Technical Report for ARCHER2-eCSE04-2

Upscaling ExaHyPE – on each and every core

Authors:
Baojiu Li, Institute for Computational Cosmology, Durham University
Holger Schulz, Department of Computer Science, Durham University
Adam Tuft, Department of Computer Science, Durham University
Tobias Weinzierl, Department of Computer Science, Durham University
Han Zhang, Institute for Computational Cosmology, Durham University

Abstract. *We study an MPI+multithreaded PDE solver for hyperbolic partial differential equations. Each thread per rank handles a subdomain of the computational domain identified by a segment of a space-filling curve. The threads spawn additional tasks which should be used to compensate for ill-balancing between the threads running in fork-join mode. Our studies show that this tasks-over-BSP paradigm is not properly supported in some OpenMP runtimes, leads to NUMA pollution and is vulnerable to tiny tasks. It also suffers from many memory movements. Once we replace user data with smart pointers and hence avoid unnecessary copying, we propose to add a NUMA-aware queuing system on top of OpenMP, to batch multiple tasks into meta tasks which can spread out over idle cores. Many of these techniques are fixes to current OpenMP runtime implementations and we expect them to become unnecessary as the OpenMP runtimes evolve. The insights thus have pathfinding character.*

Keywords: Task-based programming, NUMA, task scheduling, OpenMP

Introduction. This work originates from observations from the ExCALIBUR/H2020 FET HPC project ExaHyPE: ExaHyPE is an engine to solve hyperbolic equation systems given in first-order formulation over dynamically adaptive Cartesian meshes (AMR). It offers multiple numerical schemes ranging from low-order Finite Volume discretisations over Finite Differences to Runge-Kutta Discontinous Galerkin and ADER-DG schemes. A key paradigm of ExaHyPE is that users select the numerical scheme and guide the meshing, but do not "control" the actual simulation flow. Instead, they inject domain-specific knowledge via few callback routines, which basically set initial and boundary conditions, guide the AMR, and realise the terms of the partial differential equation. How the mesh is distributed, re-arranged, how computations are orchestrated and which compute kernel flavours are used is hidden from users. ExaHyPE thus can serve a wide variety of communities. In return, it has to be flexible with respect to different runtime characteristics. It is currently mainly used for astrophysical challenges and for seismic problems.

ExaHyPE is built on top of Peano 4, an AMR solver framework. Peano uses a non-overlapping domain decomposition guided by space-filling curves. The domain's cells are cut into chunks along the Peano space-filling curve and each chunk is assigned to one MPI rank. Using one MPI rank per core is infeasible on machines like ARCHER2 due to their sheer core count. Within each rank, we hence repeat the partitioning, such that we end up with a set of

chunks per rank which can be deployed to different threads. This approach works on systems with relatively low numbers of cores. Yet, on machines like ARCHER2, it quickly becomes infeasible once again: We end up with 128 or more subpartitions per rank/node, and all of these partitions have to synchronise and have to be balanced. This is challenging. Therefore, Peano implements a technique called enclave tasking [1] on top of its MPI+OpenMP parallel for (BSP) implementation: It identifies cells whose computations are not utterly time-critical and can run in any order. This excludes cells along MPI boundaries, where any solution update has to be sent out in a well-defined order, as well as cells along AMR boundaries where expensive resolution transfer operators and mesh changes potentially require a lot of expensive calculations, memory allocations, and synchronisation between various resolution levels. All the other cells, aka enclaves, yield tasks which can be deployed to idle cores. We end up with a producer-consumer pattern where some tasks per rank run through chunks of the domain, execute time-critical computations directly, champion all subdomain boundary exchange, and spawn all remaining work as separate tasks.

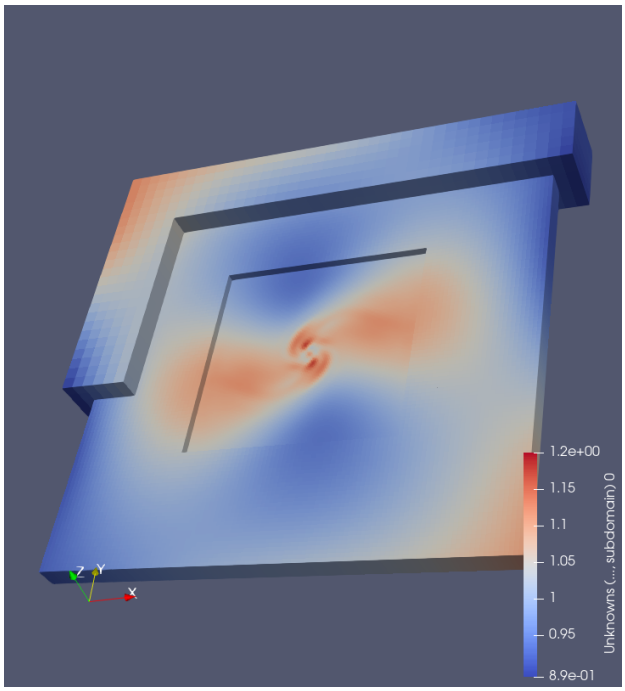
Runtime flaws in vanilla task implementation. While the “free” tasks should be used to balance out ill-balancing on the rank, our work [1] demonstrates that such a scheme suffers from three major drawbacks: (a) If threads are idle or become idle as they have finished producing further tasks, they all poll the OpenMP queues for tiny enclave tasks. This leads to massive scheduling overhead. (b) As tasks are produced on one thread and consumed by another thread, we frequently run into NUMA issues. (c) OpenMP provides no guarantee that spawned tasks enter a task queue. Instead, the runtime might decide to postpone the producing thread and to complete free tasks before. This typically happens for large task counts and implies that the producer-consumer pattern is destroyed.

However, these runtime flaws are documented for the GNU runtime, whose OpenMP runtime employs one global OpenMP task queue, whereas the LLVM runtime uses one task queue per thread. As the AMD tools as well as the NVIDIA and Intel (oneAPI) toolchain use LLVM nowadays, our project started from rerunning all experiments from [1] with LLVM. Different to the GNU runtime, LLVM performs better for massive task counts (cmp results here to data discussed in [1]). Supplementing the data decomposition with tasking pays off for all problem sizes of interest, though we still encounter a stagnation for higher core counts, where the performance gap between a sole data decomposition and a decomposition plus tasking code close. The hybrid approach does not manage to roll its runtime improvements of a factor of two over to high core counts. An introductory performance analysis conducted under the umbrella of the eCSE project highlights that the baseline Peano/ExaHyPE implementation suffers from many memory movements. These are, in line with previous publications around Peano, cache-oblivious, i.e. mainly hit the caches. Yet, they still are expensive and lead to certain erratic runtime behaviour for some core counts.

Solution architecture. Within the eCSE project, we first replaced all data copies of larger data with copies of smart pointers. The actual data resides on the heap and we only pass smart pointers between routines. Consequently, data movements are reduced though we retain all memory semantics, i.e. all data is properly freed.

Next, we proposed to add an additional, user-defined tasking layer on top of the actual OpenMP runtime: Each thread manages its own local queue into which it enqueues its tasks. This is a thread-local operation without any locks. If a thread has completed its work, it first of all completes tasks from its own local queue. Once this queue is empty, it asks companion producers to commit their completed tasks to a (logically) global queue and continues to poll this queue for vacant tasks. Such a layered approach requires us to replace the parallel for loops or taskloops for the task producers with bespoke production tasks which in turn poll their queues.

Polling of a global queue still requires many locks and can lead to congestion. We therefore refrain from committing thread-local tasks directly to OpenMP. Instead, we commit tasks into yet another user-defined queue, which is a global queue. Whenever a thread polls this queue, it scans the tasks within the queue if they are of the same type. If a poll encounters N tasks of the same type, it removes all N tasks from the queue in one rush and processes these tasks as one large meta-task. This approach mirrors the batching concept from linear algebra, where the same task (matrix) is evaluated for multiple right-hand sides. The bespoke batched compute kernels internally are again rewritten such that they fork into native OpenMP tasks such that they can occupy multiple cores.



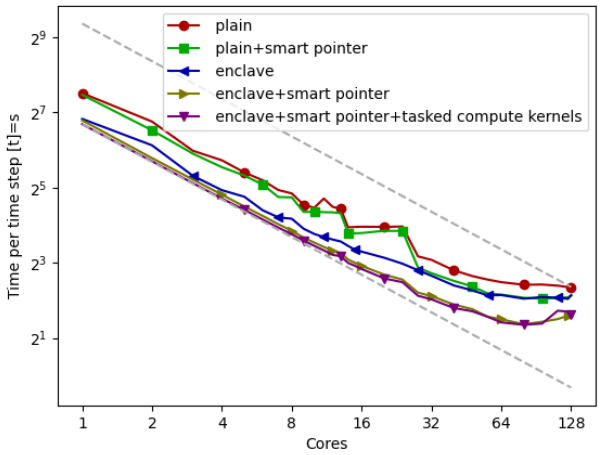
Two merging black holes. Such simulations require very efficient tasking since they employ multiple physical models over a dynamically adaptive mesh. Work by Han Zhang and Baojiu Li (ICC, Durham) exploiting eCSE research.

Performance data. We assess the impact of our work here through the CCZ4 Gauge Wave test case. This standard test in computational astrophysics employs a first-order CCZ4 formulation in ExaHyPE 2, which features 59 unknowns per degree of freedom. The same equations/setup can be used to simulate rotating black hole mergers (above). All presented data resembles data for the Euler equations [1,2,3]. For the tests, we employ a standard Finite Volume solver scheme. All data here stem from a regular grid simulation (which still

suffers from imbalances if we cannot split up the mesh into equally sized parts) and features a mesh with total=19,683 cells. We start with a setup where we host a 9x9x9 patch of Finite Volumes per cell. This is the default.

We start with single node experiments and make the code use one subdomain per thread. Each OpenMP core provided runs through a subdomain of its own. All cores synchronise "their" data after each step. This "plain" approach scales yet suffers from a step pattern for some core counts. These represent situations where the grid cannot be decomposed into partitions of the same size straightforwardly. We suffer from ill-balances as they would arise for adaptive mesh setups, too. Switching to smart pointers, i.e. avoiding memory movements, helps with higher core counts yet does not smooth out the scalability curve.

Once we add enclave tasking, we are consistently faster. However, performance is "lost" for very high core counts. In this latter case, we deploy small domain partitions per core which in turn means there are few enclave tasks. The introduction of smart pointers helps to smooth out the runtime curve. Allowing the individual compute kernels to spread out over multiple cores through additional internal tasking does not lead to a major performance gain though is robustly marginally faster. Adding plain parallel for loops to the compute kernels led to a deteriorating performance (not shown).

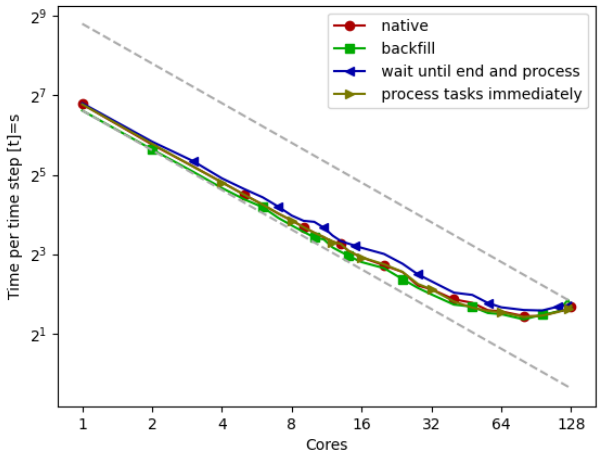


So far, all realisations have used plain OpenMP, i.e. we are not able to confirm the results with the GNU runtime [1]. With our bespoke task scheduling layers, we study various alternative scheduling schemes. We see that backfilling, i.e. the manual compensation of fork-join ill-balances, yields the best performance [2].

Evaluation and outcomes. The work completed by the eCSE project has enabled ExaHyPE to yield new insight in computational astrophysics [3]. The methodological improvements have been published and presented in leading conferences of the field [2,4], and the OpenMP "flaws" have been acknowledged and taken on board by colleagues of NVIDIA and the OpenMP Architecture Review Board.

While the introduction of additional queue layers is relatively straightforward, we acknowledge that we replicate queues which also do exist in LLVM's OpenMP runtime. This is a flaw of the current solution. A better solution would be an OpenMP interface, where the

code can explicitly control affinities and which queues to poll. Furthermore, it would be appreciated if there were an option to fuse/merge queues and to search within task queues for particular tasks of the same pattern. Finally, our approach had to rewrite the core compute kernels for batches of tasks such that they in turn yield more tasks; before the eCSE project, the kernels internally employed a plain parallel for. This is due to the fact that OpenMP tasks cannot internally use parallel fors, as a task is always tied to one core. The proposal to allow a single task to occupy multiple threads has been rejected by the OpenMP standard throughout the eCSE project (it had been on the agenda).



We end up with a code architecture which is implicitly NUMA aware, i.e. offers user-guided NUMA-aware pinning of tasks to threads, and allows the parallel execution of tasks over multiple cores/threads. However, it is not clear if OpenMP automatically leaves threads unsubscribed if employing them would lead to cache thrashing. The work has been used as a baseline to study the offloading of batches (sets) of tasks to GPUs [4].

Acknowledgements

This work was funded under the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk).

Bibliography

[1] H. Schulz, G. Brito Gadeschi, O. Rudyy, T. Weinzierl: *Task Inefficiency Patterns for a Wave Equation Solver*. In S. McIntosh-Smith, B. R. de Supinski, J. Klinkenberg: OpenMP: Enabling Massive Node-Level Parallelism, Springer, pp. 111-124 (2021)
[2] B. Li, H. Schulz, T. Weinzierl, H. Zhang: *Dynamic task fusion for a block-structured finite volume solver over a dynamically adaptive mesh with local time stepping*. ISC High Performance 2022, LNCS 13289, pp. 153-173 (2022)
[3] H. Zhang, T. Weinzierl, H. Schulz, B. Li: *Spherical accretion of collisional gas in modified gravity I: self-similar solutions and a new cosmological hydrodynamical code*. Monthly Notices of the Royal Astronomical Society, 515(2), pp. 2464-2482 (2022)
[4] M. Wille, T. Weinzierl, G. Brito Gadeschi, M. Bader: *Efficient GPU Offloading with OpenMP for a Hyperbolic Finite Volume Solver on Dynamically Adaptive Meshes*. ISC High Performance 2023, LNCS (2023) – accepted

Appendix. Here is some technical information how to re-construct the results. All the code is included in the standard Peano 4 repository.

git clone https://gitlab.lrz.de/hpcsoftware/Peano.git

While the code compiles with any recent C++17 compiler, we found that the best performance on AMD EPYCs results from Intel's oneAPI ipcx compiler. The AMD compiler wrapper around LLVM did not yield that advantageous results. The present results all stem from the following configure call:

./configure CC=icpx CXX=icpx CXXFLAGS="-Ofast --std=c++20 -mtune=native -march=native -fma -fomit-frame-pointer -fiopenmp -Wno-unknown-attributes" LDFLAGS="-fiopenmp" --with-multithreading=omp --enable-exahype --enable-loadbalancing --enable-blockstructured

After a make of the core routines, the benchmarks used here can be found in

cd benchmarks/exahype2/ccz4/gauge-wave

This benchmark is a standard astrophysics benchmark which we use to assess all performance statements. We "misuse" its convergence tests. The Python script

export PYTHONPATH=../../../../python/
python3 scalability-study.py -s fv-9

builds the whole benchmark executable.