Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves

Tobias Weinzierl[*]
School of Engineering and Computing Sciences
Durham University
Durham DH13 LE, GBR
tobias.weinzierl@durham.ac.uk

Roland Wittmann
Informatics, TUM
wittmanr@in.tum.de

Kristof Unterweger
Department of Informatics
Technische Universität
München (TUM)
85748 Garching, GER
unterweg@in.tum.de

Michael Bader
Informatics, TUM
bader@in.tum.de

Alexander Breuer
Informatics, TUM
breuera@in.tum.de

Sebastian Rettenberger
Informatics, TUM
rettenbs@in.tum.de

ABSTRACT

Spacetrees are a popular formalism to describe dynamically adaptive Cartesian grids. Though they directly yield an adaptive spatial discretisation, i.e. a mesh, it is often more efficient to augment them by regular Cartesian blocks embedded into the spacetree leaves. This facilitates stencil kernels working efficiently on homogeneous data chunks. The choice of a proper block size, however, is delicate. While large block sizes foster simple loop parallelism, vectorisation, and lead to branch-free compute kernels, they bring along disadvantages. Large blocks restrict the granularity of adaptivity and hence increase the memory footprint and lower the numerical-accuracy-per-byte efficiency. Large block sizes also reduce the block-level concurrency that can be used for dynamic load balancing. In the present paper, we therefore propose a spacetree-block coupling that can dynamically tailor the block size to the compute characteristics. For that purpose, we allow different block sizes per spacetree node. Groups of blocks of the same size are identified automatically throughout the simulation iterations, and a predictor function triggers the replacement of these blocks by one huge, regularly refined block. This predictor can pick up hardware characteristics while the dynamic adaptivity of the fine grid mesh is not constrained. We study such characteristics with a state-of-the-art shallow water solver and examine proper block size choices on AMD Bulldozer and Intel Sandy Bridge processors.

[*]Corresponding author.

HiStencils 2014
First International Workshop on High-Performance Stencil Computations
January 21, 2014, Vienna, Austria
In conjunction with HiPEAC 2014.

http://www.exastencils.org/histencils/2014/

1. INTRODUCTION

In this paper, we address an important conflict of interest faced by numerical simulations on modern architectures: while many algorithms strive to reduce the number of unknowns and required operations per accuracy via adaptivity in space and time, the latest computing architectures ask for regular data access patterns. Our objective is to team up the advantages of adaptive, octree-type meshes with regularly refined patches (blocks). We propose to merge multiple small blocks into bigger though regular blocks wherever possible, while the size of the merged blocks is chosen with respect to hardware characteristics. If adaptivity criteria refine parts of the composed regular grid regions later, the big blocks can be decomposed again.

Plain shallow water equations act as test bed for our approach well-suited for hyperbolic partial differential equations (PDEs) in general. The latter are used to model a wide range of problems of great societal and technical relevance: examples include tsunamis or earthquakes on the continental scale, radiation-sensitive cooling processes in manufacturing, as well as flow in blood vessels on the cell scale. Hyperbolic PDE models are often characterised by a multitude of scales in space and time, such that accurate solutions demand for very fine meshes—at least in certain critical regions that change in time. At the same time, the respective applications often demand for a low time to solution. Simulation-based tsunami prediction systems, for example, have to yield reasonable results within minutes.

The multitude of scales of interest for hyperbolic solvers and the local behaviour in time (reflected by the use of explicit time stepping methods) imply that efficient computational meshes for these problems need to be dynamically adaptive: they should follow the characteristic features of the solution. Furthermore, local time stepping is important where individual subgrids march in time with different time step sizes determined by the varying wave propagation speed, e.g. The finer the granularity of the adaptivity both in space and time, the "better" is the algorithm—at least in terms of the required number of unknowns and arithmetic operations.

If we express solvers with fine granular, unconstrained adaptivity in stencil notation, a large variety of stencils matching all occurring local mesh refinement situations is required. An application of a series of such stencils in turn exhibits non-uniform data access. However, modern multi- and manycore systems offering large amount of hardware threads and vector facilities with increasing register width yield the best performance for algorithms with low memory footprint and high arithmetic intensity that are split into a vast number of homogeneous tasks. Hence, invariant stencils should be applied to big homogeneous data structures. This conflict of interest renders hyperbolic solvers on block-structured adaptive Cartesian grids a prototype challenge for novel and upcoming high-performance computing architectures.

In the presented work, we address block-structured adaptive Cartesian meshes for shallow water simulations. Our meshes result from a k-spacetree formalism [15, 17] with $k = 2$ yielding a quadtree (in 2D) or octree (in 3D), where regular Cartesian grids—we denote them as *SWE blocks*—are embedded into the leaves of the spacetree. Such a scheme facilitates dynamic block adaptivity. And adaptivity facilitates a low computational effort/memory footprint per accuracy ratio. In the present paper, we however study a different selling point of spacetree adaptivity. We use it to tune the stencil code performance: on the blocks, we apply state-of-the-art Riemann kernels yielding uniform vectorised stencils [1, 3, 9, 10]. The inter-block coupling is realised through bilinear conservative stencils from [14]. A similar technique has been proposed later in [4] for the same type of equations or in [6, 7, 11], e.g., for other challenges. Adaptive time stepping, dynamic adaptivity, and local time stepping follow [14] but are beyond scope here. Instead, our approach yields a methodology to select well-suited sizes of the Cartesian blocks for a given global adaptivity pattern automatically.

Each spacetree leaf induces a regular Cartesian block. If the size of these blocks is fixed, an adaptive spacetree induces a distinct adaptive Cartesian grid. If the size of these blocks can be configured, multiple spacetrees induce the same adaptive Cartesian grid—with a regular grid being a special case of an adaptive one. Big blocks facilitate aggressive vector optimisations, loop fusion, uniform memory access patterns, and straightforward shared memory parallelisation. Small blocks mirror loop tiling, which may improve cache usage [8, 18], but also facilitates fine-grid adaptive meshes and high block concurrency if blocks can be processed in parallel. The latter gains importance when the application faces hard memory constraints, if local time stepping is realised on a per-block basis, and if concurrency and load-balancing rely on atomic blocks. In practice, one has to choose a block size compromise. In the present paper, we make the block size a technical degree of freedom, i.e. we allow a different choice of the block size per spacetree leaf. At the same time, we follow [5] and identify regular subgrids consisting of multiple regular Cartesian grids of the same size on-the-fly. Given a performance model of the stencil operations on a regular mesh with respect to the total block size, we can then dynamically coarsen the spacetree and replace multiple spacetree leaves with one leaf hosting one capacious Cartesian mesh. This optimisation is hidden from the compute kernels, i.e. the user, and does not restrict the adaptivity pattern. Simple case studies reveal its potential impact on simulations, sketch how such a performance model can guide spacetree block configurations and give estimates for the efficiency improvements.

The proposed techniques fall into the class of autotuning of stencil codes for streaming-friendly, multicore, SIMD architectures where the stencil application is tailored to a given adaptive mesh that might change dynamically rather than making the mesh follow performance considerations. On the long term, we expect the tuning facility to be become particularly interesting when we can determine throughout the application run, i.e. online, whether few processors with high frequency and wide vector registers outperform massively parallel lower frequency configurations or the other way round. Switching on and off vector facilities, changing clock speeds, or adding cores then are not any longer showstoppers but transform into energy-aware tuning parameters, as long as the stencil operation schemes follow hardware changes.

The remainder is organised as follows: We first introduce our mesh formalism and then present our application's solver together with its stencils in Section 3. These two building blocks merge into a single block-based application that is capable to adapt the mesh-to-block mapping at hands of a performance model predicting the impact on the runtime (Section 4). In Section 5, we study the block configuration-performance interplay and derive which blocks should be merged or not in the spacetree. A brief outlook and summary in Section 6 close the discussion.

2. SPACETREE MESHES WITH REGULAR CARTESIAN BLOCKS

Let $(0,1) \times (0,1) \subset \mathbb{R}^2$ be the bounding box of the computational domain. We cut this domain equidistantly into k parts along each coordinate axis. This yields k^2 non-overlapping cubes of the same size. If we continue this splitting recursively while we decide per cube autonomously whether to refine or not, we end up with an adaptive Cartesian grid.

Let $c_0 := (0,1) \times (0,1)$, and make $\mathcal{C}$ the set of all cubes resulting from the construction process. $\sqsubseteq$ is the parent-child relation on $\mathcal{C}$. If $c_i, c_j \in \mathcal{C} : c_i \sqsubseteq c_j$, c_i is one of the k^2 subcubes resulting from the refinement of c_j. Each cube has either k^2 or no children at all. Cubes without children are *leaves* from the set $\mathcal{C}_L \subseteq \mathcal{C}$, and c_0 is the *root*.

$\sqsubseteq$ induces a directed tree graph on $\mathcal{C}$. As the nodes of this graph are cubes, i.e. spatial elements, this tree is a *k-spacetree* [15]. $k = 2$ gives the special case of a quadtree. The *height* of a spacetree is the length of the shortest path in the graph. For the trivial spacetree with $\mathcal{C} = \mathcal{C}_L = \{c_0\}$, we end up with height zero. All experiments of the present work are based upon the PDE framework Peano [16] and thus use $k = 3$. We hence omit the parameter k from now on and refer to that data structure variant as spacetree (Figure 1).

Volume-based discretisations of hyperbolic equations—or partial differential equations in general—such as finite volumes or finite elements directly yield stencils on any adaptive Cartesian grid induced by a spacetree formalism. While a direct spacetree-based stencil or system matrix derivation offers great flexibility with respect to the adaptivity, efficiency considerations as well as the intention to reuse existing software fragments suggest to add an additional mapping $f : \mathcal{C}_L \mapsto \mathbb{N}$ that embeds an equidistant Cartesian mesh with $f(c) \times f(c)$ cells into each spacetree leaf c. $f \equiv 1$ embeds a trivial grid of one cell into each leaf, i.e. each spacetree leaf

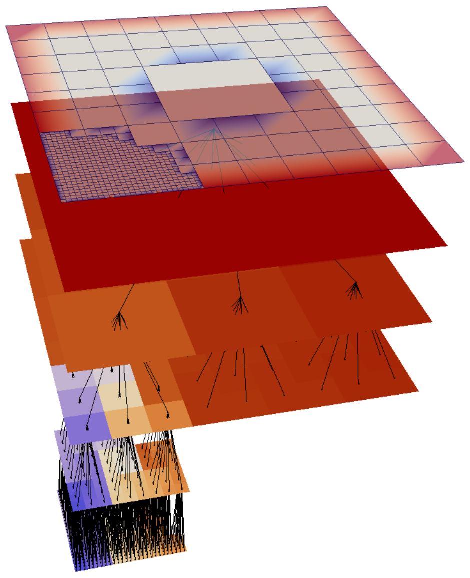

Figure 1: Adaptive Cartesian spacetree grid (top layer, transparent) with $k = 3$. The non-transparent layers below visualise the individual refinement steps, i.e. all elements of $\mathcal{C}$, with the tree relation $\sqsubseteq_{child\ of}$ as black lines.

is a cell of the computational grid Ω_h. In return,

$$f(c) = k^h \tag{1}$$

can be read as spacetree where a regular subtree of height h within the total spacetree is replaced by one spacetree node c with (1). We discuss in [5, 12, 13] how to exploit this tree replacement formalism to improve performance and introduce red-black Gauß-Seidel concurrency for direct spacetree-based PDE discretisations. From a performance point of view, it often pays off to make f return multiples of four or eight, respectively, as this fits to vector processing units—if the stencil codes exploit this fact.

Here, we start from a fixed $f(c) = n \geq 2\ \forall c \in \mathcal{C}_L$, and call the embedded regular Cartesian grids *blocks*. The spacetree then not only defines a block-structured adaptive Cartesian grid Ω_h, it also yields a non-overlapping domain decomposition of Ω_h. If we extend each $n \times n$ block by a halo layer of $\hat{n}$ cells, we obtain an overlapping domain decomposition.

Given a stencil code mapping a $(n + 2\hat{n}) \times (n + 2\hat{n})$ grid onto new values within the $n \times n$ grid and intergrid operators mapping a $n \times n$ grid onto the halo layer of another grid, we can run over the spacetree's finest level and write down any explicit time-stepping as follows:

A Run over each element of $\mathcal{C}_L$ and copy/interpolate the values of the $3^d - 1$ adjacent cells of time t onto the local halo layer. Each halo layer now holds up-to-date copies of the grid values.

B Run over each element of $\mathcal{C}_L$ and advance the values of the corresponding block from t to $t + \Delta t$.

The scheme exhibits concurrency on the block level, if we simultaneously hold simulation snapshots at t and $t + \Delta t$ per block. Then, we can update two blocks in parallel independent of each other, if they share no common vertex or face—if a neighbouring block has not advanced in time yet, simulation data of t acts as preimage of the halo initialisation, otherwise, we use the simulation data of $t + \Delta t$. For local time stepping, these two snapshots have to be interpolated anyway while the decision whether and how to advance in time also comprises wave speed considerations [14]. This scheme mirroring red-black Gauß-Seidel in linear algebra yields our *inter-block parallelisation*. It is a task-based approach with each task comprising both halo layer initialisation and unknown update.

The halo layer initialisation is pure copying of grid values into halo layers if two adjacent blocks prescribe the same grid resolution as the spacetree cubes are aligned. Their stencil is the identity. Otherwise, we realise bilinear interpolation or update fluxes according to [14] to preserve mass. Halo layer updates are cheap with respect to required floating point operations per unknown but require high memory throughput. Compared to the internal block updates, they are cheap with respect to total floating point operations as they work only on a one-dimensional submanifolds. While halo layer updates induce an overhead compared to a plain algorithm working on one regular Cartesian grid, the unknown update within the blocks dominates the overall computational workload.

3. SHALLOW WATER STENCILS

As stencil code working on the regular blocks in each leaf cell of our spacetree grid we use the SWE package [1] developed originally for teaching purposes. It processes regular Cartesian blocks of arbitrary n with $\hat{n} = 1$ halo layers. While it offers MPI parallelism and CUDA support for clusters of GPUs, we focus here on the vectorised kernels that can process a block with initialised halo data in parallel due to OpenMP. OpenMP yields our *inter-block parallelisation*.

SWE solves the basic shallow water equations given as

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t, x, y),$$

where h denotes the height of the water column (water depth), u and v encode the momentum in x- and y-direction and g is the gravitational constant ($g := 9.81\,\mathrm{m/s^2}$). The source term $S(t, x, y)$ models effects of varying ocean depth (bathymetry) or frictional or Coriolis forces. In this paper, however, we set $S(t, x, y) := 0$. Solutions are characterised by water waves (traveling at a speed of $\approx \sqrt{gh}$) triggered by initial displacements of the surface, i.e., changes in the water height h (cf. Figure 2 for some artificial settings).

SWE realises an explicit Finite Volume scheme. It leads to two computational kernels executed in each time step per block as soon as the halo layer also describing global boundary conditions is initialised:

B.1 *Computation of net updates:* For each edge, an approximate solution of the Riemann problem is computed from the quantities $Q^n_{i,j} = [h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ in the two adjacent grid cells. Following the *wave propagation* formulation [9], we compute so-called *net updates* $\mathcal{A}^{\pm}\Delta Q_{i\mp 1/2,j}$ and $\mathcal{B}^{\pm}\Delta Q_{i,j\mp 1/2}$, which determine the impact of waves entering or leaving the respective grid cells on the cell quantities.

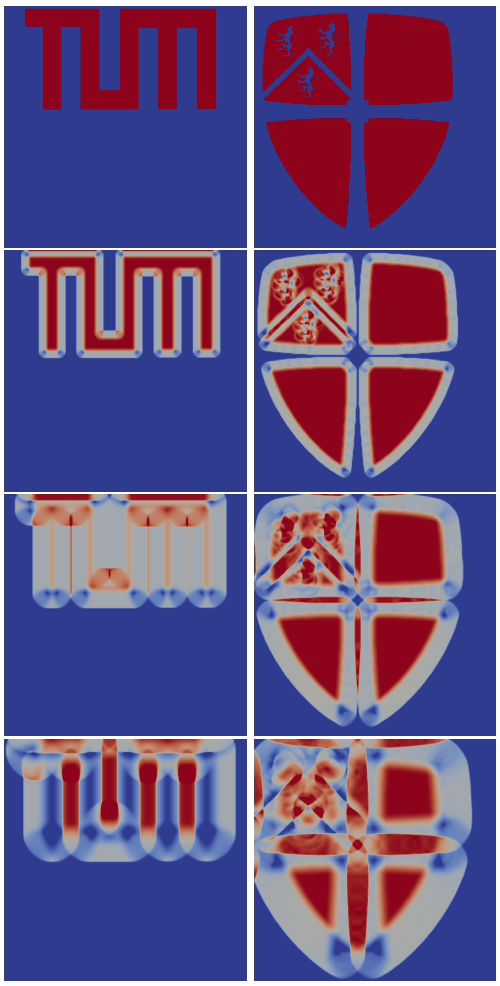

Figure 2: Two artificial water height start configurations induce waves traveling through the domain with reflecting boundary conditions. Snapshots at time $t \in \{0 \cdot 10^{-4}, 6 \cdot 10^{-4}, 1.1 \cdot 10^{-3}, 2.0 \cdot 10^{-3}\}$ resulting from a 972×972 grid.

B.2 *Updating the unknowns:* For each cell, the quantities $Q_{i,j}$ are then updated according to the balance equation

$$\begin{aligned} Q_{i,j}^{n+1} &= Q_{i,j}^{n} \\ &- \frac{\Delta t}{\Delta x}\left(\mathcal{A}^{+}\Delta Q_{i-1/2,j} + \mathcal{A}^{-}\Delta Q_{i+1/2,j}^{n}\right) \\ &- \frac{\Delta t}{\Delta y}\left(\mathcal{B}^{+}\Delta Q_{i,j-1/2} + \mathcal{B}^{-}\Delta Q_{i,j+1/2}^{n}\right), \end{aligned} \tag{2}$$

which is obtained by adding the four wave components entering the cell (i, j). The time step size Δt is restricted via the CFL condition: the maximum local wave speed (computed together with the net updates) must not exceed a fixed fraction of the mesh resolution (Δx resp. Δy) per time step.

Both, computing the net updates and updating the kernels are classical stencil-type computations, though with different characteristics. Updating the unknowns in (2) is a memory-bound loop kernel, with roughly one floating-point addition per accessed float variable as long as the steps (B.1) and (B.2) are ran one after another and not merged into one stencil. The present studies rely on a non-fused implementation. Auto-vectorisation by the compiler here can easily be achieved via a respective compiler-hint to ignore vector dependencies (`#pragma ivdep`). A similar reasoning holds for parallel for-based OpenMP parallelisation.

The loop kernel to compute the net updates runs an f-wave solver [3] on the Riemann problem. SWE provides a careful implementation of the f-wave solver that allows auto-vectorisation based on the `#pragma simd` statements introduced by the Intel compiler. Similar, it supports OpenMP concurrency. The f-wave solver requires roughly 80 floating-point operations per edge and is executed in separate loops for horizontal and vertical edges, respectively. In each loop the kernel read the quantities from two adjacent grid cells ($2 \cdot 3 = 6$ floats stored in single precision) and writes net updates for h and the normal momentum component (4 floats). Hence, the *computational intensity*, defined as the ratio of floating-point operations vs. accessed bytes of memory, of the second kernel is around two.

4. GRID TRANSITIONS

We expect the runtime per cell/stencil update to depend on the actual block size $f(c)$ whenever we update all unknowns of a block c. A naive assumption expects big blocks to be advantageous in terms of cost per unknown while small blocks allow us to tailor the grid to the solution at minimal memory cost. Given the marker

$$M(c) = \begin{cases} 0 & \text{if } c \in \mathcal{C}_L \\ n & \text{if } c \in \mathcal{C} \setminus \mathcal{C}_L \wedge \\ & \exists n: \ \forall c_i \sqsubseteq c : (M(c_i) = 0 \ \wedge f(c_i) = n)) \\ \perp & \text{else} \end{cases} \tag{3}$$

on all spacetree nodes, we know due to (1) that we can replace any node c in the spacetree with $M(c) = n > 0$ and all of its children with a new node $\hat{c} \in \mathcal{C}_L$ with $f(\hat{c}) = kn$. Such a replacement searches for a $k \times k$ arrangement of blocks of the same size, merges the corresponding spacetree nodes into their spacetree parent, and replaces the original k^2 blocks by one block. The replacement preserves the fine grid Ω_h. We hence copy the values from the original blocks, and the spacetree modification is hidden from the compute kernels. If (3) is recomputed again immediately, we may re-apply this replacement strategy.

If the start grid Ω_h is a regular Cartesian grid, such a tree replacement strategy deteriorates the spacetree after h steps with h being the height of the initial tree. On adaptive grids, it reduces the number of spacetree nodes iteratively. The interplay with dynamic adaptivity is obvious. In practice, merging always is not a good choice. Instead, it does make sense to establish a performance model $r(n)$ returning the cost per unknown for a block with $n \times n$ unknowns, and to calibrate this predictor with measurements.

The replacement of a subtree labeled with $M(c) = n$ then is advantageous if $k^2 \cdot r(n) > r(k \cdot n)$. Once a proper runtime predictor is available that takes overhead cost due to the inter-block data exchange (initialising the halo layer) into account, we end up with an autotuning approach. As each merge reduces halo layers, this autotuning also reduces the memory footprint of a given fine grid Ω_h iteratively. At the same time, each merge reduces the inter-block while it increases the intra-block concurrency.

5. RESULTS

All experiments were conducted on the Sandy Bridge and Bulldozer partitions of the CoolMAC cluster hosted at the Leibniz Supercomputing Centre. The AMD partition consists of quad-socket AMD Bulldozer Opteron 6274 nodes with 16 cores per socket, 256 GB RAM, and 2 MB exclusive L2 cache shared by two cores. They run at 2.2 GHz. The Intel partition consists of dual socket Intel Sandy Bridge-EP Xeon E5-2670 nodes with 8 real cores per socket, 128 GB RAM, and 256 KByte L2 cache per core at 2.6 GHz up to 3.3 GHz. All figures illustrate the cell updates per second for the whole simulation, i.e. include any setup or administration cost. They hence show the algorithmic throughput corresponding to the actual runtime directly.

We restrict ourselves to regular grid case studies where the block size transitions have a major impact. Statements on adaptive grids derive from the histogram of regular subtrees. The Opteron experiments are driven by the GNU compiler. Due to the pragmas, we study the vectorisation impact only on the Intel system instructed by the Intel compiler. Inter-block parallelisation is done via Intel's TBB on both systems. The intra-block parallelisation relies on OpenMP.

We first study the single core performance for grids of different size that are split into blocks of $n = 3^k \cdot 12$ (Figure 3). The Sandy Bridge system outperforms Bulldozer by up to a factor of 3.5 due to the use of single precision vector facilities and its higher frequency. For Sandy Bridge, block sizes smaller than 108 are not reasonable. Block sizes bigger than 972 also do not yield sufficient performance. The latter degradation is not observed if vectorisation is disabled. For Bulldozer, block sizes smaller than 108 also suffer from overhead. Yet, if we select bigger block sizes than 108, we observe a performance breakdown. Another breakdown is observed if we go beyond blocks of $n = 972$. The impact of Intel's turbo boost increasing the clock rate from 2.6 GHz to 3.3 GHz is not analysed further here. If interfering, it changes the results quantitatively but does not qualitatively alter the curve shape that is important to the proposed methodology.

Both systems suffer from the per-block overhead (halo layer setup and administration) for tiny block sizes. Since

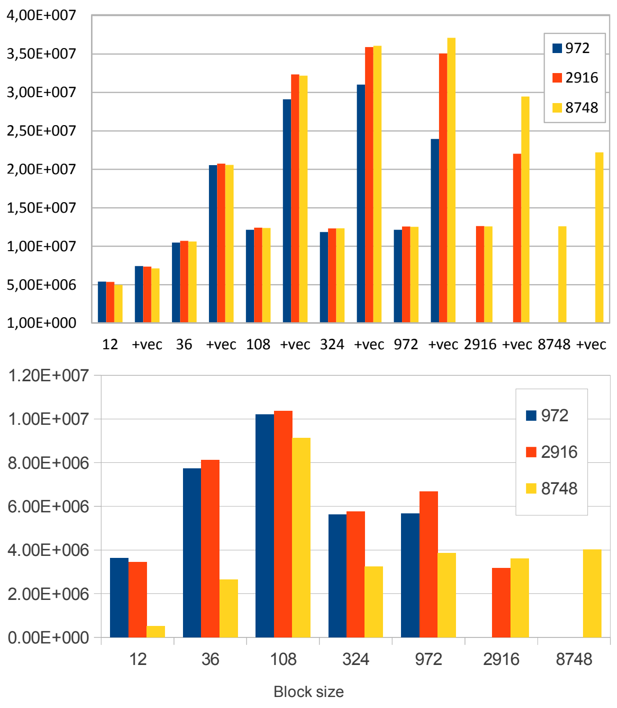

Figure 3: Throughput, i.e. cell updates per second, on a single core of Sandy Bridge (top) and Bulldozer (bottom). Each run tackles a grid of different total size split up into blocks of different size (x-axis). The Sandy Bridge measurements also compare a SIMD-loop vectorisation (+vev) to a plain implementation.

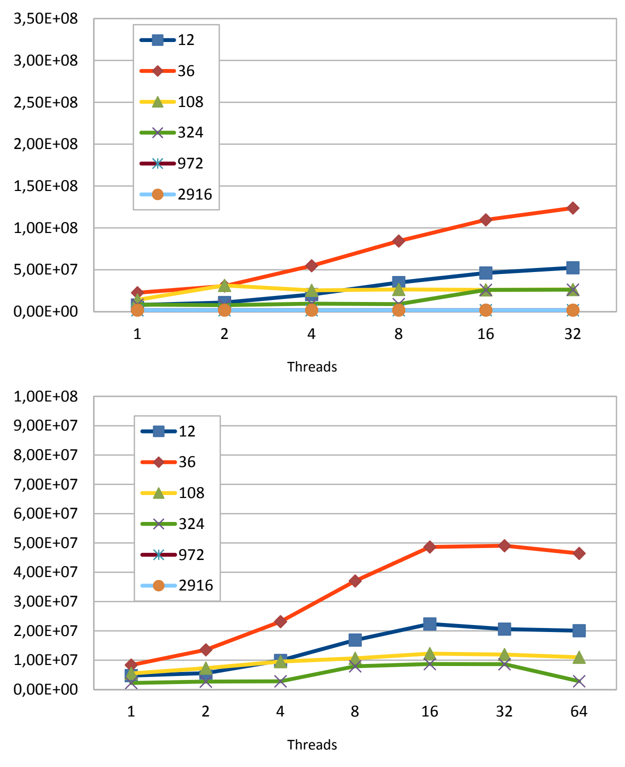

Figure 4: Inter-block parallel throughput, i.e. cell updates per second, with TBB for fixed grid of resolution 2916×2916 for Sandy Bridge (top) and Bulldozer (bottom). The grid is broken down into blocks of different size for the individual measurements, and the experiment enables different numbers of threads.

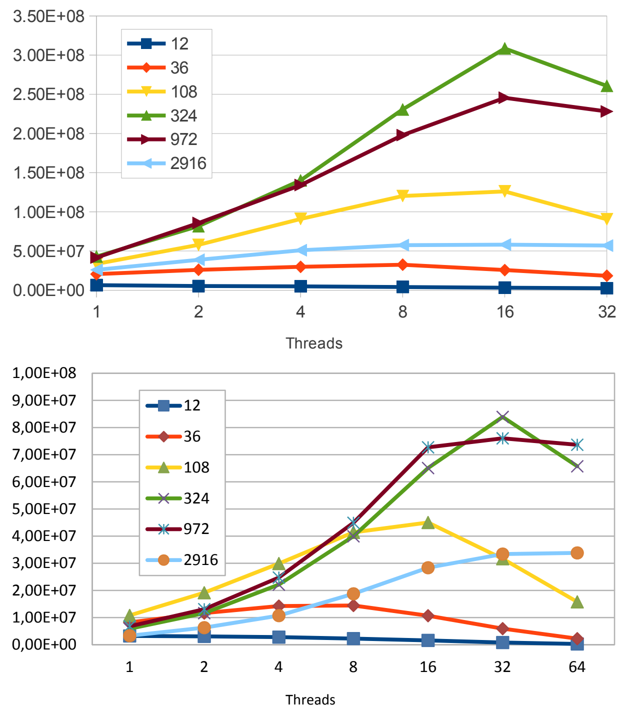

Figure 5: Intra-block parallel throughput, i.e. cell updates per second, due to OpenMP for fixed grid with 2916×2916 grid cells broken down into blocks of different size. Sandy Bridge (top) and Bulldozer (bottom). Exclusively the algorithmic phases B.1 and B.2 run in parallel due to parallel-for pragmas.

block sizes of 324 with four unknowns per cell do not fit into the L2 cache if we hold two time steps, the corresponding Bulldozer performance degradation results from the fact that the tiling cache optimisation induced implicitly by the blocking then does not avoid memory accesses anymore. Another cache threshold is hit at 972. Sandy Bridge is not that sensitive to tiling. However, its performance also degenerates for huge blocks.

If we run both codes with inter-block parallelisation where the blocks are processed in parallel (Figure 4 for $n = 2916$—other block sizes yield similar results), we observe that Sandy Bridge's hyperthreading does pay off and that both algorithms scale. While the single core performance suffers from very small block sizes, small block sizes induce a higher level of inter-block concurrency. This level in turn yields better parallel efficiency.

For the intra-block parallelisation (Figure 5), we observe a performance saturation before all cores come into play. Furthermore, the larger the blocks the better the scalability. Obviously, the inter-block parallelisation can fuse the different phases of the block updates (halo layer initialisation and the two update sweeps). However, a parallelisation exclusively of the compute loops with a serial halo layer initialisation still performs better—in particular on Intel.

Tests with a hybrid inter-/intra-block parallelisation (Figure 6) finally reveal that such a combination can not compete

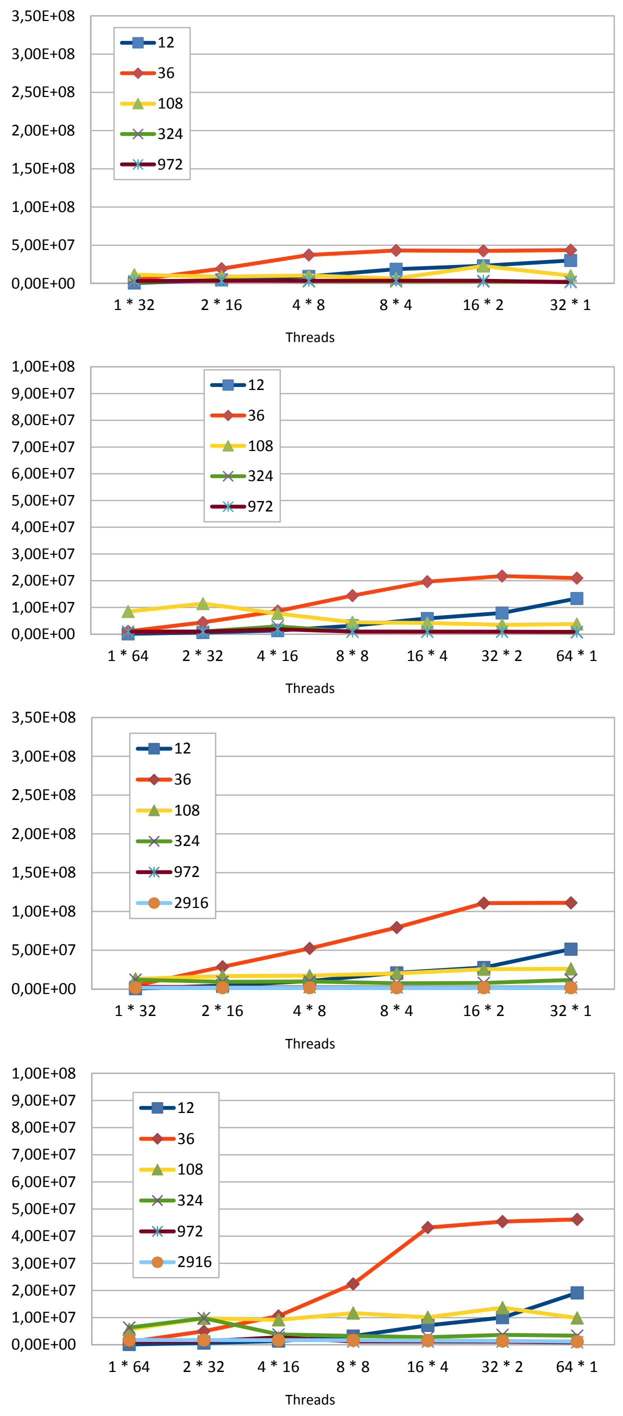

Figure 6: Throughput, i.e. cell updates per second, with intra-block and inter-block parallelisation combined. Top down: Sandy Bridge with $n = 972$, Bulldozer with $n = 972$, Sandy Bridge with $n = 2916$, and Bulldozer with $n = 2916$. Horizontally, the first number specifies the cardinality of the TBB threads whereas the number right of $*$ gives the number of enabled OpenMP threads per TBB task.

with pure inter-block concurrency as long as our grid yields sufficiently big blocks. Obviously, the stencil kernels benefit more from uniform streaming data access and a continuous filling of the compute facilities than from different algorithmic phases running concurrently. Only if the grid blocks have to remain very small (36, e.g.), the hybrid version outperforms a parallel-for variant. These experiments do not realise explicit OpenMP-to-TBB pinning, thus might be too pessimistic on some architectures for future TBB versions offering explicit pinning.

As a summary, an on-the-fly block configuration predictor should, in the present case, select a block size of around 36 at startup—this allows a reasonable per unknown performance, fine granular adaptivity, and an advantageous unknown-increase-to-runtime-reduction ratio: Whereas $n \in \{108, 324\}$ yields even higher throughput than 36, refining the grid once already yields $(k = 3)^2 = 9$ times more grid cells whereas an initial choice of $n = 36$ facilitates a finer control of the adaptivity pattern. A single core optimisation on Sandy Bridge then coarsens the spacetree as aggressively as possible as long as no block exceeds a size of 972. A Bulldozer equivalent stops already at a block size of 108. If multiple cores are available, the maximum block size should be around 324.

6. SUMMARY AND OUTLOOK

The present paper introduces a mechanism that tailors the block size of a block-structured adaptive mesh refinement solver for hyperbolic differential equations to the architecture. It starts from a given grid, preserves the fine grid, reorganises the underlying spacetree, and exploits the fact that spacetrees simplify dynamic remeshing. Basically, the approach can be rewritten as sophisticated autotuning approach for loop tiling on adaptive Cartesian grids.

While the block formalism introduces a task concurrency, using the inter-block parallelisation rarely pays off. Instead, it is more beneficial to focus on a shared memory parallelisation of the compute-intensive stencil kernels. This changes if we apply solvers with a significantly higher Flops-per-record rate—they tend to underbook the memory bus and benefit if other blocks stream in data in parallel—solvers with branches and realisations that do not exploit vector registers, or architectures with a weaker bandwidth per core. The latter also tend to undersubscribe the memory subsystem tailored to streaming applications. In the present paper, inter-block parallelism pays off if and only if the meshing enforces very small blocks. Its interplay with hard memory constraints and local time stepping one of the future challenges to study.

We want to highlight that our methodology fits perfectly to dynamically adaptive meshes, as the block structure follows the given mesh. Furthermore, it is able to react to changes in the environment. If additional cores become available, vector capabilities increase or reduce, or suddenly machine subparts with hard memory constraints shall be used in addition to/as replacement of other nodes, it is able to adapt the mesh organisation and enable the algorithm to invade such environments.

Next steps comprise the study of non-standard hardware, studies on the bigger scale, as well as shared memory and distributed memory parallelisation in combination. Of special interest however is the impact of the present ideas on different kernels with other compute characteristics.

Acknowledgements

This work partially is based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). The support of the Leibniz Supercomputing Centre under award pr63no is highly appreciated. All software is freely available at [2, 16].

7. REFERENCES

[1] M. Bader and A. Breuer. Teaching parallel programming models on a shallow-water code. In *ISPDC 2012 – 11th International Symposium on Parallel and Distributed Computing*, pages 301–308. IEEE Computer Society, 2012.

[2] M. Bader, A. Breuer, and S. Rettenberger. SWE—the Shallow Water Equations teaching code, 2013. https://github.com/TUM-I5/SWE.

[3] D.S. Bale, R.J. LeVeque, S. Mitran, and J.A. Rossmanith. A wave propagation method for conservation laws and balance laws with spatially varying flux functions. *SIAM Journal on Scientific Computing*, 24(3):955–978, 2003.

[4] K. Mandli C. Burstedde, D. Calhoun and A. R. Terrel. Forestclaw: Hybrid forest-of-octrees amr for hyperbolic conservation laws. Technical report, Universität Bonn, 2013. Preprint.

[5] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *Lecture Notes in Computer Science*, pages 567–575. Springer-Verlag, 2010.

[6] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

[7] J. Frisch, R.-P. Mundani, and E. Rank. Adaptive distributed data structure management for parallel CFD applications. In *Proc. of the 15th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013. accepted.

[8] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. F. Sibeyn, editors, *Algorithms for Memory Hierarchies 2002*, pages 213–232. Springer-Verlag, 2003.

[9] R. J. LeVeque. Wave propagation algorithms for multidimensional hyperbolic systems. *Journal of Computational Physics*, 131(2):327–353, 1997.

[10] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[11] P. Neumann. *Hybrid Multiscale Simulation Approaches For Micro- and Nanoflows*. Verlag Dr. Hut, München, 2013.

[12] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 484–496, Berlin Heidelberg, 2013. Springer-Verlag.

[13] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Sfc-based communication metadata encoding for adaptive mesh. In Michael Bader, editor, *Proceedings of the International Conference on Parallel Computing (ParCo)*, October 2013. accepted.

[14] K. Unterweger, T. Weinzierl, D. Ketcheson, and A. Ahmadia. Peanoclaw - a functionally-decomposed approach to adaptive mesh refinement with local time stepping for hyperbolic conservation law solvers. Technical report, Institut für Informatik, Technische Universität München, June 2013.

[15] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Verlag Dr. Hut, 2009.

[16] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2012. www.peano-framework.org.

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

[18] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 1–31. ACM Press, 1999.