PARALLEL DATA STRUCTURES AND ALGORITHMS FOR HIGH-DIMENSIONAL STRUCTURED ADAPTIVE MESH REFINEMENT

MAGNUS GRANDIN† AND SVERKER HOLMGREN†

Abstract. Numerical solution of high-dimensional partial differential equations often results in challenging computations. Using a uniform discretization of the spatial domain quickly becomes untractable due to the exponential increase in problem size with dimensionality. However, by employing a spatially adaptive discretization scheme the number of grid points can often be reduced significantly. In this note we describe a parallel version of an earlier presented adaptive scheme which generates the mesh by recursive bisection, allowing mesh blocks to be arbitrarily anisotropic to allow for fine structures in some directions without over-refining in other directions. We extend the serial framework by presenting parallel algorithms for organizing the mesh blocks in a distributed kdtree and the necessary operations for implementing structured adaptive mesh refinement on a parallel computer system.

1. Introduction. Accurate and efficient numerical simulation of time-dependent partial differential equations (PDEs) can be a demanding task, especially for high-dimensional problems. Such PDE problems appear in a range of research applications, such as quantum dynamics [18], systems biology [6], plasma physics [11] and financial mathematics [12]. The general class of problems that are considered are d-dimensional, time-dependent, linear PDEs

$$\frac{\partial u}{\partial t} = \hat{P}(\mathbf{x}, t)\, u, \tag{1.1}$$

where $\hat{P}$ is the continuous linear operator describing the dynamics in the model and $\mathbf{x}$ is a d-dimensional vector of spatial variables. Discretizing (1.1) in space transforms the PDE into a large semi-discrete system of ordinary differential equations (ODEs), an approach generally referred to as the method of lines.

In order to solve PDE problems like (1.1) numerically, massive-scale parallel computing resources are often needed. Then, the computational algorithms and implementations must be suitably parallelized to in order to ensure that the computational potential of these systems is realised.

One important means of reducing the computational complexity and memory requirements for PDE solvers is given by structured adaptive mesh refinement (SAMR). By dynamically adjusting the resolution of the computational mesh to match features in the solution or the computational domain, different scales of resolution can be represented and very coarse grids can be used in "uninteresting" parts of the domain [3]. For problems in two and three dimensions, there are efficient algorithms and data structures available, relying on quad/octrees for structuring the mesh blocks [7, 8]. Efficient parallel algorithms for these types of meshes have also been derived and efficiently implemented [3, 15].

However, extending quadtrees and octrees to higher dimensional trees is problematic since they yield a fan-out of $2^{\mathcal{D}}$ nodes at every branch, which leads to an exponential increase in the potential number of tree nodes to handle. In previous work [9] an alternate approach based on a kdtree [1] representation, corresponding to a recursive bisection of mesh blocks, was presented. By encoding the leaf nodes of the

†Division of Scientific Computing, Dept. of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden (magnus.grandin@it.uu.se, sverker.holmgren@it.uu.se)

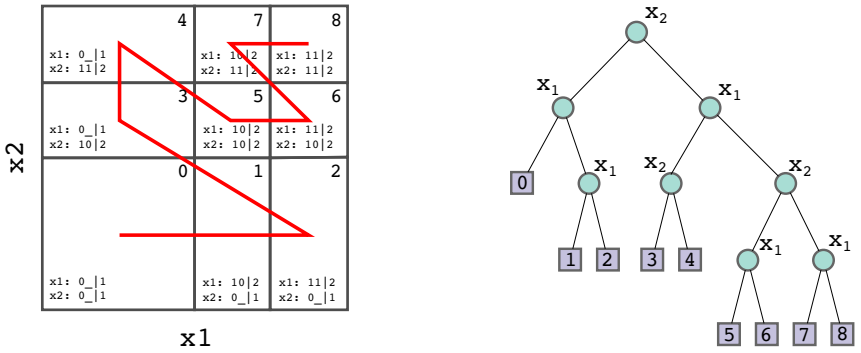


FIG. 2.1. *Anisotropic mesh decomposition with coordinates and refinement levels indicated for each node. (left) Location codes marked per dimension for each mesh node on the form* `coord|level`*. Linear order of nodes indicated in red, starting in the lower left corner. (right) Corresponding node hierarchy, internal nodes specifying the split dimension at each level.*

tree such that they store information regarding their size and location in the grid, the mesh can be represented as a linear array of leaf nodes (a *linear kdtree*). The linear order is combined with a node index map in order to obtain a hierarchichal Morton order, modifying the bit interleaving to correspond to the anisotropic refinement. This note extends the previous work presented in [9] by describing how the kdtree can be distributed among multiple processes and how the basic operations needed for the adaptive mesh refinement can be parallelized to allow for implementation on modern large-scale computer systems.

2. Serial framework. In this section we briefly review the fundamental serial algorithms and data structures that we build upon in our parallel framework. For a comprehensive description of these entities, see [9].

2.1. Location codes. We consider hyper-rectangular domains where mesh entities constitute $\mathcal{D}$-dimensional orthotopes. The *location code* of an orthotope is a unique identifier, encoding the orthotope's location and refinement levels with respect to its anchor corner (the leftmost/lowermost corner in each direction). See Figure 2.1 for an example node hierarchy and mesh with location codes explicitly indicated.

2.1.1. Construction. The coordinates and refinement levels of an orthotope are completely integer based and correspond to the orthotope's location in a unit hypercube domain. This allows for efficient bitwise integer operations and deterministic mesh behaviour. The actual coordinates of mesh blocks in the domain can be obtained by scaling the unit hypercube coordinates to the real domain boundaries. Location codes are constructed and stored as follows [3]: Each coordinate is represented by $\mathcal{B}$ bits; thus, in total $\mathcal{DB}$ bits are required for the entire coordinate set of an orthotope. Subdivision of an orthotope at level l in dimension x_i updates the lth bit in the ith coordinate accordingly. The leftmost bit of a coordinate is the most significant and corresponds to the first subdivision in the corresponding dimension, the second leftmost bit corresponds to the second subdivision, and so forth. The bits that are below the refinement level of an orthotope are ignored (set to 0). Thus, the integer size $\mathcal{B}$ relates to the maximum level of refinement in any dimension and requires only $\lceil \log_2 \mathcal{B} \rceil$ bits per dimension for storage of the refinement levels.

2.1.2. Atoms. The smallest domain units, corresponding to orthotopes of maximum refinement level in all dimensions, are referred to as atoms. Other authors refer to similar mesh entities as smallest or least descendants [3, 15]. Of particular interest are the first and the last atoms of an orthotope, since they constitute the span of the orthotope over a region in the domain. Moreover, the corner atoms of an orthotope are useful in deriving search keys for finding neighboring orthotopes.

2.1.3. Surfaces. Considering orthotopes that are of maximum refinement only in a subset of the dimensions, any lower-dimensional mesh entity can be described. An m-face surface (or m-surface) is an m-dimensional mesh element in $\mathcal{D}$-dimensional space, such that $\mathcal{D}-m$ of the orthotope's refinement levels are of maximum refinement.

2.1.4. Boundary elements. The complete set of boundary elements of an orthotope o can be described in terms of surfaces. By specifying dimension-wise directions for the surface of interest as a $\mathcal{D}$-vector f of values $\{-1, 0, 1\}$, all dimensions in which the corresponding element of f is 0 will be of the same extent as o and in the remaining dimensions the surface will be of maximum refinement. For example, if all elements of f but one are 0, we will get the face to the left/right (corresponding to $-1/1$) in the dimension in which f was non-zero. If all elements of f are non-zero, the resulting surface is the corresponding corner of o.

2.1.5. Neighboring orthotopes. A logical neighbor orthotope of an orthotope o is defined as an orthotope of identical size and shape, adjacent to o across some boundary. The primary use of this conceptual entity is to discover the insulation layer of an orthotope, as decribed in Section 4.2.

2.2. Hierarchical Morton order. Standard Morton order [13] is not well defined on anisotropic hyperrectangular domains. The regular bit interleaving will be broken across block boundaries between anisotropic elements, which leads to interrupted linear sequences and search paths. Consider the example in Figure 2.2(a). Here, the ordering of atoms (the smallest black boxes) clearly violates the order of the structure of the mesh blocks, which is an undesirable feature. A binary search for the mesh block overlapping the element highlighted in red in Figure 2.2 might be led astray by the disrupted linear sequence. If the bit interleaving is modified according to the hierarchy of split sequences, the resulting *hierarchical Morton order* [9] will lead to a more consistent linear ordering, corresponding to Figure 2.2(b). The linear sequences emerging from this ordering will be coherent in all cases and preserve locality of refinement. However, a node index map must be explicitly stored in order to derive the proper ordering of related elements. We represent this as a binary tree stored in depth-first order [14], where only two integers are required for each node; the split dimension and an offset to a nodes' right child. The resulting data structure is small and stored in linear storage for fast traversal.

2.3. Search. We employ three different algorithms for searching; binary search for the mesh element exactly matching a given orthotope (`binarySearchMatching()`), binary search for a mesh element (not guaranteed to be unique) overlapping a given orthotope (`binarySearchOverlapping()`), and finding the complete set of mesh elements overlapping a given orthotope (`findOverlappingNodes()`) [9]. All of these search routines are completely local; no parallel search is performed. Instead, as will be detailed in the subsequent sections, the relevant remote information is gathered in each process through collective communication and searching is performed locally within the collected search space.

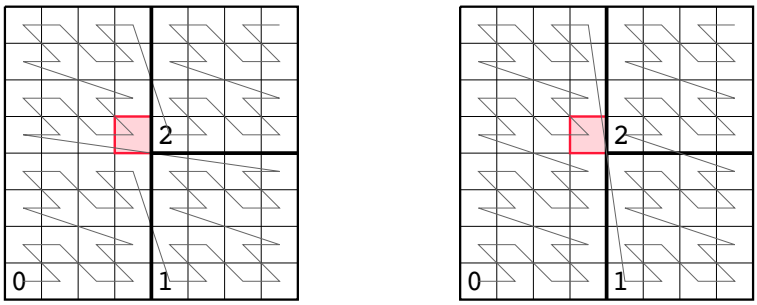


FIG. 2.2. *An anisotropic mesh with 3 blocks, showing the linear order of atoms following different indexing algorithms. (a) Regular Morton order; the linear chain of indices in block 0 is disrupted by block 1. (b) Hierarchical Morton order; the bit-interleaving and corresponding ordering has been modified such that all blocks cover a consecutive sequence of indices.*

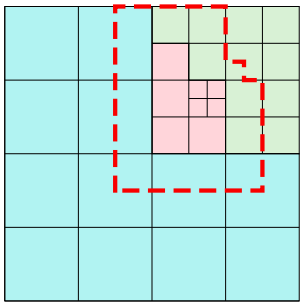
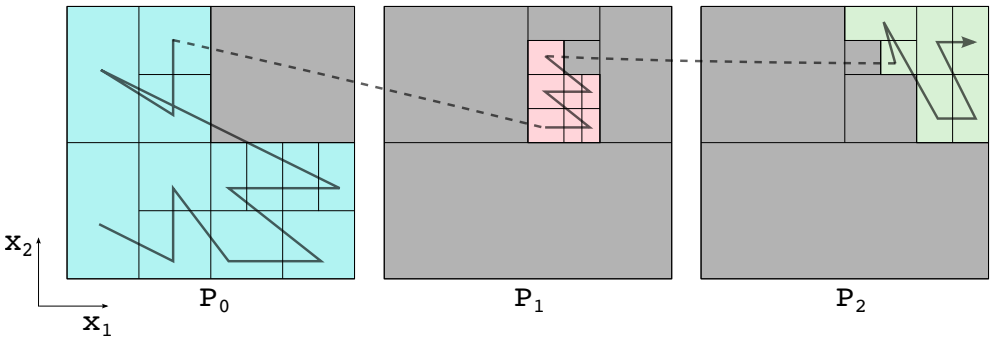


FIG. 3.1. *Example where the insulation layer around the process partition colored pink is indicated. The insulation layer is constructed by collecting the set of logical neighbors lying outside each process' orthotope range.*

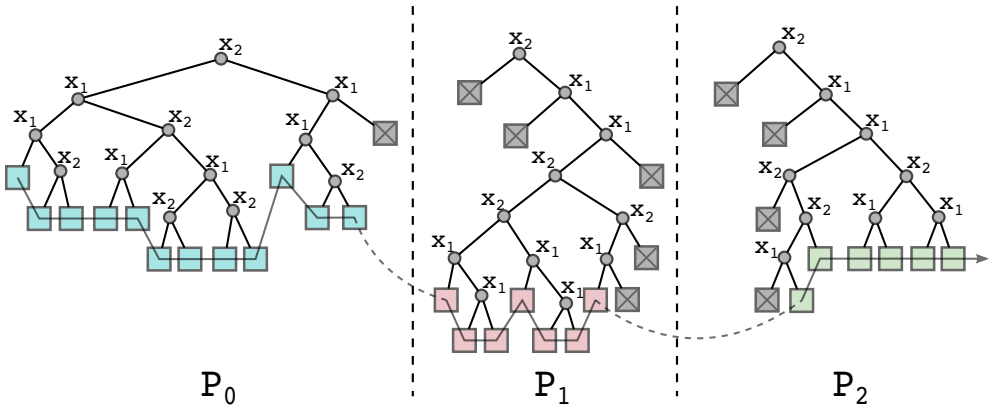
3. Distributed kdtree. We now describe the parallel mesh refinement framework, starting by constructing a parallel distributed kdtree by partitioning the global tree and letting each process manage its own local subtree. Here, no process has knowledge of the entire mesh at any time. Each process only needs to store a pruned local tree, and for performing the adaptive refinement only the overlap with its *insulation layer* [15] is required. Figure 3.1 depicts an example of the insulation layer for a mesh region (colored in pink). Furthermore, in the algorithms presented below the exchange of mesh information between neighboring processes is limited to a small constant number of communication steps per mesh update.

3.1. Local pruned subtrees. A local subtree represents (a subset of) a process' local mesh nodes and holds a separate node index map corresponding to those nodes. The node index map of a local subtree is pruned to only account for the nodes in the subtree; thus, all potential branches of the node index map that do not lead to a node prevalent in the tree are removed.

3.2. Data distribution. Each process is dynamically assigned ownership over a local subtree of mesh nodes. Thus, all processes have their own partial view of the global mesh as illustrated in Figure 3.2. The figure depicts an example mesh and the corresponding index map as distributed among three processes. The coloring of



(a) Distributed mesh, coloring indicates partitioning.



(b) Distributed node index map.

FIG. 3.2. *Parallel anisotropic mesh with corresponding node hierarchy and linear order explicitly indicated.*

mesh blocks indicates process ownership and the crossed grey boxes indicates pruned branches. Merging the local trees of all processes results in the complete global mesh.

3.3. Data migration. When mesh nodes are migrated from one process to another the transmitted data is stored in a subtree, and by merging the received subtree into the local subtree the receiving node will have updated information including the received mesh blocks. The linear order of leaf nodes is determined from the information in each respective node index map and it is imperative that all processes have the same view of the total order. Refinement and coarsening of mesh nodes is done at leaf level; hence it is a completely local operation and will not affect the global view of the index map for any other process.

4. Parallel adaptive mesh refinement. Serial algorithms for doing refinement, coarsening and 2:1 balancing of mesh blocks were described in [9]. In the parallel framework, each node first essentially employs these algorithms locally with no need for communication among the processes. Once all local updates are completed, a global 2:1 balancing step followed by work load repartitioning ensures that the mesh is globally valid and properly balanced. In order to arrive at a parallel version of the framework in [9], the serial implemention needs to be extended with routines for local pruning and merging of subtrees and with routines for transferring

subtrees between processes. Here, as the local subtrees are stored as linear depth-first search trees, merging them can be done in linear time. Note that, in the algorithms described below, there are two ways in which we need to transfer mesh data; one in which we transfer only mesh hierarchy meta data and one in which we also include the underlying grid data blocks. The latter is only used when migrating mesh blocks between nodes.

4.1. Local adaptation. The local refinement and coarsening operations are performed in-place directly in the array of leaf nodes. During refinement a by-pass queue is used to keep added elements and insert them at the right places. Coarsening does repeated refinement on sequences of nodes, overwriting the left child with its parent (deleting the right child) in order to avoid external storage. In [9], we added a sort-step at the end of refinement in order to enforce a strict order in the interleaving of the Morton indices. One important change needed to employ this scheme in a parallel setting is to avoid sorting after refinement; instead we update the index map in place similarly to the way we handle the leaf nodes. This is possible since the index map nodes are stored in depth-first order. We can avoid sorting during coarsening as well, by replacing the removed nodes in the index map with null-nodes, and compress the map after all updates are done. Avoiding sorting also keeps the split-order of nodes, which is important since a sort would potentially change the global order of nodes. Now all updates are kept local to each process. Local 2:1 balancing proceeds by finding the neighbors around each corner of an orthotope, and marks for refinement every neighbor that is more than two levels coarser than the current node.

4.2. Parallel 2:1 balancing. In order to balance the nodes in a distributed mesh with respect to the 2:1 balance requirement, we propose that a two-stage balancing step is used, combining the ideas of the parallel algorithm presented in [15] and our serial balancing algorithm presented in [9]. By constructing an insulation layer around the partition in each process, iterative communication can be avoided in the global balancing step. As is pointed out in [15], refinement occuring outside the insulation layer cannot transfer to the inside.

The algorithm for parallel balancing of mesh blocks is outlined in Algorithm 1. In the first step of global balancing, each process ensures local balancing. Each process identifies the blocks that share an interprocessor boundary with a remote block and constructs insulation orthotopes for each such boundary element. The insulation orthotopes are communicated among the processes such that each process has information about the actual neighbor blocks that overlap its insulation layer. The mesh is then updated by balancing locally with respect to the insulation layer.

4.3. Load balancing and partitioning. Dynamically refined and moving mesh partitions need continuous repartitioning in order for the computational load to be evenly distributed among the processes. This is a crucial aspect of any dynamic adapitive mesh refinement scheme and the literature offers several ways of acheiving this [3, 4, 5, 10, 15, 16, 17], with different requirements on the communication involved and the quality of the final load distribution. Space-filling curves are commonly used for load balancing purposes, creating a balanced partitioning by dividing the linear sequence in chunks. In our case, the hierarchical Morton order serves as a linearization of the mesh blocks. We propose a non-iterative scheme based on a parallel prefix-sum [2] in order to estimate the load, and a nearest-neighbor communication scheme sending data left/right for repartition of the mesh data. An outline of the procedure is presented in Algorithm 2.

Algorithm 1: `balance(kdtree` $\mathcal{T}$`)`

1: `localBalance`($\mathcal{T}$)
2: Communicate (all-to-all) the local orthotope range of each processor
3: $I \leftarrow$ `interProcessInsulationLayer`($\mathcal{T}$)
4: $Q_{p_i} = [\emptyset],\ \forall p_i \in \mathcal{P}$ ▷ *Lists of insulation orthotopes to send (per process)*
5: **for** $s \in I$ **do**
6: **foreach** $p_i \in \mathcal{P}$ with orthotope range overlapping s **do**
7: Q_{p_i}.`append`(s) ▷ *Append* s *to list of orthotopes to send to* $\mathrm{p_i}$
8: **end**
9: **end**
10: Send lists of insulation orthotopes Q_{p_i} to corresponding processes
11: **foreach** process p' sending insulation layer to p **do**
12: Receive insulation orthotopes into list $L_{p'}$
13: $M_{p'} \leftarrow [\emptyset]$
14: **for** $q \in L_{p'}$ **do**
15: $l \leftarrow$ `findOverlappingNodes`(q)
16: $M_{p'}$.`append`(l) to list of orthotopes to send back to p'
17: **end**
18: $M_{p'}$.`removeDuplicatesAndOverlaps`()
19: Sort and store orthotopes in $M_{p'}$ in $\mathcal{T}_{p'}$
20: Send $\mathcal{T}_{p'}$ to p'
21: **end**
22: **foreach** process p' sending response to p **do**
23: Receive insulation trees from p', store in exterior tree $\mathcal{T}_{ext}$
24: **end**
25: `localBalanceExtended`($\mathcal{T}, \mathcal{T}_{ext}$) ▷ *Balance local tree w.r.t exterior tree.*

5. Conclusion. In this note we have described a conceptual framework for parallelizing the schemes for structured adaptive mesh refinement in [9]. Due to the construction of the serial framework, only a few modifications and additions are needed for parallelization. Furthermore, we believe that it is possible to implement the parallel framework in an efficient way since the algorithms only depend on a small amount of global communication. There is no global data structure to maintain, instead we exploit that mesh updates are completely local and let the global index order be carried around with the data. However, in order to verify that our approach actually works, the parallel code needs to be completed and experiments for realistic PDE problem settings in higher dimensions have to be performed.

Algorithm 2: `partitionLoadAndRebalance(kdtree` $\mathcal{T}$`)`

1: $comm, \mathcal{P}, p \leftarrow$ parallel communicator, communicator size and local process number
2: $node\_counts \leftarrow \mathcal{P}$-vector of process block counts (`all_gather`)
3: $prefix\_node\_counts \leftarrow$ prefix sum of current block distribution, ($\mathcal{P}$-vector)
4: $prefix\_ideal\_counts \leftarrow$ prefix sum of target (ideal) distribution, ($\mathcal{P}$-vector)
5: **for** $p \in 0, \ldots, \mathcal{P}-1$ **do** ▷ *Compute offset between actual and target distributions*
6: $\quad prefix\_offset[p] \leftarrow prefix\_node\_counts[p] - prefix\_ideal\_counts[p]$
7: **end**
8: **if** $(p > 0)$ **and** $prefix\_offset[p-1] > 0$ **then**
9: $\quad recv\_count \leftarrow prefix\_offset[p-1]$
10: $\quad comm$.`send`$(dst : p-1, msg :$ leftmost $recv\_count$ blocks in $p)$
11: **else if** $(p < \mathcal{P}-1)$ **and** $(prefix\_offset[p] < 0)$ **then**
12: $\quad recv\_count = -prefix\_offset[p]$
13: $\quad comm$.`send`$(dst : p+1, msg :$ rightmost $recv\_count$ blocks in $p)$
14: **else if** $(p > 0)$ **and** $(prefix\_offset[p-1] < 0)$ **then**
15: $\quad send\_count = -prefix\_offset[p-1]$
16: $\quad comm$.`send`$(dst : p-1, msg :$ leftmost $recv\_count$ blocks in $p)$
17: **else if** $(p < \mathcal{P}-1)$ **and** $(prefix\_offset[p] > 0)$ **then**
18: $\quad send\_count = prefix\_offset[p]$
19: $\quad comm$.`send`$(dst : p+1, msg :$ rightmost $recv\_count$ blocks in $p)$
20: **end**

REFERENCES

[1] J. L. Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975), pp. 509–517.

[2] G. E. Blelloch, *Scans as primitive parallel operations*, IEEE Transactions on Computers, 38 (1989), pp. 1526–1538.

[3] C. Burstedde, L. C. Wilcox, and O. Ghattas, `p4est`*: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133.

[4] S. K. Das, D. J. Harvey, and R. Biswas, *Parallel processing of adaptive meshes with load balancing*, IEEE Transactions on Parallel and Distributed Systems, 12 (2001), pp. 1269–1280.

[5] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, *A generic interface for parallel and adaptive discretization schemes: Abstraction principles and the DUNE-FEM module*, Computing, 90 (2010), pp. 165–196.

[6] L. Ferm, A. Hellander, and P. Lötstedt, *An adaptive algorithm for simulation of stochastic reaction-diffusion processes*, Journal of Computational Physics, 229 (2010), pp. 343–360.

[7] S. F. Frisken and R. N. Perry, *Simple and efficient traversal methods for quadtrees and octrees*, Journal of Graphics Tools, 7 (2002), pp. 1–11.

[8] I. Gargantini, *An effective way to represent quadtrees*, Communications of the ACM, 25 (1982), pp. 905–910.

[9] M. Grandin, *Data structures and algorithms for high-dimensional structured adaptive mesh refinement*, Tech. Report ISSN 1404/2014-019, Department of Information Technology, Uppsala University, 2014.

[10] B. Hendrickson and K. Devine, *Dynamic load balancing in computational mechanics*, Computer Methods in Applied Mechanics and Engineering, 184 (2000), pp. 485–500.

[11] M. Holmström, *Hybrid Modeling of Plasmas*, in Numerical Mathematics and Advanced Applications 2009, G Kreiss, P Lötstedt, A Må lqvist, and M Neytcheva, eds., Springer, Berlin, 2010, pp. 451–458.

[12] G. Linde, J. Persson, and L. von Sydow, *A highly accurate adaptive finite difference solver for the black–scholes equation*, International Journal of Computer Mathematics, 86 (2009), pp. 2104–2121.

[13] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, International Business Machines Company, 1966.

[14] H. Samet and M. Tamminen, *Efficient component labeling of images of arbitrary dimension represented by linear bintrees*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 10 (1988), pp. 579–586.

[15] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708.

[16] J. D. Teresco, K. D. Devine, and J. E. Flaherty, *Partitioning and dynamic load balancing for the numerical solution of partial differential equations*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. M. Bruaset and A. Tveito, eds., vol. 51 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2006, pp. 55–88.

[17] T. Weinzierl and M. Mehl, *Peano: A traversal and storage scheme for octree-like adaptive cartesian multiscale grids*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2732–2760.

[18] A. Zewail, *Laser Femtochemistry*, Science, 242 (1988), pp. 1645–1653.