SFC-based Communication Metadata Encoding for Adaptive Mesh Refinement

Martin Schreiber [a,1], Tobias Weinzierl [a,2] Hans-Joachim Bungartz [a,3]
[a] *Institut für Informatik 5, Boltzmannstr. 3, 85748 Garching, Germany*

Abstract. The present paper studies two adaptive mesh refinement (AMR) codes whose grids rely on recursive subdivison in combination with space-filling curves (SFCs). A non-overlapping domain decomposition based upon these SFCs yields several well-known advantageous properties with respect to communication demands, balancing, and partition connectivity. However, the administration of the meta data, i.e. to track which partitions exchange data in which cardinality, is non-trivial due to the SFC's fractal meandering and the dynamic adaptivity. We introduce an analysed tree grammar for the meta data that restricts it without loss of information hierarchically along the subdivision tree and applies run length encoding. Hence, its meta data memory footprint is very small, and it can be computed and maintained on-the-fly even for permanently changing grids. It facilitates a fork-join pattern for shared data parallelism. And it facilitates replicated data parallelism tackling latency and bandwidth constraints respectively due to communication in the background and reduces memory requirements by avoiding adjacency information stored per element. We demonstrate this at hands of shared and distributed parallelized domain decompositions.

Keywords. dynamic adaptive mesh refinement, dynamic load balancing, space-filling curves, connectivity meta data, run length encoding

Introduction

Dynamic adaptive meshes are an important building block of many simulation codes. For applications with rapidly changing meshes, these codes hinge on dynamic load balancing as static work assignment soon becomes a showstopper on massively parallel systems. However, if the load distribution changes permanently, also the connectivity changes—adjacent subdomains disconnect, formerly disjoint partitions connect, the amount of data exchanged between two nodes changes. To bookkeep the connectivity properties, i.e. to keep track who communicates with which partner with which data cardinality, demands for memory and can grow challenging—for static but in particular for dynamic adaptive meshes.

Statements on connectivity information always depend on the underlying enumeration of grid entities. Mesh serialisation due to space-filling curves (SFCs) [1] here is a popular choice. Examples are [2,3,4,5,6,7,8,9]. If all mesh elements are ordered along

[1]schreibm@in.tum.de.
[2]tobias.weinzierl@mytum.de.
[3]bungartz@in.tum.de.

an SFC, a decomposition of the SFC induces non-overlapping subdomains. A balanced decomposition then is straightforward due to the one-dimensional character of the curve [1,4] though in practice re-decomposition and in particular the realisation in source code remain non-trivial. Furthermore, the resulting partitions exhibit an advantageous surface-volume ratio, i.e. the partitions come along with a big workload relative to their communication demands [10,4]. Finally, if any two mesh elements (triangles or hypercubes) adjacent along the SFC share a common hyperface, any partition induced by a continuous SFC segment is connected, too. For $d = 2$, the connectivity is preserved even for projections of partitions onto their $d-1$-dimensional surface submanifold—a fact that holds for $d > 2$ to the best of our knowledge exclusively for the Peano SFC [8,9]. Due to this projection property, the SFC also induces an order on the surface data that is to be exchanged.

If mesh elements can be aligned along an SFC, one can interpret the underlying data structure either as topologically flat mesh or as spacetree, a generalisation of the classical octree/quadtree concept. We follow the latter approach and augment each mesh element with meta data: which adjacent vertex or face, respectively, is processed by or to be exchanged with other partitions. It is then possible to reduce and subsume this data bottom-up along the spacetree and to store it in rather coarse tree nodes instead of the fine grid without data loss. Formally, we define an analysed tree grammar [11]. It can easily be evaluated and updated on-the-fly if the grid or the domain decomposition change. This way, we eliminate the need to maintain complicated tables about the mesh decomposition.

Our experiments study the Sierpiński curve for triangular $d = 2$-dimensional meshes and the Peano curve for $d \geq 2$ with hypercubes being mesh elements. They use two open source codes [12,13]. We show that the memory footprint to hold the connectivity data is severely reduced for low payload per element. Using the SFC grammar, this naturally leads to a shared data parallelisation where race conditions are avoided, and we discuss how the grammar can help to realise distributed memory data exchange hiding latency and memory bandwidth constraints.

The remainder is organised as follows: We first formalise our notion of a spacetree and discuss the interplay of spacetrees and adaptive mesh refinement (AMR). The serialisation of the spacetree along a SFC here is of particular interest. In Section 2, we then use this serialisation for a shared and distributed data decomposition layout. Both layouts demand for the storage of meta communication data (Section 3) whose properties are studied in Section 4. Section 5 provides a brief outlook and closes the discussion.

1. Spacetree grids serialised by space-filling curves

We consider computational grids where the whole computational domain Ω is embedded into one geometric primitive. Let this geometric primitive either be a triangle or square (for two-dimensional problems, i.e. $d = 2$) or a (hyper-)cube ($d \geq 3$). With equidistant cuts of the hypercube primitives along each coordinate axis or newest vertex bisection for triangular meshes, we obtain a set of geometric elements that are affine mappings of the original primitive (Fig. 1). As we rely on cuts, the constructed small primitives are non-overlapping. We call each of them *child* element of the original primitive. $\sqsubseteq_{childOf}$ denotes this relation. Obviously, such a construction scheme can be applied recursively

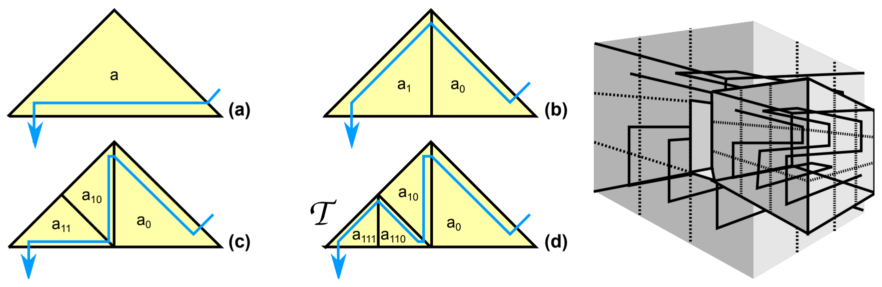

Figure 1. Grid construction with our recursively refined space tree elements $\mathscr{T}$ ordered by the Sierpiński SFC (left, $d = 2$) or the Peano SFC (right, $d = 3$).

for each of the children. It yields a set of elements $\mathscr{T}$ whose unrefined entries form an adaptive grid Ω_h. The set is structured by the underlying child-parent relation. It is a tree, a *spacetree*.

A tree induces a partial order on all geometric elements of the adaptive grids of all resolution levels. The order comprises both refined and unrefined elements (leaves). While actual solvers for partial differential equations (PDEs), e.g., might work only on the leaves of the tree, we *formally* preserve the tree here. As $\sqsubseteq_{childOf}$ does not prescribe an ordering of the children of any refined element, one way to serialise the tree into a total order is to write down a motif order for the children of one refined element and to apply this motif for each cell in the tree recursively—suitably rotated, scaled, mirrored and translated. A proper ordering resembles the construction of space-filling curves. For $d = 2$ and triangles, an ordering along the bipartitioning Sierpiński SFC is possible. For $d \geq 2$, hypercubes, and bipartitioning along each coordinate axis, Hilbert and Morton ordering can be used. For $d \geq 2$, cubes, and threepartitioning along each coordinate axis, the Peano SFC is a natural choice. All these SFCs in combination with $\sqsubseteq_{childOf}$ serialise the tree. If any two succeeding elements along the serialisation share a $d-1$ hyperface, several properties arise:

- Any continuous SFC segment S induces one connected submesh $\Omega_h|_S \subseteq \Omega_h$. This does in particular not hold for the z-ordering (Morton) where induced subpartitions might be disconnected [14].
- The SFC defines a unique order when a vertex or edge is read for the first time throughout a traversal. This order is induced by the minimum of the adjacent geometric elements along the SFC running through the elements (touch-first order) [8,9].
- A similar reasoning holds for the touch-last order, i.e. when the vertex or edge is required for the last time.
- If a grid traversal runs through a continuous SFC segment only, the two ordering properties either can be defined globally or with respect to the local curve segment S.
- Touch-first totally orders the vertices and edges. We can store data assigned to them on one input stream strictly sequentially read.
- A similar reasoning holds for the output stream and writes.
- If we invert the SFC after each traversal, the touch-last order on vertices and edges equals the inverted touch-first order of the subsequent traversal [2,8,9]. The data access follows stack (LiFo) principles.

2. Parallelisation

Let $p : \mathscr{T} \mapsto \mathbb{N} \cup \{\perp\}$ assign each geometric element of the spacetree an integer partition marker or the undefined symbol $\perp$. Each unrefined element has a marker unequal to $\perp$ (see (1)) which assigns it to a partition. Partitions in turn are dynamically mapped onto a compute unit (MPI rank, thread, core, e.g.) responsible to perform different kinds of computations on all elements of its partitions. If two leaves hold the same marker, each unrefined spacetree element in-between along the SFC traversal shall hold this marker, too. Let

$$b \in \mathscr{T} : \nexists a \sqsubseteq_{childOf} b \Rightarrow p(b) \neq \perp, \tag{1}$$

$$(\exists p \neq \perp) \wedge (\forall a \sqsubseteq_{childOf} b : \quad p(a) = p) \quad \Rightarrow p(b) = p, \qquad \text{and} \tag{2}$$

$$(a, b \sqsubseteq_{childOf} c : \; p(a) \neq p(b)) \; \Rightarrow p(c) = \perp. \tag{3}$$

If we restrict the SFC to all spacetree elements holding the same marker, this yields a connected mesh segment. The markers induce a non-overlapping domain decomposition on the spacetree's fine grid Ω_h .

We furthermore note that whenever a refined spacetree element holds a marker, all of its children and descendants hold the same marker due to (2) and (3). Markers comprise hierarchy.

If a refined spacetree element holds $\perp$, different strategies to assign it to a compute unit do exist: some codes map it to a compute unit responsible for one of the child elements, others duplicate the element for each compute unit responsible for at least one child. Both variants work for the present algorithm.

If a grid is rebalanced, p is adopted accordingly. If a grid changes, the marker of any refined former leaf and (2) yield the markers for new subgrid elements unless the grid is rebalanced. If mesh elements are removed, formerly unlabelled elements might have to be assigned to partitions to fulfill (1). While each geometric element belongs to at most one marker/compute unit, vertices and edges adjacent to several subdomains are not uniquely assigned to a compute unit. Two different strategies exist to handle this ambiguity.

With a *shared data layout*, all compute units adjacent to one vertex, edge, hyperface share one instance of this grid entity. While it is straightforward to formalise grid modifications as broadcast/reduction and to realise them on multiple copies of the grid entity, in the end the entity exists only once on one global input and output stream.

Initial read and final write access have to fit to the global touch-first and touch-last order without data races. A shared approach maps directly to cache-coherent shared memory parallelisation, PGAS or one-sided MPI.

With a *replicated data layout*, we clone vertices, edges, hyperfaces for each adjacent partition. Updates on edge and vertex data then are embarrassing parallel. However, modifications once per traversal have to be synchronised among all partitions holding duplicates. The data cardinality (number of vertices, e.g.) sent to one destination throughout a traversal by a given source equals the number of records received at the end of the traversal from this communication partner. As each local SFC is a subset of the global SFC, each partition sends data along the local touch-last order. In the subsequent iteration, each compute unit receives data along the touch-first order (cmp. stack principle) and merges duplicated data.

These activities have to address the proper communication partners, i.e. find out where to send data to and receive in return. Exploiting the causality of many read and write buffers and the fact that each spacetree refinement makes communication buffers grow at most by a given element number in parallel, we avoid overheads induced by reallocation of the stacks wherever possible. In particular, we infer upper limits for the maximum required buffer capacity a priori and hence avoid reallocation during grid traversal completely. Prospective stack allocation for vertices extends our previous work [6,15]. A replicated layout maps either to shared memory or distributed memory parallelisation.

3. Communication meta data

To realise a domain decomposition, we have to track where to send data to or which part for communication stream to read and write.

We introduce two different yet cognate approaches for the two data layouts that both allow for run length encoding of the meta data yielding a low memory footprint and are easy to derive given a decomposition. For this, let each element in the spacetree *formally* hold a tuple $(read, write, \mathscr{C})$. This is the communication meta data.

Shared data layout. In each unrefined element, touch-first defines globally how many adjacent vertices are read by this element for the very first time throughout the present traversal. *read* holds this counter. *write* is the corresponding value for writes to the output stream. For any refined spacetree element a with $p(a) \neq \perp$, let $read(a) = \sum_{b \sqsubseteq_{childOf} a} read(b)$ and and $write(a) = \sum_{b \sqsubseteq_{childOf} a} write(b)$. Both quantities can be analysed bottom-up and are a run length code specifying for whole subtrees how many data are read by this subtree from a global input stream or written, respectively.

The application to other grid entities such as edges is straightforward. We note that touch-first for $d-1$-dimensional (hyper)faces implies touch-first for all lower-dimensional grid entities, i.e. such information can be reconstructed.This note links *read* and *write* to the automaton grammars of [1,2,8,9] et al.

p induces a recursive fork pattern for a shared data traversal where all input grid entities are serialised along one input stream and all grid entities have to be written to one output stream. Let the grid traversal run through the spacetree top-down serially. Whenever it encounters a node with $r \neq \perp$, it can fork the traversal of the respective subtree and continues to traverse the remaining tree.

read defines how many grid entities are read by a particular subtree, i.e. partition, along the global touch-first order. For each traversal fork, we cull *read* elements en bloc from the input stack and assign this data chunk to the a compute unit running computations on this subdomain. Here, it has to read data along its local touch-first order.

If this read coincides with the global touch-first order, the compute unit takes the data from the input data chunk, performs computations on it, and provides them as read-only data to all other adjacent forked compute units. Otherwise, the respective grid entity is processed by another compute unit. The *write* process is similar: Prior to each traversal fork, we allocate *write* entries on the output stream en bloc. Each forked compute unit then fills this continuous chunk again following the global touch-last order and performs the corresponding reductions.

Due to overhead considerations, forking load and store traversals only pays off if the number of reads and writes is significantly high, i.e. if the load per forked compute unit is sufficiently high. It consequently does make sense to augment the marker analysis with a subtree size threshold. Our meta data analysis both defines a way how to parallelise tree load and store processes for a shared input/output serialisation, and it defines a methodological way, which data is to be read or stored, respectively, by which thread. As a result, we hold the meta data only for spacetree elements where a fork is reasonable. For all other nodes, meta data is discarded. This reduces the meta data footprint. If the grid is repartitioned or changed, an update of the meta data is straightforward throughout the subsequent tree traversal.

Replicated data layout. In each unrefined element, touch-first defines locally which adjacent vertices are read by this element for the very first time throughout the present traversal of a given partition.

touch-last yields the write counterpart. Consequently, the local touch-last order identifies whether a vertex will be reused by the local partition in this traversal. If not, it has to be synchronised with all other adjacent compute units before it is filed for the subsequent traversal, i.e. its data updates have to be shared. touch-first is the counterpart: prior to any work with the vertex, all synchronisation updates have to be received and merged into the grid entity.

Let $\mathscr{C}$ hold a sequence of $(\#n, p)$ tuples specifying along the local order of the vertices of a given element how many grid entities $\#n$ have to be sent to a particular partition p. The receive data sequence derives from the very same set due to the stack principle. For any refined spacetree element a with $p(a) \neq \perp$, we concatenate the $\mathscr{C}$ sequences of the leaves.Afterwards, subsequences $((\#n_1, p_i), (\#n_2, p_i))$ are replaced by $((\#n_1 + \#n_2, p_i))$. $\mathscr{C}$ is analysed bottom-up and run length encoded.

$\mathscr{C}$ is to be held per grid entity type (vertex, edge, face, etc.). We however propose an optimisation extending [6]: Given the adjacency information for the $\hat{d} < d$ manifolds, we can derive from this meta data which $\hat{d} - 1$ entities of each cell are adjacent to other compute units if we insert additional tuples $(0, p)$. These tuples denote that an $\hat{d} - 1$ entity connects to a partition p that is not connected to the current cell through an $\hat{d}$ entity.

A sequence $(3, p_1), (0, p_2), (0, p_3), (4, p_4)$ for edges in a two-dimensional setting, e.g., implies that a subdomain surface consists of three edges connecting to partition p_1 followed by four edges connecting to p_4. The vertex in-between links to both p_1 and p_4 as well as additional partitions p_2 and p_3. As the SFC induces compact and connected partitions, these special cases are rare and allow to plant additional information into the adjacency sequences with low memory footprint.

p induces a decomposition of the global spacetree into sequence of subtrees. We denote, extending the notion from [15], a sequence of these subtrees with one p as cluster. It is straightforward to initialise the required communication buffers prior to the traversal and to exploit the possibility to exchange data en block rather than to send individual messages.

After each traversal, $\mathscr{C}$ defines how many records are sent or fetched from other partitions. We buffer the received data and pop it from this buffer throughout the subsequent traversal. In a shared memory environment, send and receive buffers can be directly accessed by the compute units. In a distributed memory environment, they have to be exchanged via MPI, e.g., explicitly.

As the coarsened meta data hold all adjacency information, we remove it from all finer elements in the spacetree. This reduces the meta data footprint. If the grid is repartitioned, an update of the meta data is straightforward [6] either throughout a subsequent tree traversal or due to incremental modifications of the sequences interplaying with mesh element reassignments.

4. Results

All experiments were conducted on SandyBridge nodes being either part of the SuperMUC or TUM's MAC cluster. A node has two processors with eight cores each that share a total of 32 GBytes main memory.

Studies on the shared data layout were conducted with the software Peano [13]. Studies on the replicated data layout rely on Sierpiński [12]. We used Intel's Threading Building Blocks (TBB) for shared memory systems and MPI on distributed memory environments. As application scenarios, we used Euler and Shallow water equations for $d = 2$ and a simple heat diffusion setting in a inhomogeneous medium for $d = 3$ with state-of-the-art (discontinuous) Galerkin FEM discretization in space and Euler explicit 1st order integration in time.

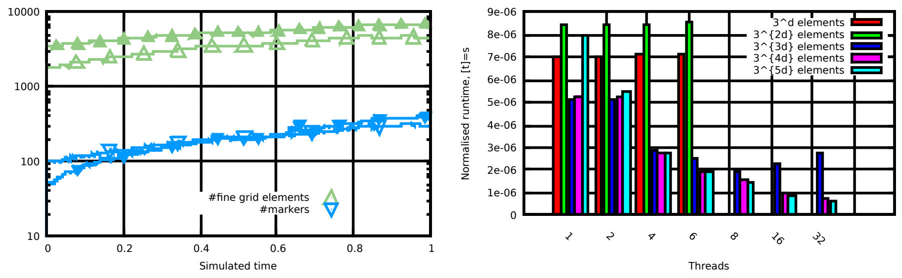

Figure 2. Left: snapshot of two characteristic $3d$ experiments tracking both the total number of elements on the finest grids and the number of tree elements holding *read* and *write* marker; right: typical cost per unknown of the load process.

The memory footprint of both data layouts is, by construction, very small but depends on the actual partition shapes, i.e. it depends on how coarse within the tree data can be held. For the shared data layout, we identified reasonable minimal values of *read* and *write* constraining the minimal forked tasks' size, i.e. the size of a partition p, by exhaustive search. $\mathcal{O}(3^{3d})$ turned out to be the smallest reasonable value independent of grid structure and threads available (Fig. 2). Furthermore, it turned out that, interplaying with [16], read and write operation forks pay off solely for regular stationary subregions of the grid—for other regions, the handling of hanging vertices and dynamic refinement outperform the loads and stores anyway.

If we hold the meta data *read* and *write* only for regular subregions, around every 20th fine grid element corresponds to a element in the spacetree annotated with the two counters (Fig. 2). Combining these two selection criteria, the number of markers relative to the fine grid elements is neglectable.

Our replicated data layout holds communication meta data per cluster. Obviously, the memory footprint depends on the shape of the clusters, their number, and their RLE compression rate. A cluster's surface-to-volume ratio is quasi-optimal due to the use of SFCs [10], i.e. it is only by a constant worse than a spherical domain layout. This optimality applies to the cluster boundary faces to be labelled and kept tracked by the meta data storage scheme. Good cluster sizes depend on the grid layout as well as overhead considerations vs. concurrency level if they are atomic entities to dynamic load balancing [6,15]. If we compare the cardinality of cluster boundary elememts (edges) to the number of meta data tuples of our approach (Fig. 3), the bigger the cluster the bigger the pay-off. For typical cluster sizes from 4096 through 8192, our scheme induces a memory footprint reduction factor from 9 to 12.

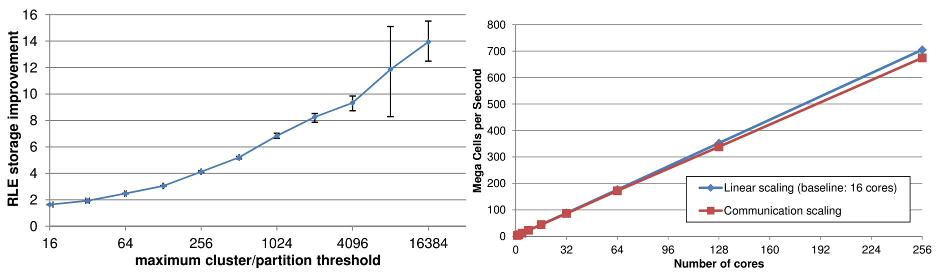

Figure 3. Left: Ratio of cluster boundary edges and vertices to $\mathcal{C}$ entries for typical maximal cluster sizes with maximum and minimum ratios; right: scaling of a simulation on a SFC based regular grid with non-blocking MPI communication for different numbers of MPI compute units (ranks).

As the communication meta data encoding yields communication data exchange cardinality and as the SFC's touch first/last order ensures that communication data is read and written continuously, communication buffers can be transferred en block. No accummulation or pre-/postprocessing overhead is induced. This insight is reflected by MPI efficiencies of $95,6\%$ (Fig. 3). A similar pattern is found for the shared data layout, where the elimination of meta data entries below a given threshold both saves memory and guides the parallelisation not to fork additional tasks for those sub-spacetrees. Given such a threshold as discussed before, we observe convincing scaling as no data synchronisation is required—race conditions are avoided a priori due to replicated data layout.

We validate our results by a theoretical analysis of the ratio between memory improvements of our RLE and per-element payload $\mathcal{W}$ for different cluster sizes. We consider a regular triangulated subdomain refined to $depth$ via bisection yielding 2^{depth} cells. For the bipartitioning Sierpiński curve, the number of communication partners for a single triangle hence is bounded by $\mathcal{S} := 3+2\cdot 5+1+\mathcal{W}$ taking into account that two out of seven adjacent elements have an edge of the element itself encoded by the meta data entries and the vertex at the triangle legs requiring only one RLE entry. This yields a required storage capacity of $\mathcal{R} := 2\cdot(3+2\cdot 5+1)$ words taken for the RLE encoding on regular grids for all possible communication information elements with the factor of two considering the memory to store the run length encoding. The ratio of per-element and RLE stored memory requirement is then given by $\frac{\mathcal{S}\cdot 2^{depth}}{\mathcal{R}+\mathcal{W}\cdot 2^{depth}} \overset{2^{depth}\to\infty, L.H.}{=} \frac{\mathcal{S}}{\mathcal{W}}+1$. Graphs for different payloads per element and cluster sizes are given in Fig. 4.

For a payload of 4 words per cell, used for typical finite volume simulations, the model shows that our RLE encoding is already beneficial for cluster with only a few cells, and typical cluster sizes are in the magnitude of hundreds of cells. For a payload exceeding 64 words per element, the savings are almost close to one, thus the benefits are not given by less memory consumption. Though, RLE based communication is still advantageous procuring communications en block.

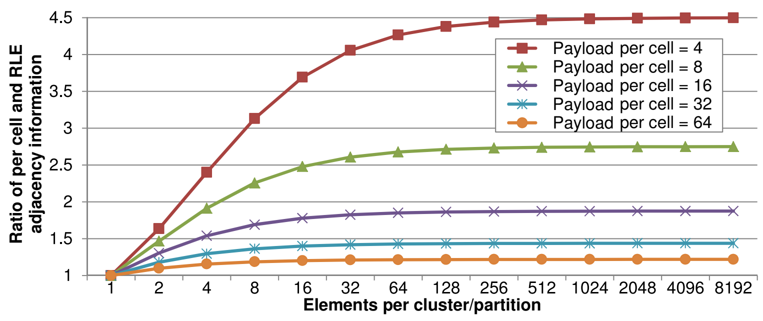

Figure 4. Factor of saved memory with RLE cluster encoding compared to storing adjacency information per-cell. The asymptotical behavior to $\frac{14}{\mathscr{W}}+1$ is clearly visible, thus providing a constant limit of the memory required for per-element adjacency information.

We only considered a single word used for communication. Extending the adjacency information with further information such as MPI ranks, identifiers, etc. would increase $\mathscr{S}$ and thus makes our RLE yet more valuable.

5. Conclusion and outlook

If parallel mesh-based PDE solvers or, in general, applications rely on the interplay of space-filling curves and spacetrees, it is a natural choice to exploit these two ingredients not only for sophisticated mesh management and computations but also for communication meta data.

The present paper introduces one such approach: an analysed tree grammar is used to derive, maintain, and store communication meta data efficiently and economically.

Our algorithmic efforts yield a small meta data memory footprint and fall elegant into place. However, they add yet another nuance of implementation complexity to the spacetree codes. In practice, one carefully has to balance whether this additional tree grammar and the savings in memory are worth the effort. In many cases, plain yet expensive lookup tables to store connectivity and topology might work as well on todays computers.

On the long run, supercomputing roadmaps however suggest that the impact of communication and the efficient realisation of its administration—in particular with respect to memory—can not be overestimated. Our algorithmic ideas then materalise one out of many building blocks.

Future steps hence comprise the combination of the present ideas with other parallelisation aspects to tackle simulations on the large-scale within the two open source code environments [12,13]. Here, we expect the present ideas to be an enabling technique. We also expect that real-world runs yield further insight about the present algorithm's properties—how the low memory footprint facilitates and interplays with dynamic load balancing, the invasion of computational resources as they might be found in urgent computing, and heterogeneous architectures.

Acknowledgements

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing (SFB/TR 89). It is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). All software is freely available at [12] or [13].

References

[1] M. Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Comp. Science and Engineering*. Springer-Verlag, 2013.

[2] M. Bader, C. Böck, J. Schwaiger, and C. A. Vigh. Dynamically Adaptive Simulations with Minimal Memory Requirement - Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal of Scientific Computing*, 32(1), 2010.

[3] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scient. Comp.*, 33(3):1103–1133, 2011.

[4] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 25(7):827–843, 1999.

[5] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[6] M. Schreiber, H.-J. Bungartz, and M. Bader. Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. Puna, India, 2012. IEEE International Conference on High Performance Computing (HiPC), IEEE Xplore.

[7] V. Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

[8] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Diss., Institut für Informatik, Technische Universität München, München, 2009.

[9] T. Weinzierl and M. Mehl. Peano—A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011.

[10] H.-J. Bungartz, M. Mehl, and T. Weinzierl. *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, chapter A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves. Springer-Verlag, 2006.

[11] D. E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12. Springer-Verlag, 1990.

[12] M. Schreiber. Sierpiński—a Framework for Full Adaptive 2D Simulations, 2013. www5.in.tum.de/sierpinski.

[13] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2013. www.peano-framework.org.

[14] M. J. Aftosmis, M. J. Berger, and S. M. Murman. Applications of space-filling curves to cartesian methods for cfd. *AIAA Paper*, 1232:2004, 2004.

[15] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In *EuroPar 2013*, 2013. (accepted).

[16] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In *Parallel Processing and Applied Mathematics, PPAM 2009*, number 1 in LNCS, 2010.