An experience report on (auto-)tuning of mesh-based PDE solvers on shared memory systems

Dominic E. Charrier and Tobias Weinzierl *

School of Engineering and Computing Sciences
Durham University
Great Britain
{dominic.e.charrier,tobias.weinzierl}@durham.ac.uk

Abstract. With the advent of manycore systems, shared memory parallelisation has gained importance in high performance computing. Once a code is decomposed into tasks or parallel regions, it becomes crucial to identify reasonable grain sizes, i.e. minimum problem sizes per task that make the algorithm expose a high concurrency at low overhead. Many papers do not detail what reasonable task sizes are, and consider their findings craftsmanship not worth discussion. We have implemented an autotuning algorithm, a machine learning approach, for a project developing a hyperbolic equation system solver. Autotuning here is important as the grid and task workload are multifaceted and change frequently during runtime. In this paper, we summarise our lessons learned. We infer tweaks and idioms for general autotuning algorithms and we clarify that such a approach does not free users completely from grain size awareness.

Keywords: autotuning, shared memory, grain size, machine learning

1 Introduction

Whenever a code is decomposed into parallel regions or tasks, the number of tasks determines the concurrency level and hence the code's potential to scale. It is common knowledge, however, that tasks must be reasonably computationally intense. Otherwise, the system spends precious time in administering the concurrency [5, p. 197]. Thus, modern parallelisation paradigms allow users to prescribe a *grain size*, a minimal subproblem size for parallel loops, while task-based approaches group logical tasks into one physical task if separate tasks were

* This work received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). It made use of the facilities of the Hamilton HPC Service of Durham University. The authors furthermore gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (www.lrz.de). All software is freely available from www.exahype.eu.

too lightweight. For plain bulk synchronous processing and non-nested tasks, finding grain sizes is often done manually via trial-and-error since developers assume that the size vs. performance curve is convex ([8, p. 37] or [6]).

Today, nested parallel loops perform efficiently—older OpenMP versions sometimes fail to deliver performance here—but yield a high-dimensional grain size optimisation problem. With the advent of manycores and hierarchical parallelisation, manual search becomes inappropriate. Sophisticated coherence protocols, performance fluctuations, and cache effects invalidate the convexity assumption to some degree. Task formalisms with inhomogeneous execution patterns gain importance. Machine learning (autotuning) which determines both the cost function and well-suited grain sizes becomes necessary.

This manuscript discusses an autotuning approach that yields reasonable grain sizes in the ExaHyPE project [2], which combines dynamically adaptive Cartesian grids [11] with ADER-DG plus local limiting [3]. Support of interacting solvers with varying polynomial order (arithmetic intensity), inhomogeneous memory access characteristics and hierarchical hardware [10] render the use of autotuning mandatory. Our goal is two-fold: To present the algorithmic concept and rationale, and to document experiences on how this algorithm is made efficient and used efficiently. Our hypothesis is that autotuning never is a pure black box but that users have to have empirical knowledge to allow autotuning to integrate into software projects successfully and perform economically. Naïve coding of autotuning software is often ill-suited for HPC. Both goals interact.

We briefly sketch ADER-DG [3] in Section 2. Its task formulation is straightforward. However, the tasks differ significantly in arithmetic intensity, and some may have largely varying runtime. We then present our autotuning concept (Section 3). It tackles the grain size integer optimisation problem [7] parameterised by real-time measurements via randomised directional search. Emphasis is put on implementation pitfalls, e.g. the identification of valid real-time measurements. In Section 4, we discuss the algorithm's impact on the simulation workflow, before we present numerical results and close the discussion.

2 Use Case: An ADER-DG Solver

In the underlying ExaHyPE project, we solve hyperbolic PDEs

$$\frac{\partial Q}{\partial t} + \boldsymbol{\nabla}\cdot\mathbf{F}(Q) = 0 \qquad \text{on } \Omega \subset \mathbb{R}^{\mathrm{d}},\ \mathrm{d} = 2,3 \tag{1}$$

subject to appropriate initial and boundary conditions. Q is the solution, $\mathbf{F}$ the conservative flux, d is the space dimension, $\boldsymbol{\nabla}\cdot(\cdot)$ denotes the tensor divergence, while $\boldsymbol{\nabla}(\cdot)$ is the vector gradient. We solve (1) on a dynamically adaptive Cartesian grid [11] with ADER-DG [3]. In its simplest form, used here, there are three phases per time step.

Per grid cell K and time step interval $[t_a, t_b]$, we first implicitly solve

$$\int_K \int_{t_a}^{t_b} \theta_h \, \frac{\partial q_h}{\partial t} \, \mathrm{d}\boldsymbol{x}\mathrm{d}t + \int_K \int_{t_a}^{t_b} \theta_h \, \boldsymbol{\nabla} \cdot \mathbf{F}(q_h) \, \mathrm{d}\boldsymbol{x}\mathrm{d}t = 0. \tag{2}$$

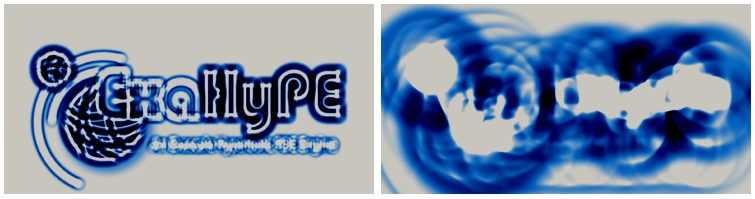

Fig. 1. Two snapshots from a $d = 2$ simulation of the Euler equations applied to an setup where the initial system energy (density) is determined by the project logo.

The space-time predictor q_h and the space-time test functions θ_h are constructed using tensor products of Lagrange polynomials over Gauss-Legendre points. Following Discontinuous Galerkin, they have compact support on each cell. Equation (2) yields a discrete fixed-point problem solved by Picard iterations [3]. All cell operations are independent of each other. The concurrent solves of (2) yield jumps along the cell faces in the solution q_h and its derivatives determining $\mathbf{F}$.

The second phase traverses all faces of the grid and computes a numerical normal flux G using q_h and $\mathbf{F}$ from both adjacent cells. We use a Rusanov Riemann solver. The solves are embarrassingly parallel with low arithmetic intensity.

In the third algorithmic phase, we traverse the cells again and solve

$$\int_K v_h \, \Delta q_h \, \mathrm{d}\boldsymbol{x} = -\int_K \int_{t_a}^{t_b} \nabla v_h : \mathbf{F}(q_h) \, \mathrm{d}\boldsymbol{x}\mathrm{d}t + \int_{\partial K} \int_{t_a}^{t_b} v_h \, G \, \mathrm{d}s\mathrm{d}t \qquad (3)$$

for $\Delta q_h = q_h(t_b) - q_h(t_a)$. The time step (3) is derived from spatially testing and partially integrating (1). It can be easily inverted given that the ansatz and test space yield a diagonal mass matrix, is evaluated per cell, and, hence, parallel.

ADER-DG describes three types of parallel tasks corresponding to phases. One is computationally heavy while two are lightweight. In our implementation, we either fuse the three task types within one grid sweep through a task formalism—one task then comprises a triad of predictor, Riemann solve and time step—or run through the grid three times and launch them through parallel fors. The runtime of the heavy tasks can typically not been predicted due to the Picard iteration. There is no single grain size well-suited for all steps.

3 Programming an Autotuning Algorithm

Our autotuning approach picks up concepts from Intel's TBB [8]. There is a central instance, a singleton [4] which is notified by the overall algorithm regarding which algorithmic phase is to be run next. We call this instance `Oracle` [6].

Our code runs through the dynamically adaptive Cartesian grid. Whenever it enters a code section which has a multithreaded implementation using tasks or contains parallel for loops, it passes the maximum problem size N of the current subproblem, and an identifier for the current code section to the `Oracle`. The

`Oracle` then returns a `GrainSize` instance. The latter holds information on the grain size to be used and the number of logical tasks which can be grouped into one physical task. After the code exits the code section, the `GrainSize` object is destroyed again.

The `GrainSize` object can also be configured to measure the time which has elapsed since its creation. The measured time is then reported back to the `Oracle` at destruction. Proper move constructors ensure this is only done once.

3.1 Algorithmic idea

The `Oracle` manages a database which stores, per entry, a code section, the algorithmic step, and further:

N_{max} the maximum problem size associated to code section and algorithmic step.
g the grain size used for this problem; $g = N_{max}$ indicates that parallelisation of this code section does not pay off.
Δg the delta from g to the previously studied grain size with $g + \Delta g \leq N_{max}$.
S_{old} the speedup obtained with this previous grain size $g + \Delta g$.
t_1 the time per problem entity needed without parallelisation.
t_g the time per problem entity needed if grain size g is used.

If no entry for these settings exists or $N > N_{max}$, a new database entry with $(N_{max} = N, g = C \cdot N, \Delta g = N - C \cdot N, S_{old} = \infty, \ldots)$ is created. $C \in \{0.5, \frac{1}{p}\}$ for p threads are convenient choices as detailed later. The `Oracle` then determines a well-suited grain size for the calling code section: For $N > g$, the invoking code is instructed to use g as grain size. Otherwise, it runs serially.

Our algorithm realises interval halving similar to [6]: We start with relatively large g and compare the multithreaded performance to a serial setting. If the serial version is faster, we deactivate the parallelisation, i.e. we set $g = N_{max}$. Otherwise, we successively shrink g with steps Δg until the resulting runtime starts rising again. Once we observe that g decrements make the runtime rise, we fall back to the previous choice of g and continue the descending search with $\Delta g/2$.

3.2 Implementation pitfalls

Whilst our approach is realised straightforwardly and similar concepts have been proposed, we identified tiny details which decide whether it is succesful. One important detail hereby is the notion of a "valid" timing. We do normalise all timings w.r.t. time per problem item: if a `GrainSize` for a problem of size N measures that the corresponding code lasts t, it reports back a time of N/t to the `Oracle`. Working with `GrainSize` instances ensures that overlapping parallelised code regions can be handled. Yet, all timings are subject to noise and, more importantly, any timing is only a characteristic sample if the underlying work per problem item is not constant. The latter is the case for our nonlinear equation system solves. Our `Oracle` thus tracks accumulated times and the number of

measurements. The resulting average time is declared valid by an additional Boolean flag once a new measurement does not change the average by more than ϵ anymore. It is not evaluated for decision making before.

Linux system timers yield useless data if all code regions are paced simultaneously. Timer invocations come along with an overhead which quickly pollutes all timings. Our solution is to introduce a global flag that determines for which code part a timer is enabled at all. After each grid sweep, this flag is randomly set to another parallel code fragment known. This way, only one code segment at a time is surveyed.

If we start to determine t_1 first, the algorithm requires a long time to enable any parallelism at all. As all timings have to converge subject to ϵ, our simulation runs in serial for a while if the `Oracle` first determines the t_1 entries in the database. This is not acceptable in HPC. Therefore, our `Oracle` randomises the grain size selection whenever it is invoked for a code fragment for which timings should be made. For one out of N_{max}/g samples, it instructs the invoking code to run serially and to report back the serial runtime. Otherwise, g shall be used and the parallel runtime t_g is updated. With shrinking grain sizes, i.e. longer simulation runtimes, fewer serial samples are taken. The sliding t_1 updates anticipate that the serial timings of code parts change if parallel regions are embedded into each other that search for well-suited grain sizes, i.e. have not converged yet.

Proper constants C determine whether the algorithm exploits a reasonable number of cores in the first place. For $C = 1/2$ in the database entry's initialisation, the maximum initial concurrency equals two. In a multicore environment, this is not acceptable. We thus choose $C = 0.5$ for $N < 2p$, i.e. for small problems compared to the thread count p, and otherwise use $C = 1/p$.

The initial ϵ choice should take the runtime distribution into account. While we may expect runtime noise to cancel out for large data sets N and, thus, that those measurements converge quickly, it is particular important to come up with working grain sizes for large subproblems quickly as those dominate the walltime. In our code, we thus scale the initial ϵ with the total serial runtime of a source code fragment. If a code fragment requests a grain size first, we ask it to run serially and to report back the time. We then scale ϵ with this time: the longer a source code fragments runs serially the more relaxed ϵ.

No fixed ϵ works for all parts of the code. Some tasks in our application solve nonlinear equation systems. Furthermore, we have nested parallelism. While a too relaxed choice of ϵ makes the `Oracle` accept garbage measurements and terminate in suboptimal (local) grain size choices, a restrictive ϵ makes measurements for some code parts never yield valid results. We thus apply widening: After each grid sweep, we analyse whether the code fragment currently studied has been supplemented with new timings and whether those timings have switched on the valid flag for our timings. If this is not the case, we widen the admissibility constraint by 10%, i.e. multiply ϵ with 1.1. 10% is a shot from the hip.

No fixed ϵ works all the simulation through. We work with large initial ϵ to come up with reasonable grain sizes choices quickly. We thus must accept inaccurate measurements at startup. Furthermore, runtime statistics do vary significantly as long as the grain sizes of embedded, nested parallel sections do vary. We thus half ϵ each time we have found a better grain size g or roll back to the previous grain size. Our `Oracle` increases the reliability of all data successively.

Track good grain sizes per problem size. We have to assume that a good grain size g depends not only on the algorithmic context but also on the problem size N. Our approach so far is N-agnostic. While a linear dependency on N might exist in some cases, we do not assume such a global relation here. Instead, our approach uses binning. We do search for good grain sizes for $N_{max} = 2$. If the code requests a grain size for $N > N_{max}$, we recursively add new database entries for $2N_{max}$. Per `Oracle` request, the database entry i is chosen for which $N_{max}(i-1) < N \leq N_{max}(i)$.

Restart measurements. After each grid sweep, we examine all database entries subject of search. If we observe that new measurements would have been made but all grain sizes belonging to the code fragment of interest are fixed, i.e. all database entries evaluated hold $\Delta g = 0$, we restart the search for these entries in one out of ten cases. This avoids that we stick to local minima always.

4 Using and Integrating Autotuning

Though we use the autotuning as black box, we found that the user has to remain aware of their integration into the simulation workflow:

Context-aware autotuning is mandatory. We found our code to react sensitively to machine type, core count, and input data sets. Some data sets may perform poorly with autotuning settings derived for other data sets. This is likely an effect of the nonlinear subalgorithms, but certainly holds for many applications. It is thus important to work with independent autotuning searches per problem setup rather than one holistic database.

Autotuning for large data sets is problematic in large-scale compute environments. Autotuning temporarily runs into inefficient parameter choices (if the grain size becomes too small, e.g.), while large single node parameter studies for the many required parameter settings might be deemed unsuitable for supercomputers or not practical. At the same time, it is important to obtain autotuning configurations on the actual target machine that later shall host a large-scale run. We thus augment our binning. Whenever the database can not host an N, a new entry for a new N_{max} copies over all setting from the next smaller N_{max}, scales them, and continues to work with those parameters. Further, if a valid parameter configuration is found for some N_{max}, our approach extrapolates this to all database entries with larger N_{max} and then makes those restart their search. This allows us to run small-scale, yet characteristic runs briefly and to automatically extrapolate reasonable grain size to large production runs.

Accuracy improves over time, i.e. the more samples the more reliable the measurement data. It is thus a natural choice to dump and reload autotuning properties. It further is very reasonable to archive them alongside the simulation data. Simulation re-runs then do not start autotuning searches from scratch but reuse performance knowledge.

We "sacrifice" only one node in a parallel environment. Autotuning introduces overhead, it has to be used carefully in large-scale simulations where all overheads have to be multiplied with the number of nodes used. We thus disable the autotuning's search on all MPI ranks besides one. All others read in the autotuning properties from a file and stick to those. The one rank tracking runtimes dumps all insight into a property file at the end of the simulation from where this knowledge becomes available to all other ranks in the next simulation. More sophisticated techniques may pass the responsibility for measurements from one rank to another throughout the simulation and propagate knowledge on-the-fly.

5 Computational Evidence

We start our computational exercises with the performance model

$$t_g = (1-\hat{f}) \cdot \frac{t_1}{min\left(\left\lfloor \frac{N}{g} \right\rfloor, p\right)} + \hat{f} \cdot t_1 + h \cdot \left\lceil \frac{N}{g} \right\rceil \text{ with } \hat{f} = f + \frac{N \bmod g}{N}(1-f)$$

which extends Amdahl's law [1] by a task administration overhead h scaling linearly with the number of tasks. $f \in [0,1]$ is the code fraction not benefiting from multithreading at all. It enters the model through $\hat{f}$ which anticipates that problems might not be decomposed exactly.

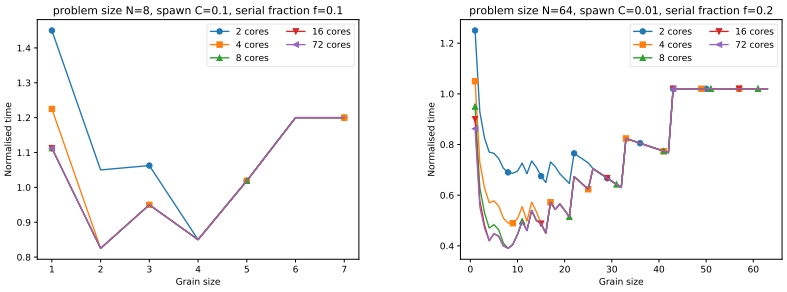

Fig. 2. Normalised time t_g/t_1 according to our performance model for $N_{max} = 8, f = 0.1, C = 10^{-1}$ (left) and $N_{max} = 64, f = 0.2, C = 10^{-2}$ (right).

Our simplistic model relying on invariant t_1 illustrates (Fig. 2) that one has to be careful not to choose the grain size too small to avoid overhead, while too

large grain sizes do not yield good speedup. This is common knowledge. Different to textbooks [8] our speedups however do not develop smoothly but exhibit a non-convex step pattern. Finally, it might be reasonable not to choose a grain size for small problems that does keep all threads p busy and thus to spare cores.

The performance model motivates our decision to trigger the search for good grain sizes with half the maximum grain size for small problems and $1/p \cdot N_{max}$ for bigger problems. As the difference between two local minima becomes the smaller the smaller g, it is reasonable to start with rather inaccurate time measurements (noise for large differences can be expected not to pollute any conclusion) and to increase the accuracy successively throughout the search. From our model, we derive that good autotuning searches for a different grain size per core number and problem size: it is reasonable to apply the binning.

Our runtime experiments were run on SuperMUC hosting Haswell Xeon E5-2697 v3 processors with 28 cores and 2.6 GHz base clock. All shared memory tests rely on Intel's TBB [8]. We studied five grain size selection strategies:

`serial` runs provide the measurement baseline and normalise all runtimes.

`dummy` is a choice of grain sizes per code part that does not anticipate the algorithmic context. We manually tuned it to yield good performance in many iterations.

`autotuning-with-finest-grid` runs the autotuning strategy.

`autotuning-from-coarse-grid` runs a cascade of autotuning experiments: it starts with a very coarse mesh, runs the autotuning, dumps the grain sizes identified, and then continues with the next finer mesh. We report only on the final run where the finest mesh sizes matches the other setups.

`autotuning-from-coarse-grid-without-learning` takes the final dump of the cascading autotuning and reruns the test again but switches off the learning, i.e. no time measurements are done and grain sizes remain invariant.

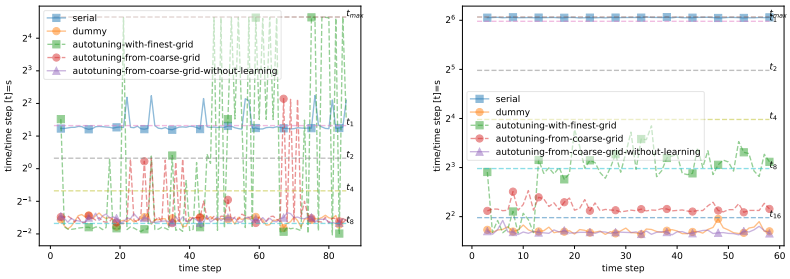

Fig. 3. Cost per time step for $d = 2$ Euler simulations where all three algorithmic steps are fused and we use polynomial order $p = 3$ (left) against a code where the three algorithmic phases are ran after each other with $p = 9$ (right).

Comparing cascading autotuning with the experiment switching off all measurements (Fig. 3) reveals that there is a significant overhead to do real-time measurements, and that there is a price to pay for the sliding updates of t_1. Once this overhead is removed, our autotuning can cope with a manual (and laborious) grain size selection. It thus makes sense to turn off autotuning wherever possible, notably on most MPI ranks.

Autotuning starting on the green field for a large problem does yield some valid grain sizes but the search process suffers from runtime spikes. The spikes result from unfortunate grain size choices that the autotuning tries and then discards. If we start autotuning on a coarse grid and then successively extrapolate the grain sizes to finer grids, we can remove the majority of these peaks.

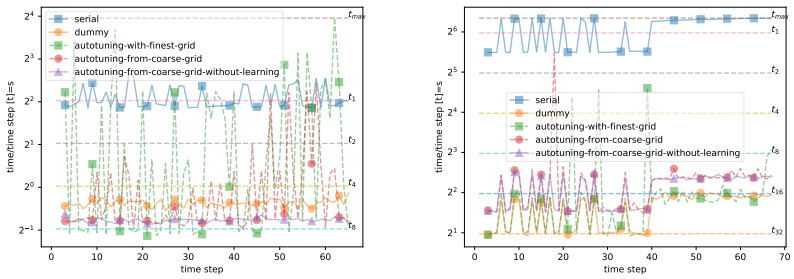

Fig. 4. Cost per time step for a $d = 2$ simulation of a shock where the ADER-DG solution is augmented with a Finite Volume limiter. $p = 3$ (left) vs. $p = 9$ (right) while all phases are fused into one grid sweep.

If we run the three ADER-DG phases consecutively, our autotuning requires longer to identify grain sizes able to compete with a manual optimisation (more than 60 time steps). It particularly struggles for the two arithmetically cheap phases. It is thus advantageous to try to fuse algorithmic phases—which can be read as a task fusion—to end up with computationally heavy individual steps.

We observe that our initial choice of $C \in \{0.5, 1/p\}$ ($C = 1/p$ is the OpenMP default for static partitioning) is reasonable. Already in the first iteration where the autotuning is unaware of N_{max}, we exploit the multicore architecture.

Once we switch from ADER-DG to limited ADER-DG (Fig. 4), autotuning becomes particularly important. Here, an additional Finite Volume scheme is interwoven into ADER-DG, eliminating numerical oscillations. As a consequence, the runtimes per cell start to vary greatly and it is hard to find globally valid good grain sizes. Our extrapolating approach is no longer robust and requires appropriate restart mechanisms.

The `Oracle`'s internal decisions are not visible from the plots. It first tries to remove parallelism from the code where parallel overhead increases the wall-time. Only afterwards, it starts to tune the grain sizes for the scaling regions.

Non-scaling features may significantly perturb the timings of the scaling regions and, thus, the `Oracle`'s decision making.

6 Conclusion

We describe an autotuning algorithm and summarise realisation decisions which made, throughout the development, the difference of whether the autotuning succeeds or not. Though the common perception of a convex runtime curve may be oversimplified, our autotuning yields proper grain size choices.

Our autotuning approach assumes codes which are completely decomposed into tasks and use parallel for loops wherever possible. Our algorithm first switches off parallelism where it does not pay off. Only then, it starts searching for optimal grain sizes for the remaining code sections. Such an approach, assuming omnipresent parallelism, seems to be a reasonable pattern for future code development. In terms of implementation difficulty, we regard it to be favourable to successive automated induction of concurrency.

An interesting next step is to augment the grain size optimisation with an additional constraint w.r.t. employed cores. We see that we can, at little loss of efficiency, for many setups reduce the number of used cores. For codes deploying multiple MPI ranks per node, other ranks then can grab these freed cores [9].

References

1. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing. In: AFIPS Proc. of the SJCC. vol. 31, pp. 483–485 (1967)
2. Bader, M., Dumbser, M., Gabriel, A., Igel, H., Rezzolla, L., Weinzierl, T.: ExaHyPE—An Exascale Hyperbolic PDE Engine (2017), http://www.exahype.org
3. Dumbser, M., Zanotti, O., Loubre, R., Diot, S.: A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. Journal of Computational Physics 278, 47–75 (2014)
4. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley Longman, 1st edn. (1994)
5. Gerber, R.: The Software Optimization Cookbook—High-performance Recipes for the Intel Architecture. Intel Press (2002)
6. Nogina, S., Unterweger, K., Weinzierl, T.: Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) Lecture Notes in Computer Science 7203: PPAM 2011. pp. 671–680 (2012)
7. Papadimitriou, C., Steiglitz, K. (eds.): Combinatorial Optimization: Algorithms and Complexity. Dover Pubn Inc (2000)
8. Reinders, J.: Intel Threading Building Blocks. O'Reilly (2007)
9. Schreiber, M., Riesinger, C., Neckel, T., Bungartz, H.J., Breuer, A.: Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization. International Journal of Parallel Programming 43(6), 1004–1027 (2015)
10. Wahib, M., Maruyama, N., Aoki, T.: Daino: a high-level framework for parallel and efficient AMR on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press (2016)

11. Weinzierl, T., Mehl, M.: Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. SIAM J. Sci. Comput. 33(5), 2732–2760 (2011)