figures 13–16.

After the partitioning is done, each processor is getting a number of cells equal to the total number of cells in the adaptive grid divided by 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. Better performance and good scaling may be observed in column 3 as the number of processors increases (see also Figure 13). When it comes to the transfer of cells between neighbour processors, the number of entries sent by one processor is different from the one received on most processor configurations chosen due to the adaptive grid: one processor might have more finer cells sitting on the partition boundary than its neighbours have on the other side of the boundary, which gives the difference in the number of ghost cells needed to be transferred from one processor to the other. The number of processors chosen also influences the shape of the elements of the partition created at runtime. Thus, having a symmetric adaptive grid on a single processor and choosing to run a parallel computation on 2 or 4 processors might lead to a symmetric partitioning of the grid on all refinement levels. This implies the minimisation of the interprocessor communication. The consequences of this symmetry can be observed in Tableau 1, where the minimum number of cells to be transferred is achieved on 2 processors. Similarly, the computation on 4 processors also gives few entries to be transferred in comparison with the rest of the configurations tested. The fact that a symmetric partitioning is obtained on 2 and 4 processors is also emphasised when inspecting the number of cells sent and received across the partition boundary: here the number of cells sent to the neighbours is equal to the number of cells received. The opposite is observed on a configuration of 5 processors, where the number of cells on the partition's boundary reaches the maximum value and hence increases the interprocessor communication. A partitioning on 5 processors is shown in Figure 9, where the asymmetry of the partitions is obvious.

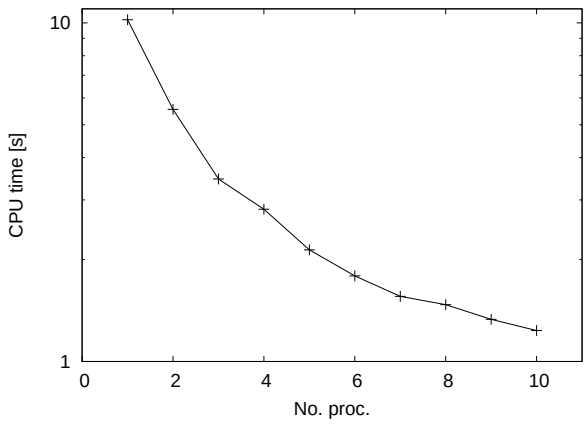

FIGURE 13. CPU time for performing MST.

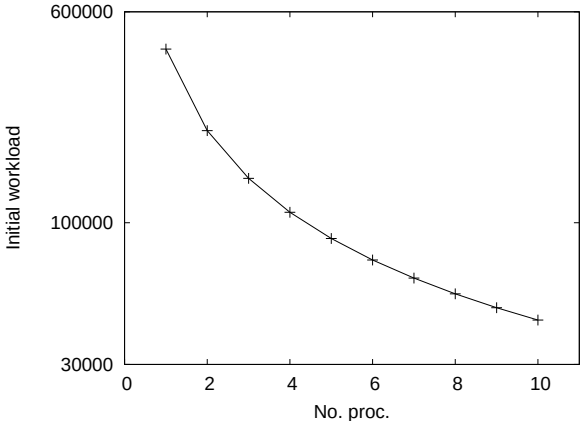

FIGURE 14. Amount of data on each processor.

Speedup. It is a well-known fact that the speedup rate does not tend to infinity with an increasing number of processors. In order to predict a reasonable number, Amdahl's law [1] can be used. It states that for a given problem the maximal speedup of a computation on p processors is bounded by $s_{\max} \leq \frac{p}{1+f(p-1)}$, where f is the

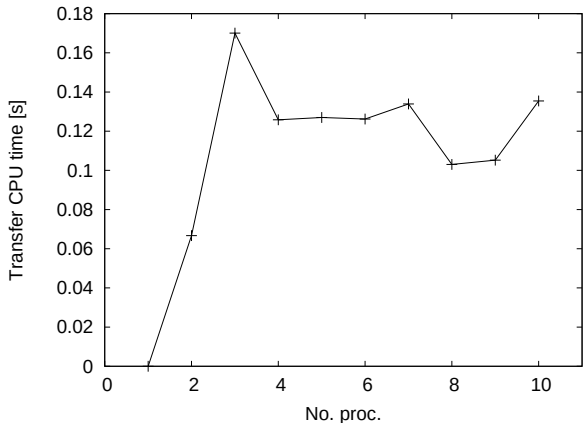

FIGURE 15. CPU time for performing data transfer.

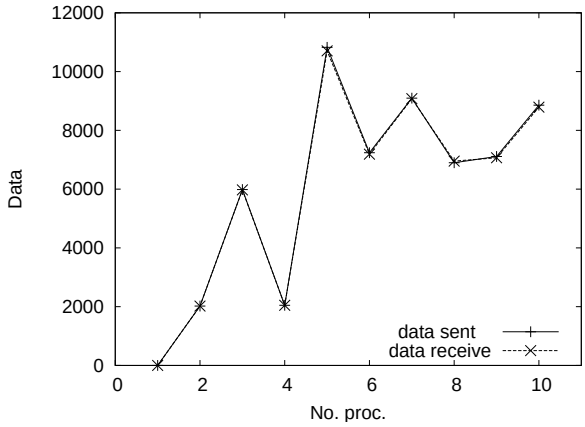

FIGURE 16. Amount of data to be sent and received between processors.

fraction of the runtime of the code that is not parallelised. Note that the parameter f gives insight whether it would be suitable to run the code on massively parallel architectures or not.

In Figure 17, we compared the scaling of our experiment with Amdahl's law assuming that the maximum speedup was measured. Using a nonlinear least squares fit, we were able to estimate the fraction of the code that is not parallelised and obtained $f = 0.0203 \pm 0.0017$.

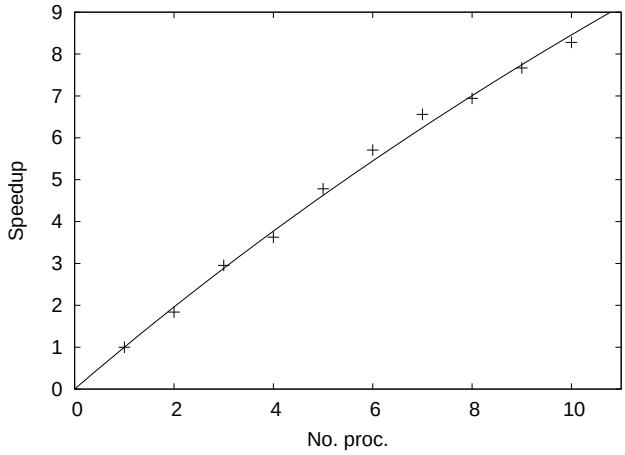

FIGURE 17. Speedup rates (+) and comparison with Amdahl's law (solid line).

Having a fraction of 2% of the program not parallelised, according to Amdahl's law, the maximum speedup that could be reached for this fixed configuration would be 50. However, Amdahl's law doesn't take into account that the fraction of the serial parts can be reduced by scaling the problem to the number of processors. So, for

a fixed configuration, infinite speedup cannot be achieved, but when the size of the problem grows, speedup could also be expected for an increasing number of processors.

4.2. **Application**

The system of vortices in the wake of airplanes continues to exist for a long period of time. It is possible to detect wake vortices as far as 100 wing spans behind the airplane, which are a hazard to following airplanes. In the framework of the collaborative research centre SFB 401, one goal was to induce instabilities into the system of vortices to accelerate their collapse. The effects of different measures taken in order to destabilise the vortices have been examined in a water tunnel. A model of a wing was mounted in a water tunnel and the velocity components in the area behind the wing were measured using particle image velocimetry. It was possible to conduct measurements over a length of 4 wing spans. The experimental analysis of a system of vortices far behind the wing poses great difficulties due to the size of the measuring system. Numerical simulations are not subject to such severe constraints and therefore Quadflow is used to examine the behaviour of vortices far behind the wing. To minimise the computational effort, the grid adaptation adjusts the refinement of the grid with the goal to resolve all important flow phenomena, while using as few cells as possible.

In the present study an instationary, quasi-incompressible, inviscid fluid flow described by the Euler equations is considered. A first assessment is presented to validate the ability of Quadflow to simulate the behaviour of the wake of an airplane. A velocity field based on the experimental measurements is prescribed as boundary condition in the inflow plane. The circumferential part of the velocity distribution $v_\Theta(r)$ is described by a Lamb-Oseen vortex according to

$$v_\Theta(r) = \frac{\Gamma}{2\pi r}\left(1 - e^{-\left(\frac{r}{d_0}\right)^2}\right). \tag{4}$$

The axial velocity component in the inflow direction is set to the constant inflow velocity of the water tunnel. The two parameters of the Lamb-Oseen vortex, circulation Γ and core radius d_0 are chosen in such a way that the model fits the measured velocity field of the wing tip vortex as close as possible. The radius r is the distance from the centre of a boundary face in the inflow plane to the vortex core.

Instead of water, which is used as fluid in the experiment, the computation relies on air as fluid. The inflow velocity in the x-direction u_∞ is computed to fulfil the condition that the Reynolds number in the computational test case is the same as in the experiment. The experimental conditions are a flow velocity $u_w = 1.1\,\mathrm{m/s}$ and a Reynolds number $Re_w = 1.9 \cdot 10^5$. From the condition $Re_{air} = Re_w$ the inflow velocity in x-direction can be determined as $u_\infty = 16.21\,\mathrm{m/s}$. For consistency the circumferential velocity v_θ has also been multiplied by the factor $\frac{u_\infty}{u_w}$. The velocity of the initial solution is set to parallel, uniform flow $u_0 = u_\infty$, $v_0 = w_0 = 0.0$.

The computation has been performed on 16 Intel Xeon E5450 processors running at 3 GHz clock speed. The computational domain matches the experimental setup whose dimensions are $l = 6\,\mathrm{m}$ in the x-direction, $b = 1.5\,\mathrm{m}$ in the y-direction and $h = 1.1\,\mathrm{m}$ in the z-direction. The boundaries parallel to the x-direction have been modelled as symmetry walls. This domain is discretised by a coarse grid with 40 cells in flow-direction, 14 cells in y-direction and 10 cells in z-direction, respectively. The number of refinement levels has been set to $L = 6$. With this setting the vortex can be resolved on the finest level by about 80 cells in the y-z-plane.

Since Quadflow solves the compressible Euler equations a preconditioning for low Mach numbers has to be applied. The preconditioning is used in a dual-time framework wherein only the dual time-derivatives are preconditioned and used for the purposes of numerical discretisation and iterative solution, cf. [33]. The spatial discretisation of the convective fluxes is based on the AUSMDV(P) flux vector splitting method [19]. For time integration the implicit midpoint rule is applied. In each timestep the unsteady residual of the Newton iterations is reduced by four orders of magnitude. The physical timestep is set to $\Delta t = 5 \cdot 10^{-5}\,\mathrm{s}$ which corresponds to a maximum CFL-number of about $CFL_{max} = 28.0$ in the domain. The grid is adapted after each timestep. After every 100th timestep the load-balancing is repeated.

To guarantee a sufficiently fine grid to resolve the vortex properly at the start of the computation, the grid on the inflow plane is refined to the maximum level, see Figure 18. Due to this procedure the first grid contains 384000 cells. When the information at the inlet has travelled through the first cell layer, the forced adaptation

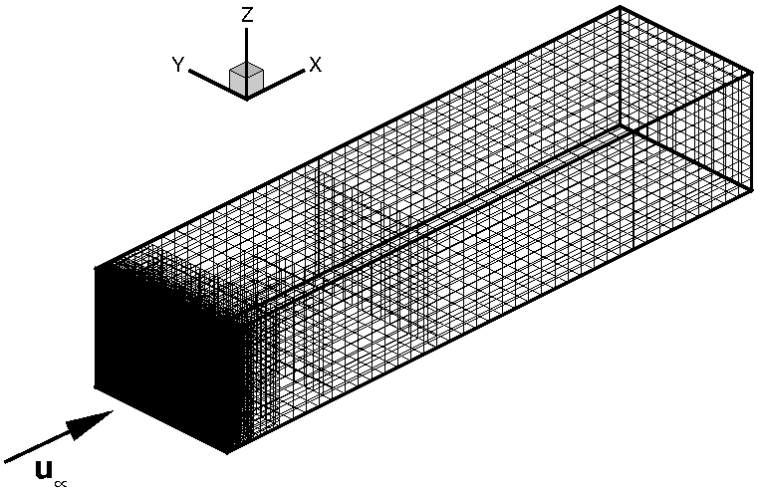

FIGURE 18. Initial computational grid.

of the cells at the inlet is not necessary anymore. From there on the grid is only adapted due to the adaptation criterion based on the multiscale analysis.

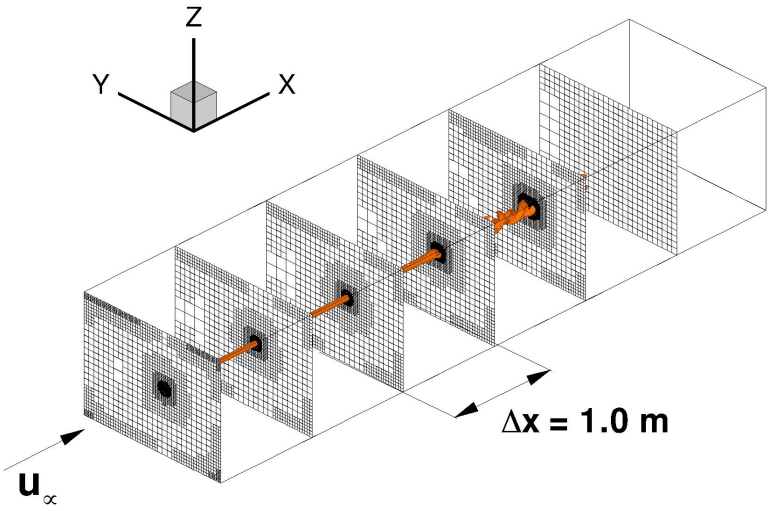

FIGURE 19. Slices of the computational grid after 5466 timesteps at six different positions and the distribution of $\lambda_2 = -3$.

After 5466 timesteps, which corresponds to a computed real time of $t = 0.27\,\mathrm{s}$, the grid contains 787000 cells. Figure 19 shows six cross sections of the mesh, which are equally spaced in x-direction with distances $\Delta x = 1.0\,\mathrm{m}$. In addition, the isosurface of the λ_2-criterion with the value $\lambda_2 = -3$ is also presented. The λ_2 criterion has been proposed by Jeong et al. [26] to detect vortices. A negative value of λ_2 identifies a vortex, whereas the smallest of these negative values marks the core of the vortex. As can be seen from Figure 19,

the vortex is transported through the computational domain. The locally adapted grid exhibits high levels of refinement only in the vicinity of the vortex. A more detailed view of the grid for the cross sections at $x = 0.0\,\mathrm{m}$ and $x = 2.0\,\mathrm{m}$ is presented in Figure 20. It can be seen that only near the vortex core the grid is refined up to the maximum level. We conclude that Quadflow is well suited to examine the instationary behaviour of a vortex. In particular the complexity reduction due to the grid adaptation makes it possible to perform the computations in reasonable time.

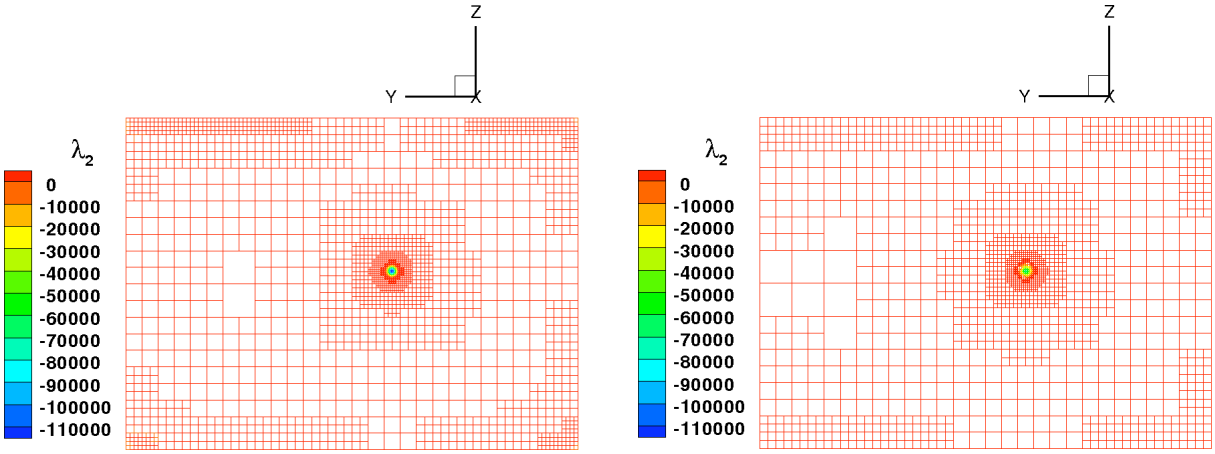

FIGURE 20. Slices of the computational grid at two different positions in the x-direction, the grid colour is consistent with the value of λ_2. Slice of the computational grid at $x = 0.0\,\mathrm{m}$ (left) and $x = 2.0\,\mathrm{m}$ (right).

Finally we perform a parameter study for the application at hand. Since the whole computation takes quite long, we confine our performance study to ten time steps only, restarting the computations after having performed the first ten timesteps. Then the vortex is located inside the domain and, hence, the grid at the inflow boundary can be adaptively refined. The initial grid then consists of 267911 cells, after ten time steps it contains 204239 cells. All computations have been performed on the new 192 node cluster from Sun Microsystems with Intel Xeon X5570 CPUs operated by the Center for Computing at RWTH Aachen.

In Table 2 we summarise the resulting CPU times needed by the flow solver, the multiscale transformation, the data transfer, the rebalancing and the total time, respectively. The results are also displayed in Figure 21, where the sums of the CPU times for the multiscale transformation, the data transfer and the rebalancing are shown in a single graph, because most of the communication is concentrated in these parts rather than in the flow solver. Therefore the flow solver scales almost perfectly whereas in the other parts of the code the communication costs become increasingly dominant with higher number of processors.

Again, we estimate the fraction f of the code that is not parallelisable. This is a crucial information that plays an important role for selecting the appropriate parallel hardware architecture. For this purpose, we apply Amdahl's law as model function. In order to apply this law, we need to compute the speedup rates with respect to the CPU time of a single processor computation. We estimate this value $T_1 = 4.39e+03\,\mathrm{s}$ from the first four time measurements for 4,8,16 and 32 processors by fitting a power function to the data using nonlinear least-squares analysis. From T_1 we determine the speedup rates shown in Figure 22. Another nonlinear least-squares analysis using Amdahl's law as model function yields $f = 0.013 \pm 0.001$ for the overall times measured. In the limit, as the number of processors goes to infinity, Amdahl's law tends to the maximal theoretical speedup, which is $1/f$. If f is too big, it is not reasonable to choose an architecture with too many processors.

No. of procs.	Flow solver [CPU s]	MST [CPU s]	Transfer [CPU s]	Rebalancing [CPU s]	Total [CPU s]
4	1.38e+03	2.80e+01	1.68e+01	2.97e-01	1.46e+03
8	8.39e+02	2.03e+01	1.08e+01	1.62e-01	9.05e+02
16	4.92e+02	7.77e+00	3.87e+00	9.44e-02	5.09e+02
32	2.19e+02	3.93e+00	1.49e+00	7.50e-02	2.27e+02
64	1.21e+02	5.17e+00	1.27e+00	2.08e-01	1.29e+02
128	6.78e+01	1.13e+01	1.88e+00	4.86e-01	8.30e+01
256	4.32e+01	2.56e+01	3.79e+00	1.11e+00	7.63e+01

TABLE 2. Performance study for the vortex simulation (10 timesteps).

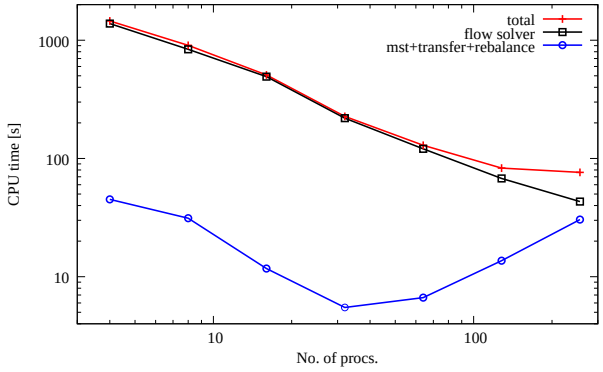

FIGURE 21. CPU time in performance study.

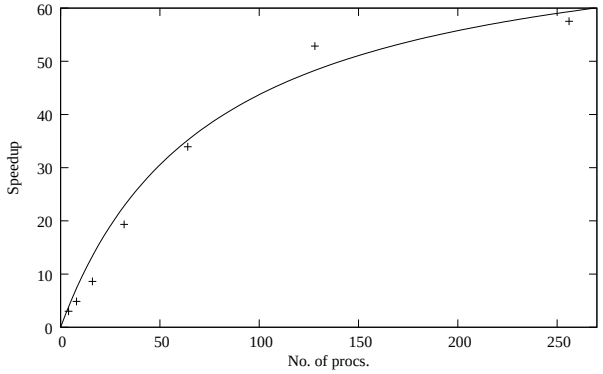

FIGURE 22. Speedup rates (+) and comparison with Amdahl's law (solid line).

5. CONCLUSION

We have presented the current state of development of the parallelised multiscale-based grid adaptation library and its embedment into the Quadflow solver. First, the basic ingredients of the multiscale library have been described and then the load-balancing issue for single-block Cartesian grids has been discussed by means of the discrete Hilbert space-filling curve. The parallel algorithms for the grid adaptation and the interprocessor communication implied have also been discussed. We concluded with presenting some performance studies that showed a good scaling of the parallel multiscale-based grid adaptation and first 3D parallel results.

The present state of implementation still lacks some parallel features that are subject to future development. In a first step, the extension of the partitioning strategy – until now applied only to Cartesian grids – to curvilinear grids is straightforward. Then, the parallelisation of adaptive multiblock grids has to be achieved. This could be realised by different space-filling curves running through each block and an appropriate partitioning strategy to determine whether a specific block should be distributed to different processors or not. Special care has to be taken of cells sitting at the blocks' boundaries on neighbouring processors, which need to be exchanged between processors in order to properly compute the flux at the faces on the blocks' boundaries.

We have to mention here that these were the first parallel results and further optimisation of the code is still to be considered in the future. In this context, issues such as cache-awareness have to be taken into account. This could require further modifications of the present data structures. For instance, Zenger et al. [23] use

space-filling curves to build up stacks that are processed linearly, which turned out to considerably reduce cache misses.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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, Basel, 1997. Birkhäuser Press.

[7] F. Bramkamp. *Unstructured h-Adaptive Finite-Volume Schemes for Compressible Viscous Fluid Flow*. PhD thesis, RWTH Aachen, 2003. `http://sylvester.bth.rwth-aachen.de/dissertationen/2003/255/03_255.pdf`.

[8] 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, Berlin, 2003. Springer.

[9] F. Bramkamp, Ph. Lamby, and S. Müller. An adaptive multiscale finite volume solver for unsteady an steady state flow computations. *J. Comput. Phys.*, 197(2):460–490, 2004.

[10] 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.

[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] 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.

[14] 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, Basel, 2002. Birkhäuser.

[15] F. Coquel, Q.L. Nguyen, M. Postel, and Q.H. Tran. Local time stepping applied to implicit-explicit methods for hyperbolic systems. Technical report, Laboratoire Jacques-Louis Lions, Paris VI, 2007.

[16] 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.

[17] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2nd edition, 2001.

[18] M. Domingues, O. Roussel, and K. Schneider. On space-time adaptive schemes for the numerical solution of partial differential equations. *ESAIM: Proceedings*, 16:181–194, 2007.

[19] J. Edwards and M.S. Liou. Low-diffusion flux-splitting methods for flows at all speeds. *AIAA Journal*, 36:1610–1617, 1998.

[20] 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`.

[21] 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.

[22] 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.

[23] F. Günther, M. Mehl, M. Pögl, and Ch. Zenger. A cache-aware algorithm for pdes on hierarchical data structures based on space-filling curves. *SIAM J. Sci. Comput.*, 28(5):1634–1650, 2006.

[24] A. Harten. Adaptive multiresolution schemes for shock computations. *J. Comput. Phys.*, 115:319–338, 1994.

[25] A. Harten. Multiresolution algorithms for the numerical solution of hyperbolic conservation laws. *Comm. Pure Appl. Math.*, 48(12):1305–1342, 1995.

[26] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285:69–94, 1995.

[27] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Supercomputing*, 1998.

[28] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71 – 85, 1998.

[29] 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`.

[30] S. Müller. *Adaptive Multiscale Schemes for Conservation Laws*, volume 27 of *Lecture Notes on Computational Science and Engineering*. Springer, Berlin, 2002.

[31] S. Müller, Ph. Helluy, and J. Ballmann. Numerical simulation of a single bubble by compressible two-phase fluids. *Int. J. Numer. Meth. Fluids*, 2009. DOI 10.1002/fld.2033.

[32] S. Müller and A. Voss. A Manual for the Template Class Library `igpm_t_lib`. IGPM–Report 197, RWTH Aachen, 2000.

[33] S.A. Pandya, S. Venkateswaran, and T.H. Pulliam. Implementation of preconditioned dual-time procedures in overflow. *AIAA Paper 2003-0072*, 2003.

[34] 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.

[35] O. Roussel and K. Schneider. Adaptive multiresolution method for combustion problems: Application to flame ball-vortex interaction. *Computers and Fluids*, 34(7):817–831, 2005.

[36] 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.

[37] O. Roussel, K. Schneider, A. Tsigulin, and H. Bockhorn. A conservative fully adaptive multiresolution algorithm for parabolic PDEs. *J. Comput. Phys.*, 188(2):493–523, 2003.

[38] H. Sagan. *Space-Filling Curves*. Springer, Berlin, 1st edition, 1994.

[39] S. Schamberger and J.-M. Wierum. Partitioning finite element meshes using space-filling curves. *Future Gener. Comput. Syst.*, 21(5):759–766, 2005.

[40] B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, 1997.

[41] 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.

[42] G. Zumbusch. *Parallel multilevel methods. Adaptive mesh refinement and loadbalancing*. Advances in Numerical Mathematics. Teubner, Wiesbaden, 2003.