Shared Memory Parallelization of Fully-Adaptive Simulations Using a Dynamic Tree-Split and -Join Approach

Martin Schreiber
Technische Universität München
Munich, Germany
Email: martin.schreiber@in.tum.de

Hans-Joachim Bungartz
Technische Universität München
Munich, Germany
Email: bungartz@in.tum.de

Michael Bader
Technische Universität München
Munich, Germany
Email: bader@in.tum.de

***Abstract*—In this work we present an approach for the parallelization of hyperbolic simulations on shared-memory architectures running on fully-adaptive grids.**

We tackle the parallelization problem with a dynamic sub-tree split- and join-approach by running computations on those split sub-trees in parallel using lightweight tasks. The traversal of sub-trees created by tree-splittings is built upon an inherently cache efficient approach for solving hyperbolic PDEs on dynamically adaptive triangular grids using a Sierpiński space filling curve. Our communication scheme among sub-trees stores the exchange-data to/from adjacent sub-trees in a consecutive memory area which is further utilized for an improved run-length-encoded data exchange.

To give results for a concrete scenario, we implemented a solver for the shallow water equations which demands for fully-adaptive grid refinement and coarsening after each time-step. Our results give detailed statistics about optimization of the split size, parallelization overhead and also strong scalability results for a simulation running on multi-socket Intel and AMD architectures.

I. Introduction

Due to the current trends towards many-core architectures [1], efficient shared-memory parallelization is becoming a highly important aspect in numerical simulations, including the solution of partial differential equations (PDE) with a high computational demand. On the numerical side, the use of Discontinuous Galerkin methods, and also of classical Finite Volume methods, receives growing attention for solving hyperbolic equations on dynamically adaptive grids, especially for problems that involve shock-formation and wave-propagation, such as the shallow water equations (SWE) – a model used in Tsunami simulation, e. g. – and similar hyperbolic problems [2], [3]. As these models allow varying approximation orders per element and also dynamic choice of flux computation or Riemann solvers (e. g. those presented in [4], [5]), simulations will typically lead to unpredictable work-load per element, but also require adaptive grids in each time-step [6]. Also possible extensions – e. g. local-time-stepping (LTS) for hyperbolic PDEs [7], [8], especially for cluster-based LTS [9] – pose new demands on cluster-oriented storage schemes and an efficient parallelization of full-adaptive grids.

Traditional partitioning strategies strongly rely on the same or at least pre-computable work-load on each grid cell. This is no longer the case when considering the previously mentioned improvements of LTS, but also unpredictable workload – e. g. by using flux-solvers with an unknown number of Newton-iterations, skipping of adaptivity traversals for already conforming grids in a cluster or loading datasets from a database.

The Sierpiński space filling curve (SFC) and stack-based communications for cache optimized communication and computations is successfully used in serial processing [10], [11], [12]. Here we present an approach for the parallelization using massive tree-splitting methods which enables us to tackle the aforementioned problems. To achieve good load balancing, SFCs are usually cut into equally sized partitions for MPI communication (among others [13], [14], [15], [16], [17]). All those methods so far induced storage of additional data about adjacent cells in each cell or at least computing this adjacency information. We shifted the storage of such management data to a higher refinement-tree level to strongly reduce this computation and storage overhead. We utilize split- and join-operations on Sierpiński SFC sub-trees containing a bulk of grid cells for parallelization since they are natural borders when splitting the binary tree into tree branches. Therefore no modifications and thus no further overhead has to be introduced to the traversals of these sub-trees. Instead of creating only one sub-tree per core (e. g. [18]) we use *massive splitting* to overcome the load-balancing limitations created by such tree-splittings.

Our implementation is based on a Sierpiński SFC algorithm presented in [12] which is further outlined in Sec. II. The novel SFC parallelization concept is presented in Sec. III which uses a massive tree split- and join-approach on fully-adaptive grids. We show solutions for the parallel communications based on stacks and how to efficiently update the communication information about adjacent sub-partitions. Finally we present performance-benchmark results in Sec. IV.

II. Sierpiński traversal to solve SWE

We start with an introduction to our Sierpiński approach, which is based on newest vertex bisection grid-generation,

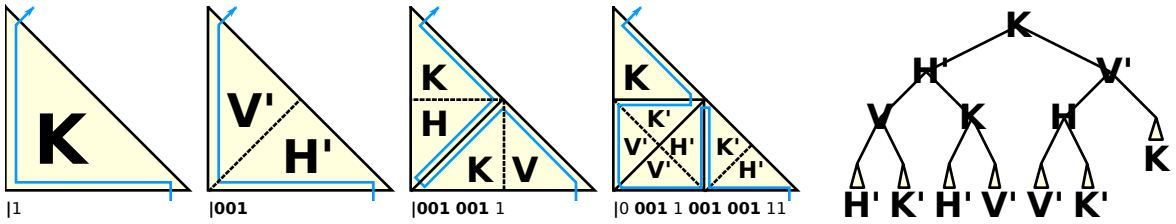

Fig. 1. Left images: Successive grid-refinement by applying the grammar given in Fig. 2. The bit string below each image represents the data stored on the structure-stack with "┫" marking the bottom of the stack. Refinement of a grid-cell is handled by replacing a "0" on the stack representing a tree-leaf with the string "001". Note that these replacement operations are done in a stream based manner.
Right image: structure-tree for the rightmost sketch of an adaptive grid.

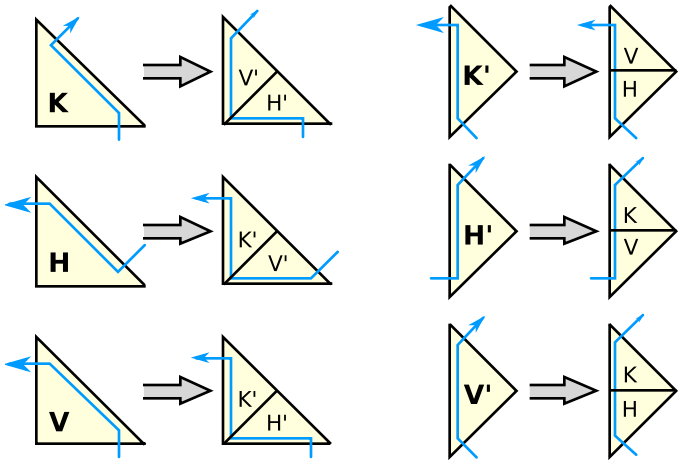

Fig. 2. Recursive description of 2D Sierpiński SFC. Refining a grid-cell is recursively defined by applying the corresponding rules. During refinement, the SFC itself is realigned close to the hypotenuse. This is later on useful for distinguishing right edges which are to the right side when following the SFC and left edges vice versa. *K*, *H* and *V* represent the so called even grammar whereas the stroked grammars *K'*, *H'* and *V'* are further denoted as odd grammar.

and show how to run cache oblivious traversals on such grids to present our parallelization of the original algorithm. This Section explains our approach on a triangular-shaped domain which is extended to a parallelization approach for arbitrary triangulated domains in the next Section.

A. Recursive definition

1) Grid structure: A tree-based recursive definition for the creation of grids based on the 2D Sierpiński SFC used in this work is given in Fig. 2. Cells are bisected according to certain patterns that reflect the local course of the SFC in this cells. For the *K* and *K'* pattern the curve enters the triangle via one of both legs and leaves it across the hypotenuse, for *H* and *H'* the triangle is entered across the *h*ypotenuse and leaves it across one of the triangle-legs. *V* and *V'* is given by the curve following a *V*-like scheme.

In the recursive definition the SFC is always drawn close to the hypotenuse. Later on, this is used to classify whether an edge is on the *left* or *right side*: Following the Sierpiński SFC, an edge is on the *left* side whenever its edge midpoint is on the left side of the curve and vice versa.

The grid-refinement information is stored on the so called *structure-stack*[1] in depth-first-order with 0 representing a tree-leaf and 1 an inner tree node to recursively follow the definition. During the grid-traversal, structure information stored on the *structure-stack* is successively removed from the top of the stack, thus giving us a stream based access. An example is given in Fig. 1 with the corresponding structure-stacks below each grid. During a refinement of a grid cell, 001 is streamed to the new structure-stack instead of a 0. For a coarsening, the operation is vice versa.

2) Element-data: The element-data needed to run the computation for each grid-cell is also stored on a stack using a *streaming-access*: only *pop-* or *push-operations* are allowed for a stack. During adaptivity traversals, an element-data is removed from the top of the *element-data-stack* for each leaf-cell. After processing, this element-data is pushed to an output *element-data-stack*. This requires one input- and one output-stack. Without considering adaptivity and thus no stack size variations, the *element-data-stack* elements are accessed by using non-destructive stack pop-operations handing back a reference to the element data stored on the stack. In this way the element-data is updated only, avoiding write-access to an additional element-data-stack. Therefore only a single stack is necessary, which is used for element-data input and output.

B. Edge-based communication

While the SFC is not used when only traversing the grid and accessing element-data, the locality and further properties becomes crucial for an efficient and inherently cache efficient exchange of data from one grid-cell to an adjacent one.

The SWE model we implemented (see Sec. II-D) relies on two different kinds of datasets: Element-data which is persistently stored for each triangle-cell on the *element-data-stack* and non-persistent data which is temporarily stored for edge-based data-exchange to/from adjacent triangle-cells. We handle the edge-based communication to/from adjacent triangles via so called *left* and *right stacks*. The nomenclature is based on the placement of the edge across which data is send to or received from the adjacent grid-cell. If the edge-midpoint is located on the left side from the point of view when following the SFC inside the triangle grid-cell and drawing the SFC close to the hypotenuse, the send- or receive-operation is

[1]this is not the stack-frame of programs but a software-managed stack

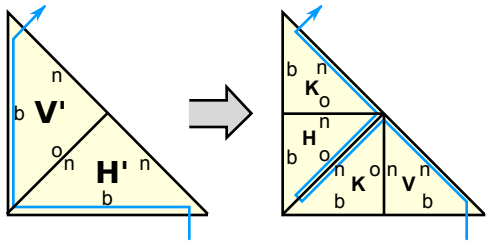

Fig. 3. Recursive inheritance of edge types. *n*: new edge, *o*: old edge, *b*: domain boundary. The boundary type is used to apply boundary conditions during the traversals.

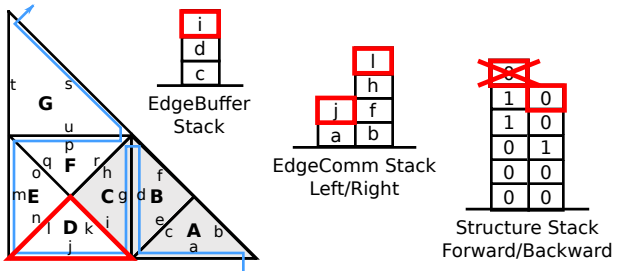

Fig. 4. Specific state during a forward traversal. The letters close to the edges represent different data-sets which have to be transmitted to adjacent triangles. Triangles with darker background were processed up to the current state. Red bold framed elements are modified in the current state of processing grid-cell D after reading a 0 from the structure-stack: First of all, i is popped from the right stack (edge type of right catheti is *old*) and pushed to the edge-buffer-stack. Then l is pushed to the right stack (edge type of left catheti is *new*) and j to the left stack (edge type of hypotenuse is *new*). To setup the structure-stack to a postfix-order for the backward traversal, the 0 is pushed to the backward structure-stack.

done via the left stack. Otherwise the stack which is used for communication is the right one.

Furthermore edge types are introduced by extending the recursive definition by edge types *new*, *old* and *boundary*. Each edge of the child inherits the edge type of the overlying edge of the parent. An example is given in Fig. 3. The edge type of the edge which is created by following the recursive definition is of type *new* for the triangle which is accessed at first following the SFC and *old* for the edge which is associated with the triangle accessed afterwards. This accounts for the communication way: Looking at Fig. 3, the first triangle *H'* creates *new* information and the second triangles *V'* reads the *old* information with the communication being managed via stacks. For edges of type *new*, exchange-data is pushed to the corresponding left or right stack while exchange-data is popped from the left or right stack when the edge type is *old*. Edges of type boundary are handled directly during the traversal by setting the edge communication data to appropriate boundary condition values.

To run a single time-step we run a forward- and backward-traversal: For a single element-wise computation based on the adjacent triangle edge data, a *forward-traversal* is used to store the edge data of type *new* to the respective left and right stacks. While during the forward traversal the data can be transmitted only to grid-cells which are accessed next when following the SFC, an additional backward traversal is used afterwards.

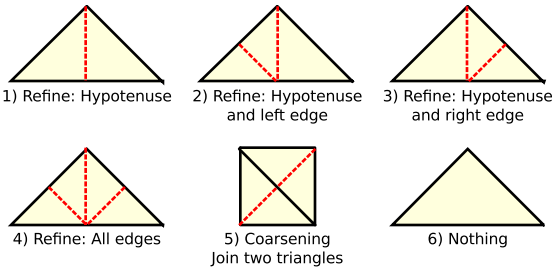

Fig. 5. All possible states which are stored on the *adaptivity-state-stack* during the refinement traversals.

To make the already forwarded data available during this backward traversal, an *edge-buffer-stack* is further used. Edge data read from edges of type *old* are pushed to the edge-buffer-stack to make them available for the computation during the backward traversal.

An example is given in Fig. 4 for a specific forward traversal step. Looking at the red framed triangle, the value 0 is read from the structure-stack. During the forward traversal, a new structure-stack for the backward traversal is also created allowing to execute a recursive backward traversal of the SFC by using post-order push operations to store leaf (0) and inner-node (1) information to the backward structure-stack. Starting the processing of a leaf element, the value 0 is pushed to the backward-structure-stack.

The edge data i which was previously stored from the adjacent triangle C is popped from the *right edge-communication-stack* and pushed to the *edge-buffer-stack*. Afterwards the new data j is stored to the left stack since the edge mid-point is to the left side of the SFC and the data l to the right stack.

During the *backward traversal*, all edge-communication-data from adjacent triangles are available via the left and right stack as well as the edge-communication-buffer. Loading references to this edge-communication-data, the computation for processing of the element local data is started.

C. Adaptivity traversals

Adaptivity traversals are used for refinement- and coarsening-operations on grid-cells. After this adaptivity operations, the new grid has to fulfill the property of not having any hanging node – e. g. created by the red dashed edge in the left image in Fig. 6 – and the element-data being stored along the SFC. During the first forward adaptivity traversal, the triangle grid-cells are refined to reduce numerical discretization errors or are coarsened by joining with an adjacent triangle. Possible adaptivity states which are in particular refinements, coarsenings and no adaptivity are given in Fig. 5. Those states are stored on the *adaptivity-state-stack* with one element associated to exactly one element-data-stack element.

To avoid hanging nodes, as many forward- and backward-traversals as are necessary are executed with refinement information being forwarded via the edge-based communication schemes presented in the previous section (see Fig. 6 for simple example). A refinement request is sent via those edges where a new node is going to be created due to an inserted

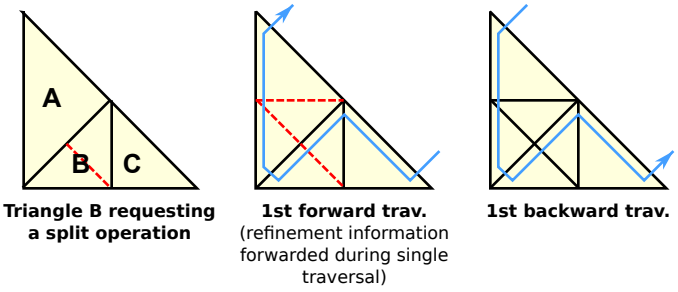

Fig. 6. Adaptivity traversals to create conforming grid by avoiding hanging nodes. Local refinement request was triggered for grid-triangle B (refinement state 1). Due to the new hanging node created by the new edge, the refinement request is forwarded to triangle A (refinement state 3).

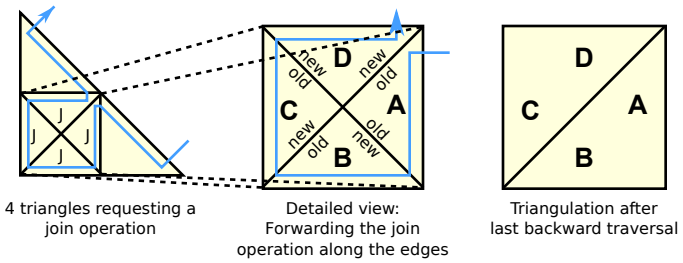

Fig. 7. Four grid-triangles requesting a *J*oin operation.

edge. The adjacent triangle then reads this adaptivity information and updates its adaptivity state to avoid the hanging node. Those traversals are executed until the adaptivity-states-flags represent a grid without hanging nodes.

For coarsening operations, an *agreement protocol* is used as shown in Fig. 7. Coarsening requests have to be transmitted across both triangle-leg edges. Without loss of generality, we only consider triangle-leg edge types of type *old* and *new*. For the first grid cell *A*, both catheti are of type *new*, forwarding the coarsening request across both catheti edges. As long as all other grid cells (*B* and *C*) also request a coarsening, the original coarsening request of *A* is forwarded to *D*. Since all edge types of triangle-element *D* are of type *old*, both forwarded requests have to request a coarsening. Otherwise *D* does not agree to the coarsening. During the next traversal in the reversed direction, this information is only forwarded when all four triangle-elements agree to the coarsening operation.

Finally, a backward traversal reads the element-data from the *input-element-data-stack* and stores respectively either interpolated or restricted element-data to the *output-element-data-stack* depending on the corresponding adaptivity-state (see Fig. 5) stored on the *adaptivity-state-stack*.

D. Application

To test our approach and to give concrete results for a simulation, a shallow-water-equation (SWE) model with explicit Euler time-stepping and constant bathymetry was implemented. Here we give a very short overview of the derivation of the discretized Discontinuous Galerkin (DG) method with local Lax-Friedrichs (LxF) fluxes to simulate such a system (see [19], [20], [21] for more details). For each grid cell as many state vector as there are degree of freedoms (DOFs) in a single triangle are stored to the element-data. For the 0^{th} order basis functions there is only a single state vector located in the middle of the triangle. We also implemented the 1^{st} order DG method with a state vector at each midpoint of an edge (Crouzeix-Raviart nodal points [22]). This state vector U stores the height h of the surface, the height averaged momentum $\vec{u} = (u_x, u_y)^T$, and also the bathymetry b which is constant in our simulation and thus neglected in the derivation: $U = (h, u_x, u_y, b)^T = (h, v_x h, v_y h, b)^T$. The velocity can be directly computed by $v_i = u_i/h$. Furthermore the SWE in its homogeneous form is given by conservation law of hyperbolic equations

$$\frac{dU}{dt} + \nabla \cdot F(U) = 0$$

with $F(U) = \left(\vec{v}h, \vec{v}v_x h + \frac{1}{2}gh^2 e_x, \vec{v}v_y h + \frac{1}{2}gh^2 e_y\right)^T$ as the flux-function and the basis vector $\vec{e}$. By applying the Gaussian divergence theorem we get the weak formulation

$$\underbrace{\int_T \frac{dU}{dt}\varphi_i dT}_{\text{mass-term}} - \underbrace{\int_T F(U)\cdot\nabla\varphi_i dT}_{\text{stiffness-term}} + \underbrace{\oint_T F(U)\varphi_i \cdot ds}_{\text{flux-term}} = 0 \tag{1}$$

with T representing a triangle grid cell. Next, we substitute the solution U with basis-function approximated solution $\tilde{U} = \sum_{i=1}^{n} \tilde{U}_i \varphi_i$ and evaluate the integral over the mass-term to obtain a constant c. Using an explicit Euler time-step the update for a single time-step Δt can then be expressed by the discretized stiffness-term and the flux-term $\tilde{U}_i(t+\Delta t) = \tilde{U}_i(t) + c\cdot(\mathcal{S}(\tilde{U}(t)) + \mathcal{F}(\tilde{U}(t), \tilde{U}^+(t)))$ with $\tilde{U}^+$ representing the closest state vector in the adjacent cell. While $\mathcal{S}$ can be precomputed, the flux-term which represents the exchange of mass and momentum crossing the edge to/from the adjacent cell has to be computed in conjunction with flux-data from the adjacent cell. Various flux-solvers have been developed which compute the fluxes with different quality and computational costs. We used the local Lax-Friedrichs flux [19] with dynamically chosen viscosity coefficient for our test cases which introduces a numerical diffusion for stability reasons. By using the property of rotational invariance of the SWE [23] the velocity components of each state vector are projected to the corresponding edge-normal space before applying the flux computation. Therefore using this specific flux computation in our framework does not restrict implementation of other improved Riemann solvers, such as the one presented in [5].

E. Traversators & Kernels

In our software framework, we organize the algorithm via *traversators* and *kernels*. The term *traversator* accounts only for the parts of the traversal which are responsible for the edge-communication-data management, pushing and popping data from stacks, to read and write data to streams but also to follow the recursive definition. The *kernels* itself are responsible to offer interfaces providing storing edge-communication-data, running element-data and edge-communication-data based

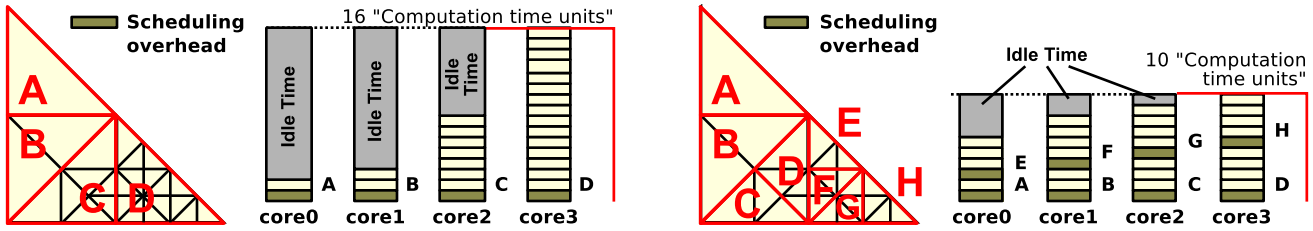

Fig. 8. Left image: Parallelization by creating $\#cores$ sub-partitions. Right image: Parallelization by creating by far more splits than there are cores available on the system.

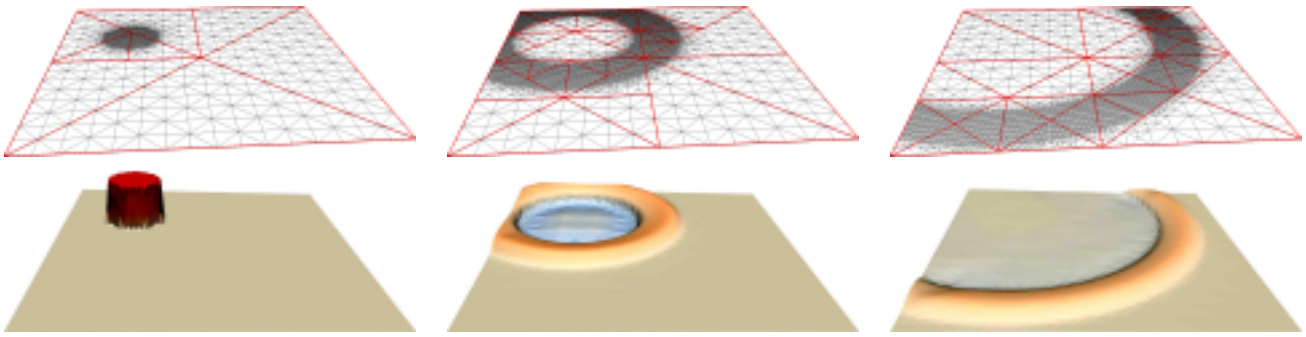

Fig. 9. Radial breaking-dam simulation scenario with initial depth of 6 and maximum refinement depth of 12. Sub-partitions are split when the number of grid cells exceeds 512. Left handed images: Fine grid cells in black and sub-partitions in red. Right handed screenshots: Extracted water surface of simulation.

computations and also refinement- or coarsening-requests for adaptivity traversals.

While *traversators* and *kernels* are completely disjunct, the implementation of a SFC tree-traversal as it is used to run a single time-step update is instantiated with a *kernel* defined by the user and the corresponding *traversator* offered by the framework.

III. Parallelization

A. Massive workload creation

A traditional way to parallelize PDEs is to cut the data stored along the SFC into #cores pieces with equally sized lengths (among others [13], [14], [15], [16], [17]). In this work, we utilize a different way to allow a parallelization: Splitting the original tree many times at the new tree root. To our knowledge, parallelization with a tree-splitting approach was so far only done by splitting the grid-tree into as many partitions as there are cores installed on the system ([18]). This would create a high idle time for unbalanced grids (left image in Fig. 8).

To overcome this limitation our approach creates by far more partitions, further denoted as sub-partitions, than there are cores available on the system – a massive workload splitting (Right image in Fig. 8 and the red framed sub-partitions in Fig. 9).

B. Task-based parallelization

To be ready for upcoming many-core systems, operations executed for each sub-partition have to be created very fast in parallel. We store the sub-partitions at the leaf-nodes in a *binary tree* (see right image in Fig. 8 for initial domain-triangulation tree). One task is executed for each tree node. Assuming a regular balanced sub-partition tree, this would lead to 2^n number of tasks running operations on leaf-nodes and thus sub-partitions within n recursions executed in parallel.

C. Sub-partition

For the parallelization of our SFC-based massive splitting approach, we aimed at the following three goals:

(1) *Independence*: Sub-partitions can be scheduled in arbitrary order to allow task based work-stealing. (2) *Cache efficiency*: Operations local to sub-partitions are handled in an as cache efficient way as possible. (3) *Read-only access to other sub-partitions*: Actions executed on a single sub-partition do not access other sub-partitions by using write operations. This becomes a crucial component once extending this implementation to MPI using push and pull MPI access schemes.

D. Skeleton of sub-partition

A single sub-partition is implemented in an object-oriented manner providing the following data-structures and features:

Local stacks: Used to run computations on sub-partitions in arbitrary sub-partition order. Structure stacks (forward/backward) store the adaptive triangle structure (Sec. II-A1), element-data-stacks (forward/backward) store element local data (Sec. II-A2), edge-communication-stacks (left/right) (Sec. II-B) and exchange-edge-communication-stacks (left/right) (Sec. III-G) exchange simulation data with adjacent sub-partitions, edge-buffer-stacks store non-persistent edge communication data for the backward traversal (Sec. II-B), adaptive-edge-communication-stacks (left/right) and exchange-adaptive-edge-communication-stacks (left/right)

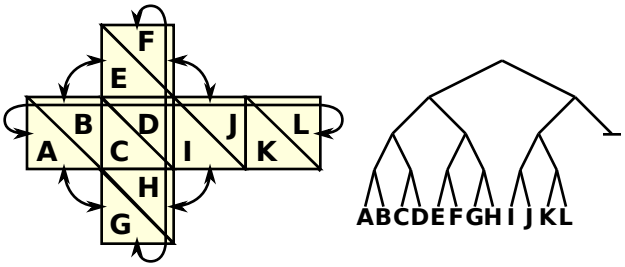

Fig. 10. Initial domain triangulation with root triangles and periodic boundary with the corresponding generic sub-partition tree.

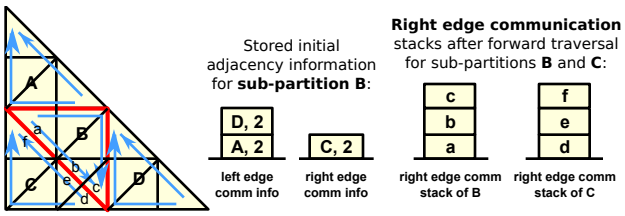

Fig. 11. After traversal of the sub-partitions B and C, the edge-communication datasets on the hypotenuse are stored consecutively on the respective stack. The RLE information about edge communication data for the sub-partition can then be used to know the size and starting position (in case of multiple adjacent sub-partitions) of the block which is read from the adjacent sub-partition to the local stacks.

are used to exchange adaptivity information data with adjacent sub-partitions (Sec. II-C), adaptive-state-stack storing the current adaptivity state to create a conforming grid (Sec. II-C).

Edge communication information (ECI): Information about adjacent sub-partitions and the edge-communication-data which has to be read from the adjacent sub-partitions (Sec. III-E, ff.).

SplitJoinInformation: Information to run tree-split/-join operations on this sub-partition (Sec. III-I, ff.).

Access structures to sub-partition binary tree structure to read data from adjacent sub-partitions for split/join operations and for reading required exchange data.

The application developer extends this *sub-partition skeleton* by *user-defined kernels and compatible traversators*.

E. Edge communication information (ECI)

Starting with an *arbitrary initial domain triangulation* (example given in right image in Fig. 10) adjacent sub-partitions are determined automatically either by comparing the edge vertices or by user-driven setup. For each sub-partition, one ECI dataset is created for the sub-partitions next to the hypotenuse and another one for the two sub-partitions next to both triangle legs – each accounting for edges shared with adjacent triangles either on the left or right side of the SFC curve.

F. Run-length encoded edge communication information (RLE ECI)

Each ECI is set up with a list of RLE ECI's. Working with stack-based edge communication for grid traversal, the data which has to be read by another sub-partition is stored in a consecutive memory area on one of the edge-communication-stacks. This is achieved by setting the outer sub-partition edge types to *new* before executing a traversal (see Fig. 11). To make use of this consecutive storage we use a run-length-encoding (RLE) to store information about the number of shared edges to an adjacent sub-partitions in the ECI. Knowing the number of elements which have to be transferred, improved code is used for exchange of edge-communication data between sub-partitions. See Fig. 12 for an example of the RLE ECI in the second column.

G. Exchange of data between sub-partitions

Considering only *new* and *old* edge types without loss of generality, a forward and backward traversal including inter-sub-partition communication-data is handled in 3 steps: 1) For the *forward traversal*, the edge types of the sub-partition boundaries are set to *new*, thus writing the communication data to the local edge-communication-stacks. In this way, edge data which has to be read by adjacent sub-partitions is stored consecutively on the stacks for the forward traversal. 2) Then the data is *exchanged by copy operations* from edge-communication-stacks of adjacent sub-partitions to local exchange-edge-communication-stacks using local- and adjacent-ECI. 3) Finally for the *backward traversals*, the boundary types are set to *old* with the input edge-communication-stacks set to exchange-edge-communication-stacks. In this way the backward traversal is reading the edge-communication-data of the adjacent sub-partitions.

H. Updating the edge-communication information for adaptivity traversals

Updating the ECI is based on the last backward adaptive traversal writing the refine/coarsen information to the *adaptive-edge-communication-stacks*. Fig. 12 illustrates an example of such a traversal process. If a new node has to be created on an edge due to a refinement, the refinement request R is written to the corresponding adaptivity-edge-communication-stack, otherwise the value 0. In case of coarsening a triangle, the coarsening identifier C is written to the stack only for both triangle legs. Using this identifiers on the *adaptive-edge-communication-stacks*, the ECI is updated after each adaptive pass: For each ECI, as many elements storing adaptive information from the *adaptive-edge-communication-stacks* are read to match the currently stored ECI. When a coarsening information was read, an additional coarsening information has to follow and the RLE counter is decreased by 1. Whenever a refinement information was read, the RLE counter has to be increased by 1.

I. Tree-split and -joins for parallelization on a static grid

We start with the description of the tree-split and -joins for static grids without considering refinements and coarsenings of grid cells. Without loss of generality, we describe the algorithm on the backward traversal for an even recursion method (e. g. K in Fig. 2) without domain boundaries next to the sub-partition (Fig. 13). During each split- or join-operation, the ECI of

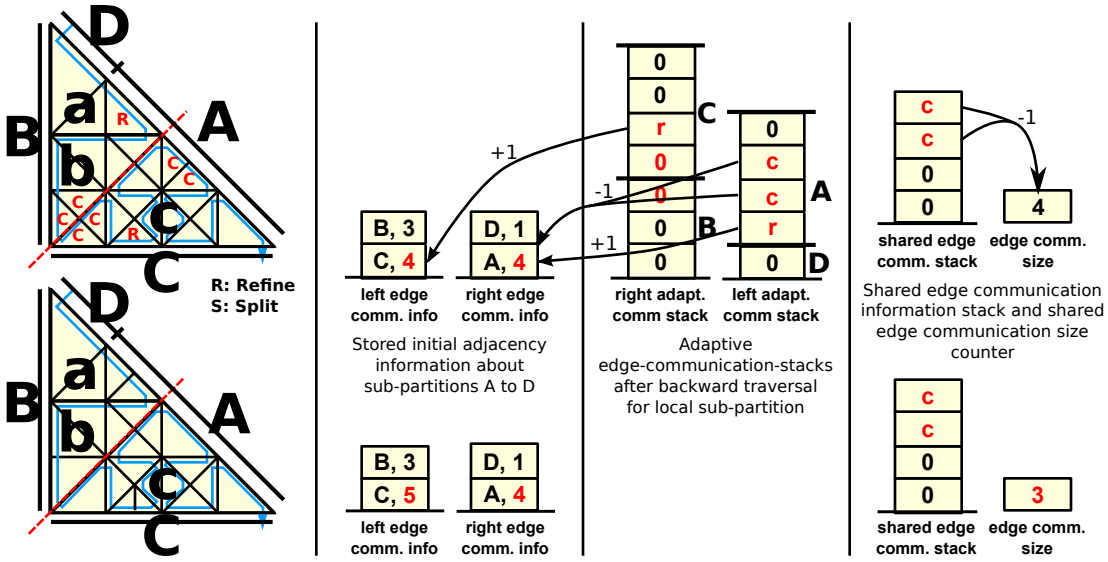

Fig. 12. Sketch of the last backward traversal during the adaptivity pass. After the traversal, the refining and coarsening information on the left and right adaptivity-edge-communication-stacks are used to update the left and right ECI.

each split or joined sub-partition has to be modified to be in a consistent state to match the border edges of the adjacent sub-partitions. Also the number of elements on the *shared stack* has to be known, which represents the number of shared edges of both split sub-partitions. To gain this implicitly stored information of a sub-partition, we implemented separated traversals of areas **a**, **b** and **c** (see Fig. 13). The sub-partition boundary edge types were also set to *new* to get the required information pushed to the edge-communication stacks:

After the traversal of area **a** is finished and the data for the adaptive-edge-communication-stacks was stored on the adaptive-edge-communication-stack, the algorithm remembers the number of elements on the left and right stack. Running the next traversal for area **b**, the number of the elements on the *shared stack* can be determined using the previous information about the number of elements on communication stacks after traversal of area **a** and the currently stored number of elements on the stack. After executing the methods for area **c**, all information for the split- or join-operation can be finally determined to set up the new ECI.

J. Split and joins for parallelization with dynamic adaptivity

The determination of the split- and join-information is implemented in the last adaptivity backward traversal to avoid an additional traversal. Therefore also the information which was described in Section III-I would be non-consistent since this was determined with the assumption that the traversal is executed on a static grid. When considering adaptivity, adaptive refinement- or coarsening-information is stored to adaptive-edge-communication-stacks if there is a refine- or coarsening-operation concerning the edge. This information is used to update the split/join information described in the previous Section III-I to create the correct RLE ECI in case of a split/join operation. To know where to split the *structure-* and *element-data-stacks*, also the number of currently stored items on the *element-data-stack* has to be known after traversal of **b**.

Updating the split/join information about the shared edges has to be handled directly after **b** was processed (see Fig. 13). Since the number of elements stored on the adaptive-edge-communication-stacks represent only the state before the adaptivity, the number of adaptively changed elements on the shared stack has to be determined immediately. Otherwise the traversal of **c** would modify this adaptivity information to a possibly inconsistent state. By analyzing the respective elements on the corresponding edge-communication-stack, the counter for the shared edges is updated.

K. Updating ECI after split or join of adjacent sub-partitions

After split and/or join operations on adjacent sub-partitions the local ECI has to be updated to match the newly generated or removed adjacent sub-partitions. To know whether an adjacent sub-partition was split or joined with another sub-partition, the adjacent sub-partition skeleton is extended by a *transfer-state-flag* with the following possible values: *NO_TRANSFER* is used to flag sub-partitions with no split/join operation during the last time-step. A sub-partition node which was split during the last time-step is flagged with *SPLIT_PARENT* while *SPLIT_CHILD* is used for a child which was created by a split operation. A node which was created by a join operation is marked with *JOINED_PARENT* and respectively *JOINED_CHILD* is used for a node which was joined with its sibling.

Since a detailed description of all possible (split $\times$ join $\times$ none) constellations would be out of the scope, only the constellation with both adjacent sub-partitions being split is further explained. Each child which was created by a split operation in the sub-partition tree looks up the adjacent sub-partitions RLE ECI using the sub-partition ID of its parent node. This is important since the adjacent sub-partition has

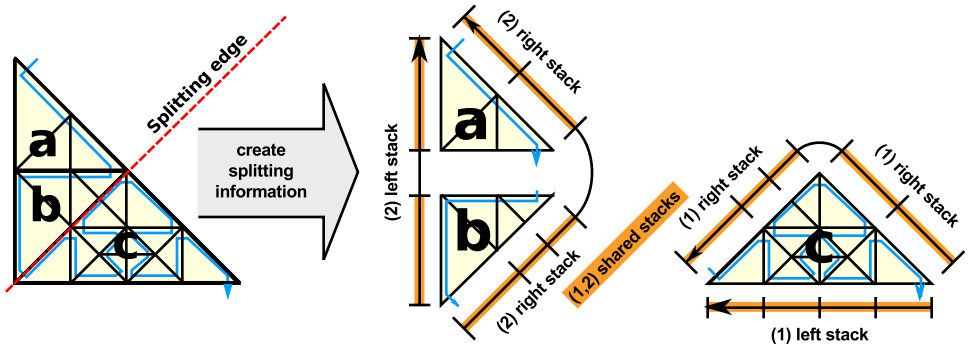

Fig. 13. Separate execution of recursion methods *A*, *B* and *C* to determine elements stored on right, left and shared stack. Since this is integrated in the last backward traversal of the adaptivity pass, the left and right stacks are exchanged and given in forward notation.

no knowledge about the adjacent split operation. With the adjacent transfer state set to *SPLIT_PARENT*, the adjacent sub-partition was split as well. This requires assembly of the new local ECI by looking up the ECI of both newly created adjacent children. The traversal order of both children plays a crucial role: depending on whether the first child in SFC order searches for the corresponding communication partner or the second child, the adjacent sub-partition tree has to be traversed in reversed order while updating the ECI.

IV. Results

We tested our implementation on 2 different shared-memory platforms. *Platform AMD* contains 4 AMD Opteron 6276 processors with 8 Bulldozer modules per CPU (16 threads) and *platform Intel* has 4 Intel Xeon CPUs (E7-4850@2.00GHz with 10 cores per CPU / 20 threads per CPUs). For our model we used a 1st order DG method with the flux being computed using the local Lax-Friedrich flux (Rusanov's method) approximation.

Our simulation setup is as follows: A flat sea-ground 300 meters below the sea surface is used with a domain length of 5000 m. Up to 8 levels of refinements are allowed with the refinement/coarsening triggered respectively with the surface above/below 300.02/300.01 meter leading to a highly adaptive behavior. We used two triangles setting up a square sized domain centered at the origin. The initial grid-resolution is given by d representing the initial adaptive passes forcing refinement operations on all triangle-cells. Thus 2^{d+1} triangle-cells are set up during grid-initialization for the square sized domain. The initial radial dam is centered at (-1250 m, 1000 m), a radius of 250 m and its surface to be 301 meter above the ground (see Fig. 9). The sub-partition join-threshold is set to be the half of the split-threshold. Two sub-partitions sharing the same parent and undershooting the join-threshold with their number of grid-triangles are joined. Performance results are given in "Million Triangles Per Second" (MTPS). The number of triangles during the simulation for this scenario varies over time and is given in Fig. 14.

A. Optimal splitting size

Choosing specific splitting sizes – at which number of grid-triangles the sub-partition has to be split – has influence on the

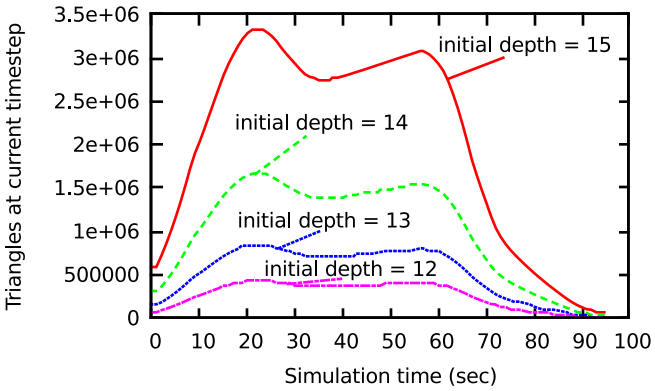

Fig. 14. Triangle distribution for different initial domain resolution.

scalability and performance. Therefore we give an overview of the relationship between the splitting size and the impact to the overall runtime. By *decreasing the sub-partition size* a *better load-balancing* is achieved in average. However this would also lead to *more sub-partitions* increasing the number of tasks and thus the overall *task execution overhead*. Since more sub-partitions are created, also the *overhead of exchange of additional edge-communication data* is increased. The overall cache efficiency is also reduced by running operations on several sub-partitions which are not stored in SFCs across sub-partitions.

Performance charts for a simulation running on *platform Intel* using different splitting sizes and problem sizes are given in Fig. 15. For simulations running on a domain with higher number of grid-cells, the performance is increased in average for increasing the splitting size.

B. Parallelization overhead

To test the parallelization overhead, we ran 7 different benchmarks (Fig. 16). The simulation parameters were set to the same as for the square sized benchmark but with the rightmost triangle removed. Thus the domain is set up only by a single root triangle to allow a serial execution on one sub-partition. An initial depth of d represents 2^d initial grid triangle cells. The MTPS of this benchmark were divided by

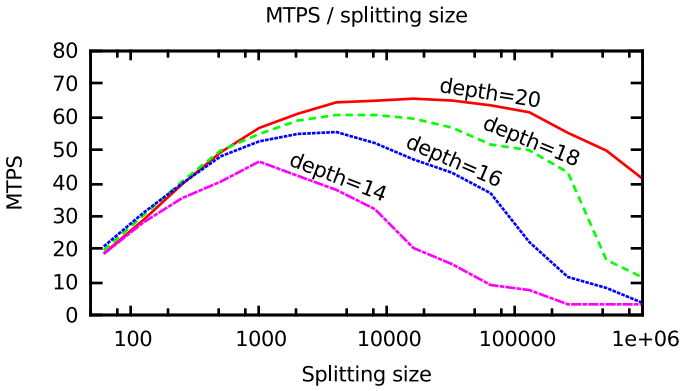

Fig. 15. MTPS for given initial refinement depth and different splitting size running on 40 cores on *platform Intel*.

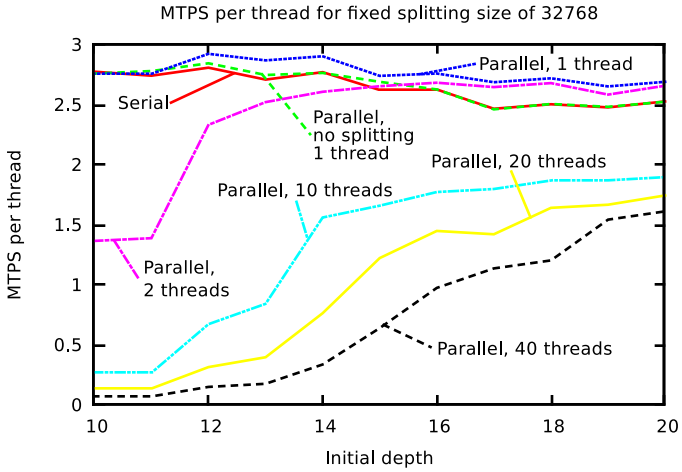

Fig. 16. MTPS per thread for serial and parallel simulations. The domain is set up with a single domain triangle. A fixed splitting size for the sub-partitions of 32768 was used.

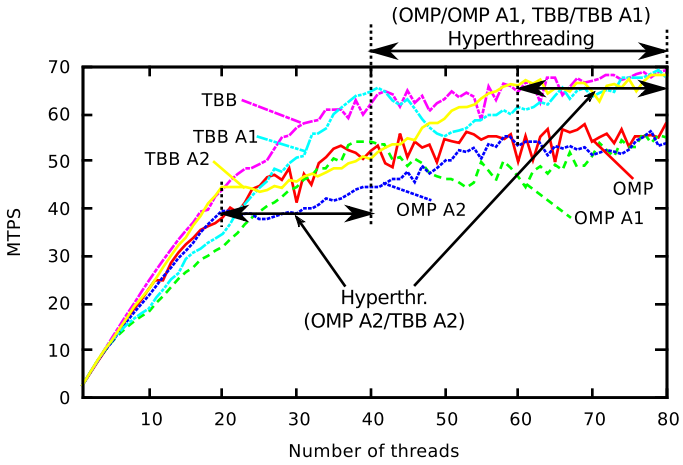

Fig. 17. Strong scaling results for *platform Intel* with 40 cores and 40 additional hyperthreaded cores.

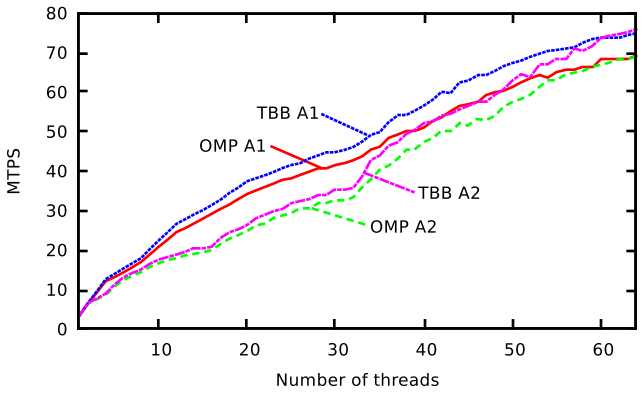

Fig. 18. Strong scaling for platform AMD with 64 cores.

the number of used threads for better comparison. The serial version was executed on a single core only.

The *Serial* version represents the version with all parallelization in the code disabled. The process was bound to run on the first core on the first socket only.

The version *Parallel, no splitting, one thread* was executed on a single core with sub-partition split threshold set to infinity avoiding any splits to determine the generic overhead of the additional layer. This version creates MTPS close to the non-parallel version.

Furthermore we benchmarked the parallel version with a splitting-size of 32768 running on different numbers of core. Due to overheads described in Sec. IV-A an overhead is visibly introduced to the parallel version with split sub-partitions.

Even if the sub-partition tree has to be traversed in the previous benchmark, there may be still no overhead for the task creation included which is considered in the next benchmark (*Parallel, 2 threads*). This benchmark was executed with 2 threads to force the creation of tasks including the overhead instead of directly executing the task which is the case for a single thread. Due to the splitting size which exceeds the number of grid cells in the domain, no parallelization speed-up is obtained for 2 threads and such small problem sizes.

C. Strong scaling

Results for strong scaling of *platform Intel* are presented in Fig. 17 and for *platform AMD* in Fig. 18. This simulation was initialized with an initial refinement depth of 20 with a maximum relative refinement depth of 8 for setup of the radial dam-break scenario. 100 time-steps were computed with 19.0 Mio. grid cells processed in average per time-step.

Core-affinities were set for simulations marked with *A1* or *A2*. For A1, the threads were pinned linearly to the cores. For *platform Intel* this pins the first 10 threads to the first 10 cores of the CPU on socket 1, the next 10 threads to the CPU on the 2nd socket, etc. Affinities for threads 41 to 80 are set in the same way which utilizes the hyper-threading technology.

We implemented the parallelization using Threading Building Blocks[2] (TBB) tasks and OpenMP tasks.

For *platform Intel*, the scalability is increased by using *TBB* without affinities and *TBB A1* setting thread affinities. Pinning the threads to cores with the *A2* scheme uses the same cores for

[2]threadingbuildingblocks.org

threads 1 to 20 as for hyperthreaded threads 21 to 40 (*TBB A2*). Therefore the scalability decreases for the simulation utilizing threads 21 to 40 and thus hyperthreaded cores, increasing again for threads 41 to 60 and decreasing for threads 61 and 80.

Another parallelization was implemented by using OpenMP tasks with the *untied* clause for each sub-partition tree node (see Sec. III-B). Due to known issues for OpenMP task constructs (see e. g. [24]), the scalability is not as good for unbalanced trees as by using TBB for parallelization.

For *platform AMD* scalability graph could be obtained similar to the *platform Intel* comparing only the non-hyperthreaded scalability graphs. Also the break in the scalability graph at the half of the maximum number of threads for *TBB A2* is also visible for this pinning scheme due to the shared floating-point units for 2 particular cores in this architecture.

V. Summary and outlook

In this work we presented a new SFC-based parallelization approach for shared memory systems. Compared to existing SFC splitting approaches, *massive splitting* provides an alternative for parallelization on shared-memory architectures.

Since all sub-partitions simply represent clusters, an extension to cluster-based local-time-stepping methods is part of our ongoing work. Also skipping adaptive traversals for already consistent sub-partitions is expected to improve the run-time.

Further improvements are expected to be achieved by NUMA aware allocation of memory, improved task execution using thread affinities and NUMA aware execution of those tasks. For improved scheduling, a priorization of tasks based on the computational workload on the sub-partition and other ways to split the domain are further assumed to reduce the CPU idle time. Also the utilization of SIMD commands to evaluate fluxes and element data operations for several triangle-cells is one of the most challenging parts for fully-adaptive grids. Since all accesses to adjacent partitions are executed in a read-only manner, also an extension to distributed memory systems using MPI is part of our ongoing research.

Since there are many ways to implement the presented algorithm and to allow a reproducibility of our results, a publication of source-code seems to be mandatory. Therefore we released the source-code on *http://www5.in.tum.de/sierpi/* with version *2012-05-14* being the basis of this paper.

Acknowledgement

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

We like to thank the Institute for Multiscale Simulation, Friedrich-Alexander Universität Erlangen-Nürnberg, for giving us access to their AMD cluster.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, 2011.

[2] N. Chevaugeon, J. Xin, P. Hu, X. Li, D. Cler, J. E. Flaherty, and M. S. Shephard, "Discontinuous Galerkin methods applied to shock and blast problems," *J. Sci. Comput.*, vol. 22-23, June 2005.

[3] W. Lai and A. A. Khan, "A discontinuous Galerkin method for two-dimensional shock wave modeling," *Model. Simul. Eng.*, vol. 2011, Jan. 2011.

[4] P. Liu, *Advanced numerical models for simulating tsunami waves and runup*, 2008.

[5] D. L. G. Randall J. LeVeque and M. J. Berger, *Tsunami modelling with adaptively refined finite volume methods Acta Numerica*, 2011.

[6] S. Pranowo, J. Behrens, J. Schlicht, and C. Ziemer, *Adaptive mesh refinement applied to tsunami modeling: TsunaFLASH*, 2010.

[7] M. O. Domingues, S. M. Gomes, O. Roussel, and K. Schneider, "Spacetime adaptive multiresolution methods for hyperbolic conservation laws: Applications to compressible Euler equations," *Applied Numerical Mathematics*, vol. 59, no. 9, 2009.

[8] M. Dumbser, M. Käser, and E. Toro, "An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes V: Local Time Stepping and p-Adaptivity," *Geophys. J. Int.*, vol. 171, no. 2, 2007.

[9] C. E. Castro, M. Käser, and E. F. Toro, "Space-time adaptive numerical methods for geophysical applications," *Roy. Soc. Phil. Trans. A*, vol. 367, oct 2009.

[10] M. Bader, S. Schraufstetter, C. Vigh, and J. Behrens, "Memory Efficient Adaptive Mesh Generation and Implementation of Multigrid Algorithms Using Sierpinski Curves," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, 2008.

[11] M. Bader and C. Zenger, "Efficient Storage and Processing of Adaptive Triangular Grids Using Sierpinski Curves," in *Computational Science – ICCS 2006*, ser. Lecture Notes in Computer Science, vol. 3991. Springer, May 2006.

[12] 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*, vol. 32, no. 1, 2010.

[13] J. Behrens and J. Zimmermann, "Parallelizing an Unstructured Grid Generator with a Space-Filling Curve Approach," in *EURO-PAR 2000*. Springer, 2000, pp. 815–823.

[14] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, "Dynamic Octree Load Balancing Using Space-Filling Curves," Williams College Department of Computer Science, Tech. Rep. CS-03-01, 2003.

[15] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New Challenges in Dynamic Load Balancing," vol. 52, p. 2005, 2004.

[16] W. F. Mitchell, "A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids," *Journal of Parallel and Distributed Computing*, vol. 67, no. 4, 2007.

[17] M. B. O. Meister, K. Rahnema, "A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids, Advances in Parallel Computing, IOS Press," in *Proc. of the International Conference on Parallel Computing (ParCo 2011)*, 2011.

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

[19] R. J. LeVeque, *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press , 2002.

[20] V. Aizinger, "A discontinuous Galerkin method for two-dimensional flow and transport in shallow water," *Advances in Water Resources*, vol. 25, pp. 67–84, 2002.

[21] J.-F. Remacle, S. S. Frazo, X. Li, and M. S. Shephard, "An adaptive discretization of shallow-water equations based on discontinuous Galerkin methods," *International Journal for Numerical Methods in Fluids*, vol. 52, no. 8, 2006.

[22] M. Crouzeix and P. A. Raviart, "Conforming and non-conforming finite element methods for solving the stationary Stokes equations," *RAIRO Anal Numer*, vol. 7, 1973.

[23] E. Toro, *Shock-capturing methods for free-surface shallow flows*. John Wiley, 2001.

[24] S. Olivier and J. Prins, "Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs," in *Evolving OpenMP in an Age of Extreme Parallelism*, ser. Lecture Notes in Computer Science, M. Müller, B. de Supinski, and B. Chapman, Eds. Springer Berlin / Heidelberg, 2009, vol. 5568, pp. 63–78.