Cluster Optimization and Parallelization of Simulations with Dynamically Adaptive Grids

Martin Schreiber, Tobias Weinzierl, and Hans-Joachim Bungartz

Technische Universität München,
Boltzmannstrasse 3, 85748 Garching
{martin.schreiber,weinzier,bungartz}@in.tum.de

Abstract. The present paper studies solvers for partial differential equations that work on dynamically adaptive grids stemming from spacetrees. Due to the underlying tree formalism, such grids efficiently can be decomposed into connected grid regions (clusters) on-the-fly. A graph on those clusters classified according to their grid invariancy, workload, multi-core affinity, and further meta data represents the inter-cluster communication. While stationary clusters already can be handled more efficiently than their dynamic counterparts, we propose to treat them as atomic grid entities and introduce a skip mechanism that allows the grid traversal to omit those regions completely. The communication graph ensures that the cluster data nevertheless are kept consistent, and several shared memory parallelization strategies are feasible. A hyperbolic benchmark that has to remesh selected mesh regions iteratively to preserve conforming tessellations acts as benchmark for the present work. We discuss runtime improvements resulting from the skip mechanism and the implications on shared memory performance and load balancing.

Keywords: dynamic adaptivity, cluster skipping, shared memory load balancing, space-filling curve.

1 Introduction

Mesh-based solvers for partial differential equations (PDEs) that rely on the combination of recursive spatial sub-refinement with space-filling curves (SFCs) are popular in multiple application fields [1,6,13,15,21]. They embed the computational domain into a geometric primitive or a strip of primitives, and subdivide the primitives locally and recursively into smaller primitives. Those primitives are ordered along the SFC. Such a spacetree formalism facilitates dynamically adaptive grids and parallel mesh processing, as the curve prescribes a unique total mesh element order that can be cut into equally sized partitions for parallelization. In particular matrix-free solvers with heterogeneous solution smoothness such as explicit schemes for hyperbolic conservation laws resolving shock fronts benefit from the dynamic adaptivity [11]. They then usually sweep the grid once per time step and update (partially) the solution in each grid cell [7,20].
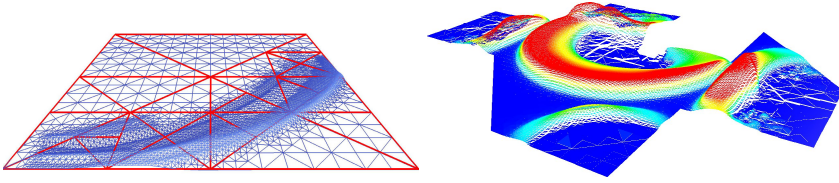
Fig. 1. Shock wave runs through domain while the grid changes dynamically and decomposes into clusters. Some of them were removed from the right illustration.

An efficient single compute node grid traversal here is essential. Multiple spacetree codes report on memory-efficient encodings, while the space-filling curve implies high memory access locality due to hash tables [9] or stack-based grid data schemes [1,21,22], e.g. Both ingredients tackle the memory bandwidth challenge which is expected to be one of the most crucial challenges in upcoming architectures [4]. We refer to [21,22] and remarks therein for measurements. Skipping coarser levels of the tree and to traverse only its leaves are further techniques reducing the total workload [2,6] if the geometric multi-scale structure is not required. Tree cuts developed for geometric multigrid solvers in turn allow to skip grid regions under-running a given mesh size threshold [21]. To the best of our knowledge, there is however neither a formalism nor an analysis of a technique that allows to skip whole grid regions independent of their resolution.

The present paper discusses an approach where multi-scale grid regions are skipped throughout the subsequent traversal. This speeds up algorithmic phases where either only spatial subregions are of interest or a holistic mesh processing does not justify the effort. Examples are meshing traversals reconstructing a proper 2:1 balancing [17] or local time stepping where regions lagging behind in time have to be updated prior to other mesh elements [7,20]. An example for the latter are solvers for linear equation systems that update preferentially sets of unknowns with significant residuals [16]. While the present work focuses on triangle-based meshing in combination with the Sierpiński space-filling curve [3], all paradigms can directly be applied to other SFC-based codes.

Obviously, an on-the-fly choice of spatial subsets handled by the traversal interplays with the traversal's concurrency and the parallelization—in particular if massive numbers of tightly coupled cores have to be handled that are sensitive to NUMA effects, ill-balancing, latency, and tasking overhead [4,18]. We introduce a shared memory parallelization that does not deteriorate due to the skipping and compare it to straightforward task-based parallelization. Reduced memory access and improved data affinity here compensate the reduced concurrency level.

The remainder is organized as follows: We first briefly describe the mesh paradigm and define the term cluster (Sect. 2). In Sect. 3, we then pick up this formalism to introduce the cluster skip mechanism. Implications of this mechanism on the shared memory load balancing and communication behavior are subject of the subsequent section, where we also introduce our affinity-aware

implementation. Some results for a benchmark exhibit promising performance properties, before a brief conclusion and outlook in Sect. 7 close the discussion.

2   Grid Construction and Clustering

Our grid follows the spacetree/-forest formalism [1,6,21,22]: The computational domain is embedded into a triangle or a strip of triangles. For each triangle, we autonomously decide whether this triangle shall be split once. Such a scheme yields a binary tree or forest where triangles obtained due to a split are children of their preimage. Both the splitting rule and the order of the two children follow the construction scheme of the Sierpiński space-filling curve (SFC), i.e. the curve prescribes which triangle faces may be split, and the curve induces an order on the children [1,2,18]. The SFC in combination with depth-first defines a total order on all triangles of all levels. Let the *level* of a triangle be the minimum number of refinement steps required to construct the triangle. The initial triangle or the initial triangle strip, respectively, have level zero. Unrefined triangles are *leaves*. Unknowns are assigned to leaves only. We reiterate from [21,22] that a depth-first traversal of the spacetree induces an element-wise traversal of the leaves. Such a depth-first traversal can be formalized and realized as stack automaton [14,21] triggering in turn a matrix-free solver.

Starting from the notion of a binary triangle tree yielding the adaptive grid or a binary forest, respectively, we introduce the following notions: A *cluster* is a subtree of the binary mesh tree. It is identified by a unique tree node (empty circle in Fig. 2). If a triangle belongs to a cluster, all its successors, i.e. all finer triangles covered by it, belong to the same cluster, too. As the SFC defines a depth-first total order on all triangles, it induces an order on the clusters.

Let $\mathcal{C}$ be the set of clusters, and let each cluster have a list of neighbor clusters, i.e. clusters whose triangles shared at least one face or a part of it with a triangle from the respective cluster. The following algorithm clusters the spacetree:

- Assign each leaf a weight $W = 1$ and a marker $R = 0$.
- Let each cluster in $\mathcal{C}$ hold exactly one unrefined triangle and vice versa. Each cluster has at most three neighbors. The cluster cardinality equals the number of leaves.
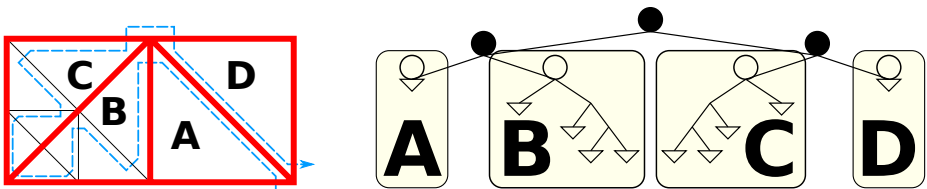- Run through the spacetree bottom-up:

Fig. 2. Domain triangulation with clusters marked with thick borders (left). Representation of same grid by a binary tree constructed with the Sierpiński SFC (right).

- Set a refined node's weight to the sum of the left and the right child, i.e. $W = W_{lhs} + W_{rhs}$.
- If $W \leq W_{join}$ being a given threshold, merge the two clusters of the right and the left child. Replace these two cluster triangles in $\mathcal{C}$ by their parent, i.e. reduce $\mathcal{C}$'s cardinality by one. Also merge the neighbor lists of the two children. This is an operation with linear complexity.

- Run through the spacetree top-down:
  - If a node is the left child of a parent triangle, set $R = R_{parent}$.
  - If a node is a right child of a parent triangle, set $R = R_{parent} + W_{lhs}$.

The algorithm assigns each leaf to one cluster (Fig. 2). The clusters' size is controlled via W_{join}. Each cluster has a distinguished coarsest triangle holding its W and R value, and each cluster knows all of its neighbor clusters. Once such a clustering is found, we easily can adopt it whenever the grid changes. For this, it also does make sense to introduce a split weight W_{split} as counterpart of W_{join}. Whenever a triangle is refined, its two children inherit the cluster affiliation. If a cluster exceeds the threshold W_{split}, it decomposes into two clusters—each one represented by the two triangles on the 1st recursion level. There is no need to construct clusters from scratch several times.

The clusters define a graph on the mesh where each graph node is a cluster. Two nodes are connected if their clusters shared a common face. This graph is small compared to the connectivity graph of the original mesh.

Our algorithm refers to a binary tree. An extension to a binary forest is straightforward. Furthermore, a bottom-up construction of the clusters starting from the whole tree or forest in practice is not an optimal choice. Instead, it does make sense to create the binary tree up to given level. The triangles of this initial level then prescribe an *initial clustering*. Starting from the initial clustering, the grid is refined further and the clustering is adopted.

Our cluster analysis and mesh traversal fits to a recursive realization. Though straightforward, in practice it might make sense to reduce recursion overhead. One approach is to replace it by an iterative scheme to avoid call-stack overhead [2,8]. Formally, such a transformation equals recursion unrolling. If recursion unrolling is applied within clusters only and if clusters hold exclusively totally balanced subtrees, i.e. all leaves within one particular cluster have the same level, clustering and non-recursive realization mirror the optimization from [8]. Such an approach however relies on invariant grid regions and has to be used carefully if the grid changes frequently. Related work furthermore stores the leaves of the clusters only and reconstructs the coarser levels of the tree bottom-up [2,15]. Our implementation holds the whole tree and sticks with a recursive realization, but case distinctions within the recursive code are eliminated aggressively: Automaton states together with their possible transitions are rewritten by a code-generator into specialized functions with a minimal set of case distinctions [19] severely reducing branching mis-predictions. Argument-controlled context profiling validates that this pays off [10]. Furthermore, PDE-specific operations are invoked on the leaves only.

3   Dynamically Adaptive Cluster Reordering and Skipping

Let f be a marker operation on triangles that identifies those to be refined or coarsened next. Our element-wise traversal runs through the grid or tree, respectively, and evaluates f on each triangle. The subsequent mesh sweep then refines or coarsens which might yield non-conforming grids, i.e. grids with hanging nodes. As we rely on conforming tessellations, the non-conform refinement or coarsening mark further triangles. We have to traverse the grid multiple times until the global grid becomes conform again (Fig. 3), i.e. grid modifications might trigger a cascade of grid traversals propagating the grid updates.

Let each triangle hold a state $S \in \{0, 7\}$ and one marker per face encoded in a 3-tuple $T_f = (000)$. If f modifies a triangle, it updates its state as well as the face meta information. The number of possible refinements and meta information updates is fixed (Fig. 4). From the face meta data adjacent cells can derive how they have to adopt to make the grid conform. Along the SFC this information propagation resembles Gauß-Seidel. Otherwise it is a Jacobi-like information spreading which motivates the fact that multiple sweeps are typically necessary to make the grid conforming.

The state encodes the triangle's local state, the incoming marker adjacent refinement/coarsening information. The following table gives the new triangle state as well as the marker forwarded to adjacent triangles:

	state	incoming edge marker
		000	001	010	011	100	101	110	111
no request	**0**	000,**0**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
INVALID	**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**	000,**1**
local coarsening request	**2**	000,**2**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
local refine request	**3**	100,**4**	100,**5**	100,**6**	100,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp	**4**	000,**4**	000,**5**	000,**6**	000,**7**	000,**4**	000,**5**	000,**6**	000,**7**
refined: hyp, left	**5**	000,**5**	000,**5**	000,**7**	000,**7**	000,**5**	000,**5**	000,**7**	000,**7**
refined: hyp, right	**6**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**	000,**6**	000,**7**
refined: hyp, right, left	**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**	000,**7**

We consider clusters to be atomic entities coupled to other clusters via their one-dimensional boundary sub-manifold. Let an *active* cluster be a cluster where f has marked elements. Obviously, this property can be reduced throughout a traversal of the cluster tree. If all triangles hold $T_f = (000)$ after the traversal, the cluster is not active. A cluster's *active* flag is an or-combination of all triangle meta data.

The marker propagation is a face data exchange. It directly implies a marker semantics on clusters. A cluster without local refinement becomes active if and only if a neighboring cluster has set a face marker in the iteration along a common face. As we have the neighborhood relation at hand, this leads to a publish-poll pattern:
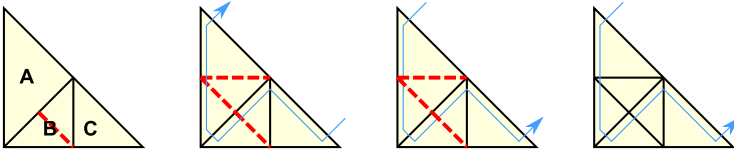
Fig. 3. From left to right: grid with three triangles A,B, and C. As B is refined, A has to be refined twice to preserve a conform tessellation.
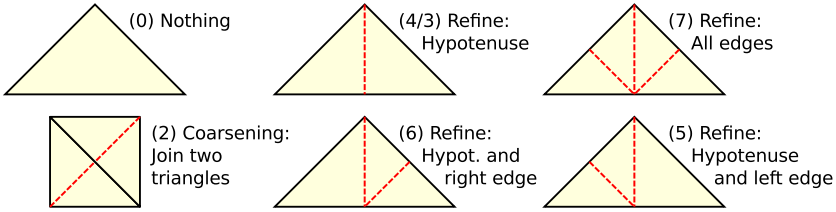

Fig. 4. Different states from the automaton to form a conforming grid. The dashed lines are new edges of the refined triangle.

- If a cluster is *active*, the traversal automaton has to run through its grid. Marker information from the boundary faces is polled. This might set the cluster's state to active again. It also might set markers along the cluster's boundary, i.e. publish new markers.
- If a cluster is *not active*, the traversal automaton runs over the boundary markers published by the neighbors. If all of them are unset, no grid updates within the cluster are necessary. If one marker is set, the cluster is active and is traversed.

With the second case distinction at hand, we are able to *skip clusters* throughout the traversal when we know a priori, i.e. when the automaton enters the cluster's coarsest triangle, that they are not active. In this case, we continue with grid elements of another not yet processed cluster. As each sweep polls the marker information and updates only subregions of the grid, the overall data flow pattern resembles a petri net on the cluster graph.

For the realization of data exchange along clusters representing fractions of the SFC, we refer to [1,2,18] and [21,22] for $d > 2$ with hypercubes instead of triangles. The SFC distinguishes right from left neighbors uniquely, i.e. the published markers' cardinality is bounded by the surface. As the triangles are lined up like pearls on a string along the Sierpiński SFC, the cluster partitions are connected and exhibit a quasi-optimal surface-volume ratio [5]. The SFC furthermore linearizes the left and right boundary uniquely whereas the boundary fractions of relevance for one particular neighbour are connected, continuous, and already published in the read order [5,18,21].

Though it can happen that clusters are set active multiple times due to grid conformity sweeps, we do not observe such a behavior often. Furthermore, if we assume f to be idempotent on the triangle, the proof is straightforward

that the marker update mechanism does not induce cycles. While the order for the refinement and coarsening here is given implicitly, reordering for different purposes (local time stepping, e.g.) might yield additional benefit.

4   Cluster-Based Parallelization

Without considering parallelization so far, we observe that leaves are coupled due to their faces to neighbouring triangles. For our present applications, leaves write data to their common faces in one sweep. At the end of the traversal, both triangles adjacent to any face have accumulated their data on the face. These data typically are input parameters to flux computations. In the next iteration, the data is read within the triangle and the triangle's values are updated. Such a data flow mirrors the marker pattern of the previous section.

With cluster based parallelization and by considering clusters to be atomic grid entities, we can split the data exchange into two phases. First, clusters write data to their interface faces: faces shared with other cluster. If grid cluster interface faces are duplicated per cluster, this write process is thread-safe. With a stack- and stream-based approach, the data for each adjacent cluster is consecutively stored in memory. We can run length encode which face data are sent to which neighbour and exchange data belonging to a particular neighbor en block and efficiently. Then, we can merge the duplicated interfaces explicitly prior to the next grid traversal. By running the merge operation with consideration of the order of clusters along the SFC, duplicate flux computations can be avoided at the cluster interface. Due to the sub-manifold and the minimal surface property [5], these merge operations are cheap compared to the overall updates and the memory overhead is small.

Depth-first traversals of trees are a classic demonstrator for task-based parallelism where a shared data structure's disjoint subsets are handled by different threads. No data overhead besides cluster surface communication buffers is induced. We pick up this property to derive two different parallelization schemes. In combination with skipping, minimal cluster sizes, software-based affinities as supported by TBB, and the comparison of TBB and OpenMP tasks, this yields a multitude of different parallel algorithmic flavors.

For our *massive tree split* approach, we make the traversal automaton traverse the grid top-down. In each node, it spawns a right and a left task. If a child identifies a cluster, this subtask is not split up further. As the number of clusters typically exceeds the number of tasks, the grid traversal floods the system with tasks, delegates the distribution of tasks to threads to the runtime system, and relies on work stealing to achieve well-balanced workload. Subdomains handled by one thread might be discontinuous. Due to nondeterministic work stealing, the assignment to threads even might change from grid traversal to grid traversal. We expect data affinity penalties from this property that has to be compensated by a high concurrency level, and point out that TBB supports a manual choice of task affinities.

For our *owner-computes* approach, we analyze the tree attributes: The property W on the spacetree's root determines the total workload on the grid. Given
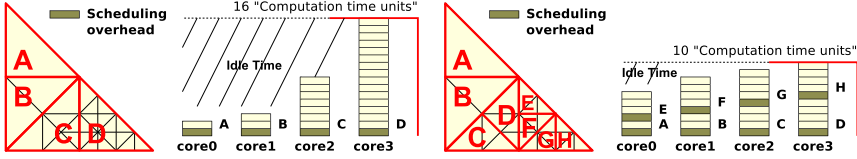
Fig. 5. Left image: A cluster size which leads to as many cluster as there are cores available on the system would lead to a workload imbalance. Right image: Creating more clusters than there are cores available leads to a better balanced problem even by considering the tasking overhead marked with dark-yellow boxes.

p threads, a thread i knows that each cluster with $R \in [iW/p, (i+1)W/p[$ is to be handled by this thread for a rather balanced work decomposition. Consequently, each thread can run through the tree processing only triangles or clusters, respectively, fitting into its work interval. This is a concurrent read due to publish-poll. Such a behavior mirrors the logical merge of multiple subsequent clusters along the Sierpiński curve for one task.

Traditional SFC parallelization [5,6,9,15,17,18,21] cuts the curve into equally sized chunks and distributes these chunks among the cores. With threshold based cluster splitting, such an equal balancing is not possible anymore, and it is obvious that an owner-computes scheme suffers from ill-balancing whereas flood filling might compensate ill-balancing due to work stealing. We point out that clusters of size one with an owner-computes scheme mirror a traditional SFC-based parallelization where the workload is cut into equally sized pieces along the curve. Our clustering either has to compensate ill-balancing due to an efficiency gain, or clustering has to tackle potential ill-balancing explicitly.

For the latter approach, we use *scan clustering* decomposing clusters on-the-fly into their tree if a cluster overlaps with an optimal SFC partitioning. This mechanism weakens the clusters' atomic property, but allows for a fine granular a priori load balancing. An additional paralllization degree of freedom arises if any cluster exceeding a given size is decomposed automatically, while the big clusters are preserved for the skip mechanism. We then again rely on work stealing to tackle ill-balancing (Fig. 5).

5   Benchmark Scenario

A setup based on the shallow water equations (SWE) computing a radial breaking damn in a basin acts as benchmark for the present paper. The rectangular basin has side length of $5000m$. It is filled with fluid of a depth of $10m$ (sea level). The initial condition is a radial breaking dam with radius $500m$ around the point $(-2000m, 2500m)^T$ relative to the origin at $(0,0)^T$. Its height relative to the sea level is $1m$. We apply non-reflecting boundary conditions (Fig. 1).

The system is discretized using discontinuous Galerkin method with 1st order cell basis functions and 3rd order Gaussian quadrature on each face. An explicit

Euler time-stepping scheme with Rusanov fluxes acts as time stepping. Throughout the simulation, we refine each triangle with a water surface displacement relative to the normal sea level exceeding the threshold $0.1m$ and allow coarsening for triangles with a threshold above $0.01m$. The refinement is bounded by the maximal triangle level 8. Different algorithmic phases realized by grid traversals do exist: Setup and visualization, time stepping, and consistency traversals recovering the mesh conformity. The latter typically modify only few triangles.

6 Results

All experiments were conducted on an Intel Westmere with 4 Intel Xeon CPUs (E7-4850@2.00GHz) and 256 GB memory totally available on the platform. This gives 4×10 physical cores plus hyper-threading. For the parallelization, we have a TBB and OpenMP realization due to Intel Composer XE (ver. 2013.1.117). We used affinity bit-masks to hard-limit the number of threads and avoid TBB's automatic worker task creation. Thread affinities are set such that they map the first ten threads to the ten physical cores on the CPU on the first socket. Thread numbers 41–80 are mapped onto hyper-threading cores.

Prior to algorithmic studies, we first determined for a good cluster size for both OpenMP and TBB (Fig. 6). It results from extensive search. While TBB outperforms OpenMP for most settings, 8192 is a natural choice for the cluster size threshold. This value is used from hereon. We also stick with TBBs.

Next, we studied cluster skipping distinguishing adaptivity traversals, computation traversals, as well as cluster construction (Fig. 7). The construction time is negligible, the simulation time itself is independent of the skipping. The skipping however reduces the time spent to make the grid conforming when it has changed before. We observe that this improvement is the better the fewer threads are used which is a natural result from the inhomogeneity of the workload due to splitting, i.e. work balancing gains impact but also introduces overhead. However, the splitting optimization is robust. A normalization of the run-times without skipping reveals that the normalized speedup degradation is marginal.
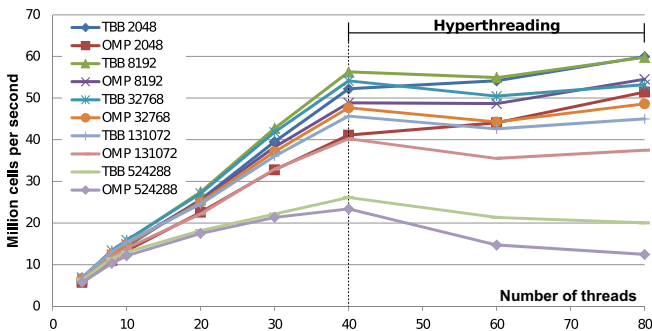

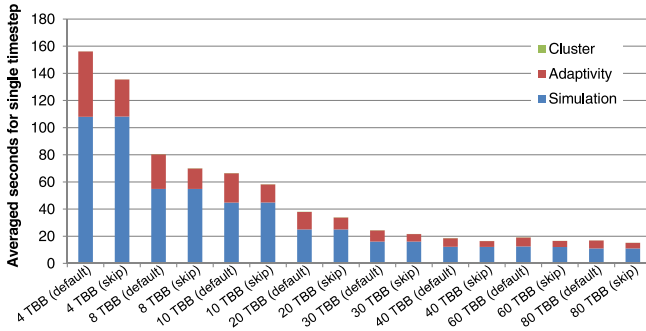**Fig. 6.** OpenMP vs. TBB tasking comparing different cluster sizes

Fig. 7. Detailed timings for each simulation phase. Time taken for clustering phase is invisible small.
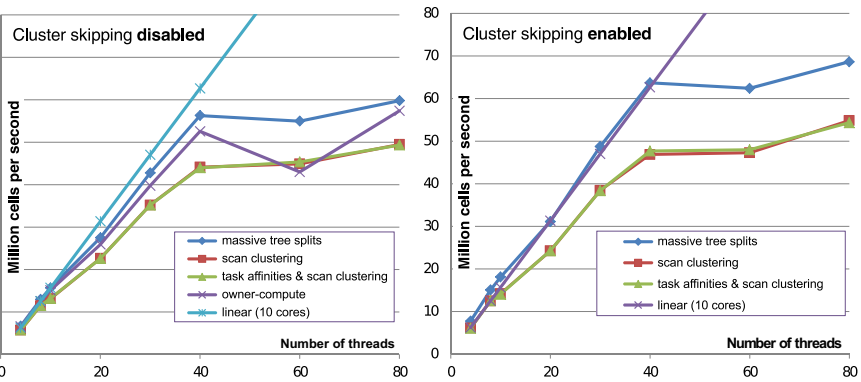

Fig. 8. Comparison of different parallelization strategies with an without skipping for short simulation time with few adaptivity traversals and few skips

For one hundred time steps, we compared different combinations of skipping and parallelization strategies (Fig. 8). The skipping again pays off and allows us to obtain linear speedup in some cases, i.e. the algorithmic optimization helps to close the gap between optimal and observed scaling—however only compared to non-skipping algorithms. Massive tree splits outperform the other parallel approaches as long as the cluster size is chosen reasonable (Fig. 6) and does not hinder the algorithm to exploit all cores. The owner-computes scheme cannot compete even though we use scan clustering obtaining theoretically almost perfectly balanced work decompositions. However, owner-computes with its manual data affinity yields better performance than TBB's task affinity feature. For this experiment, flooding the runtime system with tasks and cluster skipping are the methods of choice.

With first simulation results for a short run at hand, we studied the same setup for 15,000 time steps (Fig. 9). Longer observation intervals imply more grid changes. Cluster splitting still is improved by task affinities, but not significantly.
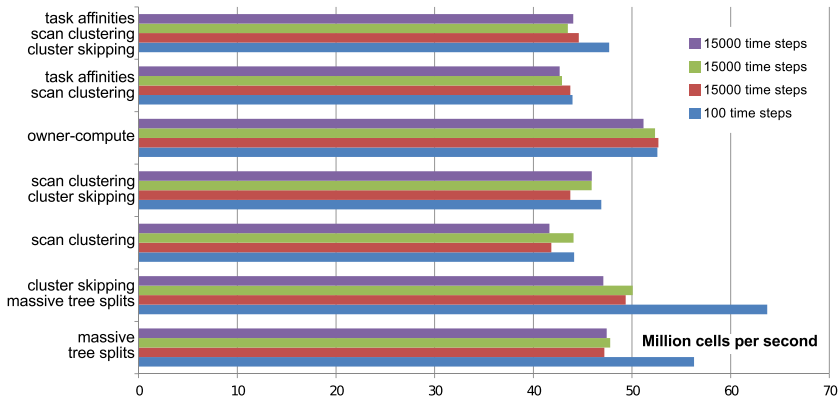
Fig. 9. Runtime per time step averaged over all algorithm phases. One measurement with 100 time steps, three samples with 15,000 time steps.

Though we assume the massive tree splits to yield good balancing due to work stealing, it is the only strategy that is not robust with respect to simulation time. We ascribe this to NUMA effects in combination with a touch-first data policy. In contrast, the owner-computes scheme outperforms the other strategies.

Our experiments reveal that for our setups, a equally balanced workload due to task stealing, e.g., is essential for the first grid traversals. When the grid changes significantly and the per-cluster workload as well as cluster distribution become inhomogeneous as well, affinity effects gain importance. A scheme that fixes the affinities due to a lack of task concurrency then outperforms task affinities assigned manually. The better work distribution with task stealing or equally cutting the SFC cannot compensate this.

7   Outlook

Future work comprises the development of an appropriate cost model anticipating skipping, workload inhomogeneity, and affinity issues. Furthermore, the interplay of the skip mechanism with a distributed memory parallelization is interesting as skips reduce cluster communication. Finally, we expect a better support of user-controlled affinity in programming languages and libraries. We are looking forward to use this or to contribute to this ourselves. Methodologically, an important locality-aware aspect for Invasive Computing [12] was created with forced affinities providing better performance for long simulation runs. On the application side, the present algorithms have to proof of value for implicit schemes where clusters and skips interplay with equation system solvers.

References

1. Bader, M., Böck, C., Schwaiger, J., Vigh, C.A.: Dynamically Adaptive Simulations with Minimal Memory Requirement - Solving the Shallow Water Equations Using Sierpinski Curves. SISC 32(1) (2010)
2. Bader, M., Rahnema, K., Vigh, C.: Memory-Efficient Sierpinski-Order Traversals on Dynamically Adaptive, Recursively Structured Triangular Grids. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 302–312. Springer, Heidelberg (2012)
3. Bartholdi, J.J., Goldsman, P.: Vertex-labeling algorithms for the hilbert spacefilling curve. Software: Practice and Experience 31(5), 395–408 (2001)
4. Borkar, S., Chien, A.A.: The future of microproc. Commun. ACM 54 (2011)
5. Bungartz, H.-J., Mehl, M., Weinzierl, T.: A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1064–1074. Springer, Heidelberg (2006)
6. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. SISC (3) (2011)
7. Dumbser, M., Käser, M., Toro, E.: An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes V: Local Time Stepping and p-Adaptivity. Geophysical Journal Int. 171(2), 695–717 (2007)
8. Eckhardt, W., Weinzierl, T.: A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 567–575. Springer, Heidelberg (2010)
9. Griebel, M., Zumbusch, G.: Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. Parallel Comp. 25(7), 827–843 (1999)
10. Küstner, T., Weidendorfer, J., Weinzierl, T.: Argument controlled profiling. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009 Workshops. LNCS, vol. 6043, pp. 177–184. Springer, Heidelberg (2010)
11. LeVeque, R.J., George, D.L., Berger, M.J.: Tsunami modelling with adaptively refined finite volume methods. Acta Numerica 20, 211–289 (2011)
12. Bader, M., Bungartz, H.-J., Schreiber, M.: Invasive computing on high performance shared memory systems. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing the Multicore-Challenge III 2012. LNCS, vol. 7686, pp. 1–12. Springer, Heidelberg (2013)
13. March, W.B., et al.: Optimizing the comp. of n-point correlations on large-scale astronomical data. In: Proc. of the Int. Conf. on High Perf. Comp., Netw., Stor. and Analysis, SC 2012. IEEE Computer Society Press (2012)
14. Meister, O., Rahnema, K., Bader, M.: A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In: De Boschhere, K., D'Hollander, E.H., Joubert, G.R., Padua, D., Peters, F. (eds.) Applications, Tools and Techniques on the Road to Exascale Computing. Advances in Parallel Computing, ParCo 2012, Gent, vol. 22, pp. 251–260. IOS Press (May 2012) ISSN: 0927-5452
15. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterog. arch. In: Proc. of the 2010 ACM/IEEE Int. Conf. for HPC, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society (2010)
16. Rüde, U.: Mathematical and computational techniques for multilevel adaptive methods. Frontiers in Applied Mathematics, vol. 13. SIAM (1993)

17. Sampath, R.S., Biros, G.: A parallel geometric multigrid method for finite elements on octree meshes. SISC 32(3), 1361–1392 (2010)
18. Schreiber, M., Bungartz, H.-J., Bader, M.: Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. In: IEEE Int. Conf. on High Performance Comp., HiPC (2012)
19. Schreiber, M., et al.: Generation of parameter-optimised algorithms for recursive mesh traversal algorithms (to be published, 2013)
20. Unterweger, K., Weinzierl, T., Ketcheson, D., Ahmadia, A.: Peanoclaw—a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperb. conservation law solvers. Technical report, Technische Universität München (2013)
21. Weinzierl, T.: A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids. Verlag Dr. Hut (2009)
22. Weinzierl, T., Mehl, M.: Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. SIAM Journal on Scientific Comp. 33(5), 2732–2760 (2011)