

Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver

Dominic E Charrier¹, Benjamin Hazelwood¹,
Ekaterina Tutlyaeva², Michael Bader³, Michael Dumbser⁴,
Andrey Kudryavtsev⁵, Alexander Moskovsky²
and Tobias Weinzierl¹

The International Journal of High
Performance Computing Applications
2019, Vol. 33(5) 973–986
© The Author(s) 2019



Article reuse guidelines:

sagepub.com/journals-permissions

DOI: 10.1177/1094342019842645

journals.sagepub.com/home/hpc



Abstract

We study the performance behaviour of a seismic simulation using the ExaHyPE engine with a specific focus on memory characteristics and energy needs. ExaHyPE combines dynamically adaptive mesh refinement (AMR) with ADER-DG. It is parallelized using tasks, and it is cache efficient. AMR plus ADER-DG yields a task graph which is highly dynamic in nature and comprises both arithmetically expensive tasks and tasks which challenge the memory's latency. The expensive tasks and thus the whole code benefit from AVX vectorization, although we suffer from memory access bursts. A frequency reduction of the chip improves the code's energy-to-solution. Yet, it does not mitigate burst effects. The bursts' latency penalty becomes worse once we add Intel Optane technology, increase the core count significantly or make individual, computationally heavy tasks fall out of close caches. Thread overbooking to hide away these latency penalties becomes contra-productive with noninclusive caches as it destroys the cache and vectorization character. In cases where memory-intensive and computationally expensive tasks overlap, ExaHyPE's cache-oblivious implementation nevertheless can exploit deep, noninclusive, heterogeneous memory effectively, as main memory misses arise infrequently and slow down only few cores. We thus propose that upcoming supercomputing simulation codes with dynamic, inhomogeneous task graphs are actively supported by thread runtimes in intermixing tasks of different compute character, and we propose that future hardware actively allows codes to downclock the cores running particular task types.

Keywords

Adaptive mesh refinement, hyperbolic, Intel Optane technology, energy, cache behaviour

1. Introduction

The memory architectures in mainstream supercomputing (Intel-inspired architectures) become more and more inhomogeneous. We classify these trends into clock tick, vertical and horizontal inhomogeneity (Figure 1). Modern architectures can modify the chip frequencies of some components. Modern memory hierarchies are built in layers with the chip's registers on the top, persistent memory at the bottom and caches in-between. This yields the vertical dimension. As access from the CPU registers to the main memory is very expensive, the caches hold data copies temporarily. Small intermediate memory layers can deliver data quickly to the cores. Modern chips are predominantly multi-socket systems. This yields the horizontal dimension. Although the main memory and some intermediate memory layers are logically shared between all cores, the chip technically is split up into sets of cores with their own

memories and memory controllers. Data access cost within one layer depends on whether data reside on the local segment of memory or have to be fetched from memory technically associated with other sets of cores.

Neither vertical and horizontal nor frequency diversity is new. Their character however evolves and their impact

¹ Department of Computer Science, Durham University, Durham, UK

² RSC Group, Moscow, Russia

³ Department of Informatics, Technical University of Munich, Munich, Germany

⁴ Dipartimento di Ingegneria Civile Ambientale e Meccanica, Università degli Studi di Trento, Trento, Italy

⁵ Intel, Folsom, CA, USA

Corresponding author:

Tobias Weinzierl, Department of Computer Science, Durham University, Stockton Road, Durham DH1 3LE, UK.

Email: tobias.weinzierl@durham.ac.uk

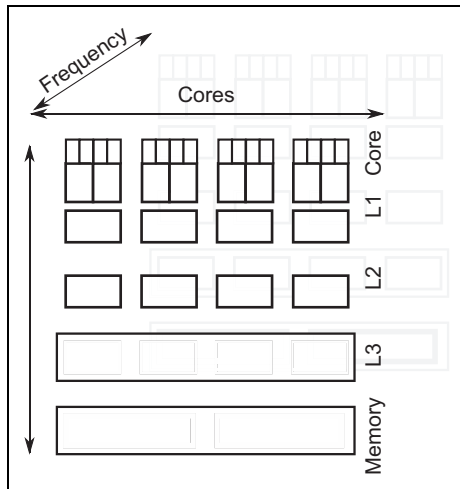


Figure 1. Vertical inhomogeneity of the data access cost, and thus speed, arises from multiple cache levels and different cache coherence strategies (inclusive vs. noninclusive). With the Intel Optane technology, main memory effectively becomes a fourth cache and an additional memory layer is added at the bottom. Horizontal inhomogeneity arises from the fact that memory is logically shared yet physically distributed. Further diversity stems from the fact that a core hosts multiple (hyper-)threads which in turn might accommodate multiple logical threads. A third diversity dimension is introduced by the opportunity to calibrate the cores' frequency.

on code performance increases. There are at least three recent hardware trends to consider: With an increase of core counts, non-uniform memory access (NUMA) effects gain importance. More cores and their caches have to be synchronized, while the pressure on the main memory increases. With the arrival of more inhomogeneous or new beyond-main memory storage technology (Intel[®] Optane[™] technology or MCDRAM) which introduce new cache layers, as well as with the farewell of inclusive caching with the Intel Xeon[®] Scalable processors (Skylake) – though likely to be compensated to some degree with the advent of mesh interconnects – we witness increased non-uniformity when it comes to memory accesses. With the opportunity to downclock or upclock system components – either triggered by users or the energy controllers on board – we finally face further fluctuations in effective speed.

These hardware features are imposed on high-performance computing (HPC) simulations by the vendors. Despite co-design efforts, *algorithmic and hardware evolution seem to diverge for some of the most advanced simulation codes*. Code developers invest significant development time into the vectorization of their core compute kernels. Downclocking hits vectorization. Code developers invest significant time into a flexible task decomposition of their codes to uncover the maximum concurrency. Yet, modern numerics yield task graphs that have heterogeneous compute characteristics, non-predictable runtime cost and dependencies changing frequently. An example are predictor–corrector schemes with expensive predictors and cheap correctors, built on top of Newton- or

Picard-iterations with dynamic termination criteria and dynamic adaptive mesh refinement (AMR). Horizontally, diverse multicore systems challenge NUMA-aware scheduling. Finally, developers invest significant time into cache blocking and compute routines of high arithmetic intensity which exploit all vector registers. New memory layers typically deliver improved bandwidth and storage size but also increase latency. For embarrassingly parallel codes streaming data through the cores as we find them in machine learning, in-memory database systems (Boyandin, 2018) or dense matrix–matrix multiplications (Kudryavtsev, 2018), latency poses a manageable challenge: threads accessing remote memory are postponed and switched with other threads. We ‘asynchronize’ threads and memory accesses. Intel’s IMDT is explicitly built with this in mind (Figure 2). Indeed, moving data into the main memory upon request can even improve the performance, as moving the data into the ‘right’ memory location eliminates NUMA penalties without complicated first-touch optimizations (Kudryavtsev, 2018). Such a programming model, being similar to CUDA, is however problematic for codes which are tailored towards cache reuse, exploit all registers and thus suffer from context switches.

Our case study on a complex AMR code with a non-homogeneous task pattern showcases flavours of this hardware–software divergence. It suggests that memory latency becomes a major showstopper. Unfortunately, (i) frequency modifications are ill-suited to tackle the latency problem – they help to improve the energy efficiency though; (ii) task/thread oversubscription is ill-suited to hide latency if data swapped out are not reliably backed up in the next-level cache; and (iii) additional memory layers amplify latency penalties. We however uncover that a heterogeneous task graph where tasks of different computational character are intermixed reduces the memory pressure and latency penalty. As a consequence, our code performs, by means of memory characteristics, better with dynamic AMR than with regular grids once the task character difference (compute- vs. memory-heavy) is reasonably high and dynamic AMR starts to mix those different tasks. This counter-intuitive result is, to the best of our knowledge, novel, and our report also seems to be the first in a line that studies the impact of the Intel Optane technology on a non-trivial solver for partial differential equations (PDEs) from both a performance and an energy consumption view.

The case study is structured as follows. We give an overview over our benchmark code base ExaHyPE, before we phrase our research hypotheses: Chip frequency allows us to balance between speed and energy efficiency, oversubscription with tasks helps us to moderate latency penalties and increased memory latency harms notably applications with an inhomogeneous compute task pattern which destroys streaming character. They circumscribe common expectations. Besides hypothesis 1, our results falsify these assumptions. The text next describes the two test machines. In the subsequent section, we benchmark the code’s runtime characteristics, before we try to find

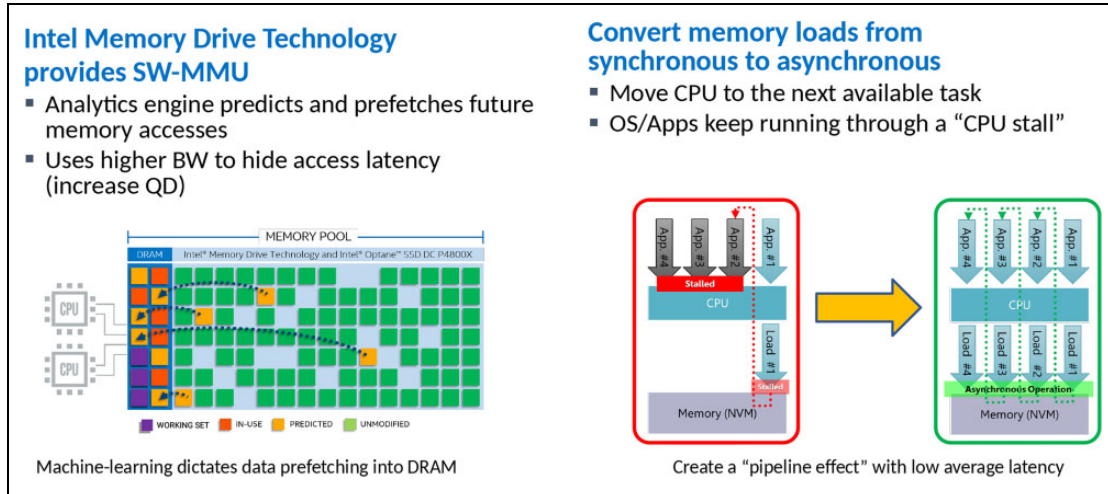


Figure 2. Intel Optane SSD DC P4800X Series with IMDT operation mode. It relies on software IP for memory management. Newer products such as Intel Optane DC Persistent Memory are promised to integrate on DIMM form-factor and also to introduce a memory mode fully managed by the CPU without extra software; and hence smaller cost penalty. IMDT: Intel Memory Drive Technology.

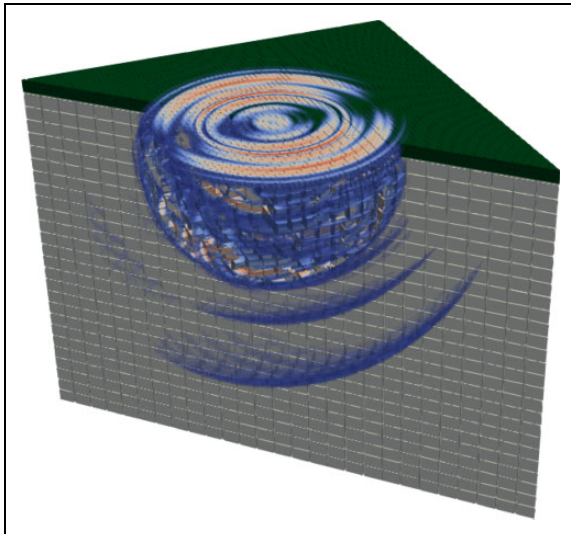


Figure 3. Cut through the LOH.I simulation. The domain consists of a homogeneous material with a thin crust on top. It is homogeneous, too, but of a different material than the remainder. A point source earthquake is inserted into the crust, that is, just below the surface. Waves propagate from this point. Regular grid visualization through the experiments runs with both regular and dynamically adaptive meshes.

evidence for our hypotheses. The findings are summarized in our conclusion and guide future work.

2. The ExaHyPE benchmark code

Our experiments study a strongly simplified and idealized earthquake scenario (Figure 3) realized through the ExaHyPE engine (Bader et al., 2014–2019). ExaHyPE solves hyperbolic differential equations in their first-order formulation with Arbitrary high-order DERivatives Discontinuous Galerkin (ADER-DG). ADER-DG is a predictor–corrector scheme (Charrier and Weinzierl, 2018; Dumbser

and Käser, 2006) which traverses a grid tessellating the computational domain and first computes per mesh cell a predicted solution evolution. This prediction is of the same order in time as the spatial order (typically $p \in \{3, 4, \dots, 9\}$) and is determined implicitly. Solving such an implicit space-time problem is computationally possible as we neglect the solution in neighbouring cells. It is a cell-local prediction. The implicit solve of high order renders the predictor computationally intense. Riemann solves in a second step tackle the arising jumps in the predicted solutions along cell faces, before a corrector step sums up the result of the prediction and the Riemann solves. These two follow-up steps are arithmetically cheap.

ExaHyPE employs a dynamically adaptive Cartesian grid. It is constructed from a spacetree (Weinzierl and Mehl, 2011; Weinzierl, 2018). The term spacetree describes a generalization of the octree/quadtree concept. Our code thus falls into the class of structured AMR or block-structured AMR where individual blocks are tiny (Dubey et al., 2016). The meshing supports dynamically adaptive grids which may change in each and every time step. On purpose, we neglect multi-node runs. They inevitably yield load balancing challenges which hide the per-node memory effects studied here. The mesh topology determines which ADER-DG tasks can be run in parallel (Charrier et al., 2018). All Riemann solves, for example, are embarrassingly parallel but require input from their two neighbouring cells. Along adaptivity boundaries, more than two cells are involved. As the adaptivity changes, every time step induces a different multicore task pattern, that is, the sequence and structure of parallel work items never is the same between any two time steps. We do not have an invariant task graph.

Combining a predictor–corrector scheme with task-based parallelism implies that (i) very compute-intensive steps take turns with tasks that are computationally cheap; (ii) the memory demands change as dynamic mesh

refinement allocates additional blocks in the main memory, while mesh coarsening releases memory segments; and (iii) the concurrency profile of the code is time-dependent and changing such that dynamic tasking with task stealing is required. To cope with these characteristics, our code base is subject to three optimizations.

2.1. Homogenization of the task execution

In ADER-DG, all cell-based (correction and prediction) tasks are independent of each other. All Riemann tasks are independent of each other, too. Between those types, dependencies exist: A predictor requires the input from the correction which in turn requires the result of $2d$ Riemann solves. Each Riemann solve requires input from the two predictions of adjacent cells.

ExaHyPE offers two task processing modes. In its basic variant, it first issues one type of tasks, processes these tasks (which are all independent of each other) and then continues with the next type of tasks. Per time step, it first spawns all predictor tasks, then all Riemann tasks and finally all corrector tasks. Dynamic adaptivity introduces additional grid modification tasks. Each sweep is homogeneous with respect to its compute profile. The total time step however exhibits inhomogeneous character. This scheme is equivalent to a breadth-first traversal of the task graph.

An alternative variant is the fused mode (Charrier and Weinzierl, 2018). It issues a task as soon as its input data are available. A Riemann solve starts as soon as the predictions of the two adjacent cells become available – it does not wait for all predictions to terminate – and a corrector task is issued immediately once all $2d$ Riemann solves on the cell's adjacent faces are solved. We issue tasks as soon as possible. This induces some overhead to find out whether a task is already ready. Yet, the latter approach allows the task runtime to orchestrate tasks of different types to run concurrently. This homogenizes, that is, averages the character of the tasks over a time step. Although we have no absolute control on the processing order of the tasks – this is up to the task runtime – we may assume that the task graph is processed close to a depth-first order (Reinders, 2007).

2.2. Temporal and spatial blocking

ADER-DG's predictor inherently realizes spatial and temporal blocking of data accesses (Kowarschik and Weiß, 2003): The expensive implicit solves are not run on the whole mesh but on a per-cell basis. This means many floating point operations are executed over a relatively small set of data. On top of this, ExaHyPE's grid traversal localizes all data accesses further. It traverses the grid along a space-filling curve whose Hölder continuity yields a spatial and temporal locality of data accesses (Weinzierl, 2018). The result of the Riemann solve feeds into the face's adjacent cells. The probability that an adjacent cell is processed shortly after is high.

Together with the optimization resulting from the homogenization, ExaHyPE realizes a cache-oblivious algorithm, which fuses correction and prediction. These cell operations are executed directly after another and merged into one task.

2.3. Optimization of task core routines

ExaHyPE customizes the engine: As soon as architecture, number of equation unknowns, PDE-term types and polynomial orders are known – as they are for our LOH.1 setup – an ExaHyPE preprocessor (toolkit) can rewrite the most time-consuming code parts into manually vectorized, tailored code kernels. For these, it employs Advanced Vector Extension (AVX) instructions, appropriate alignment and padding, as well as aggressive function inlining: The application-generic engine machinery is rewritten without virtual function calls.

3. Research hypotheses

We consider our ExaHyPE benchmark to be a prime candidate to study and assess new memory hierarchies, as the code exhibits multiple characteristic properties of modern simulation software: First, high-order, locally implicit approaches are one popular way forward to exploit vectorization. Second, we expect many future simulation codes to consist of different task types. Computationally demanding tasks – the workhorses – take turns with other, cheaper tasks which are however mandatory for advanced numerics. We focus on a predictor–corrector scheme here. Another popular example for such an algorithmic imprint is multigrid codes with expensive fine grid smoothers and cheap coarse equation systems. Third, we expect the majority of future codes to exploit some kind of dynamic adaptivity to invest compute power where it pays off most. As a result, task graphs, memory footprint and compute facility needs are never invariant or temporarily homogeneous. Notably, we assume proper a priori prefetching to become very difficult or even impossible for the runtimes. Fourth, we assume that cache blocking – realized here implicitly through a computationally heavy predictor which acts on one cell of the mesh only – removal of virtual function calls, padding, manual vectorization and so forth are state of the art for any compute kernel.

By means of our non-trivial benchmark setup, we follow up on the following hypotheses:

1. A frequency increase of the compute units helps to improve the time-of-solution, while a frequency reduction improves the energy-of-solution ratio. It also weakens the latency penalty.
2. Task oversubscription helps to hide latency effects. The popular (light) oversubscription pattern from CUDA enters mainstream processors.
3. The increased latency introduced by additional memory layers (fourth-level cache) harms notably

those runs that exhibit a strongly inhomogeneous data access pattern (dynamic AMR). In hardware, prefetching breaks down.

4. Benchmark setup and system

Our benchmark is run on the following server configurations. The first one is an Intel Xeon E5-2650V4 (Broadwell) cluster in a dual-socket configuration with 24 cores. They run at 2.4 GHz. TurboBoost can increase this up to 2.9 GHz, but a core executing AVX(2) instructions might fall back to a minimum of 1.8 GHz to stay within the Thermal Design Power (TDP) limits (Microway, 2018). Each node has access to 64 GB of 2.4 GHz TruDDR4 memory. Each Broadwell CPU utilizes a hierarchy of inclusive caches ($12 \times (32 + 32)$ KiB, 12×256 KiB and 12×2.5 MiB). While the L3 is shared and retains copies of the L1 and L2 content once data are moved into these, L1 and L2 are individual to each core. They are only shared by the two hyperthreads. The node is able to deliver around 120 GB/s in the Stream TRIAD (McCalpin, 1995) benchmark. It has a theoretical double precision peak performance between 2×105.6 (non-AVX mode and baseline speed) and 2×556.8 Gflop/s if we use FMA3 with full turbo boost (Microway, 2018). Although artificial – the all-core turbo is, for example, capped at 2.5 GHz – TRIAD and peak performance allow us to classify codes as bandwidth- or compute-bound.

The second configuration is a dual-socket Intel Xeon Scalable Gold 6150 with 18 physical cores per socket, clocked at 2.70 GHz, and equipped with 192 GB of 2.7 GHz DDR4 memory (12 ranks of 16 GB modules). The chip may reduce the base clock frequency to 2.3 GHz for AVX2 and to 1.9 GHz for AVX-512 (WikiChip, 2018). The other way round, thermal velocity boost permits individual cores to upclock up to 3.7 GHz temporarily. The node's three cache levels ($18 \times (32 + 32)$ KiB, 18×1 MiB and 18×1.375 MiB) work noninclusive: Once data are moved into L1/L2, there is no guarantee that the L3 retains a copy. Loads issued by other cores might remove them. L3 is logically shared between the cores, while L1 and L2 are individual to each core. They are only shared by the two hyperthreads. According to our benchmarks, the node delivers 125 GB/s for the Stream TRIAD benchmark. Its theoretical peak is 2×388.8 Gflop/s with base frequency in non-AVX mode. With upclocking and FMA3, the two AVX units allow us in theory to squeeze out $2 \times 2,131.2$ Gflop/s.

In our experiments, the latter system is expanded with $6 \times$ Intel DC Optane SSD P4800X, that is, 375 GB, non-volatile memory. The SSDs are connected via PCIe-switch IC to the CPU, while the Intel Memory Drive Technology (IMDT) implements software-defined memory on top of the Intel Optane technology SSDs (cf. Figure 1). This IMDT uses part of the overall memory capacity from the DRAM for caching, prefetching and endurance protection, that is, the drives become transparently available to the operating system as system memory. Although our memory

totals to roughly 1.4 TB, we stick to the default IMDT settings recommended by Intel which limits the available memory to $8 \times$ the main memory. Larger ratios than $1 : 8$ would lead to performance drops according to the vendor.

All shared memory parallelization relies on Intel's Threading Building Blocks (TBB) (Reinders, 2007) while Intel's 2018 C++ compiler translated all codes. We use Likwid (Treibig et al., 2010) to read out hardware and energy counters made available through Intel's Running Average Power Limit (RAPL). On the Purley platform chip, we use energy sensors which are directly attached to the board.

Our experiments study the LOH.1 benchmark (Day and Bradley, 2001; The SPICE Code Validation 2006) realized through the ExaHyPE engine (Bader et al., 2014–2019). LOH.1's artificial setup splits up a cubic domain into two horizontal layers of material. An earthquake is then induced as point source inside the cube. Sensors close to the domain surface track incoming waves. While LOH.1 is artificial, it exhibits real-world simulation characteristics with its material transition, a source term and non-trivial inference and reflection patterns. To obtain high-quality results at reasonable cost, a feature-based refinement criterion follows the steepest solution gradients and shocks. The mesh spreads from the point source.

5. Benchmark code characteristics

5.1. Automatic frequency alterations

We kick off our experiments with studies on the Broadwell chip. For statements on the code's scaling, it is important first to understand the frequency behaviour under load. On Broadwell, a single- or dual-core setup drives the chip at around 2.5 GHz (Figure 4). If we however use all 24 cores and run our optimized code variant using AVX, the node is downclocked to around 2.165 GHz on average. If we manually disable AVX, the downclocking is less severe (2.35 GHz on average). For all-core loads, overclocking is not used of the chip's own accord. We observe one core – predominantly being busy with task production and scheduling – to perform at close-to-nominal speed. All others clock down. Scalability graphs have to take the amortized downclocking into account.

5.2. Scalability

To assess the impact of frequency, horizontal and vertical diversity, it would make limited sense to benchmark serial or non-scaling code. Before we study our code's memory and energy characteristics, we thus validate that the code scales reasonably on Broadwell (Figure 5). Qualitatively similar results arise on the Intel Xeon Scalable chip. This holds despite the significantly changed memory architecture.

As the corrector step is merged into the predictor in the fused scheme, we benchmark the fused scheme against the nonfused implementation and decompose the latter's behaviour into the scaling of the Riemann solve and the scaling

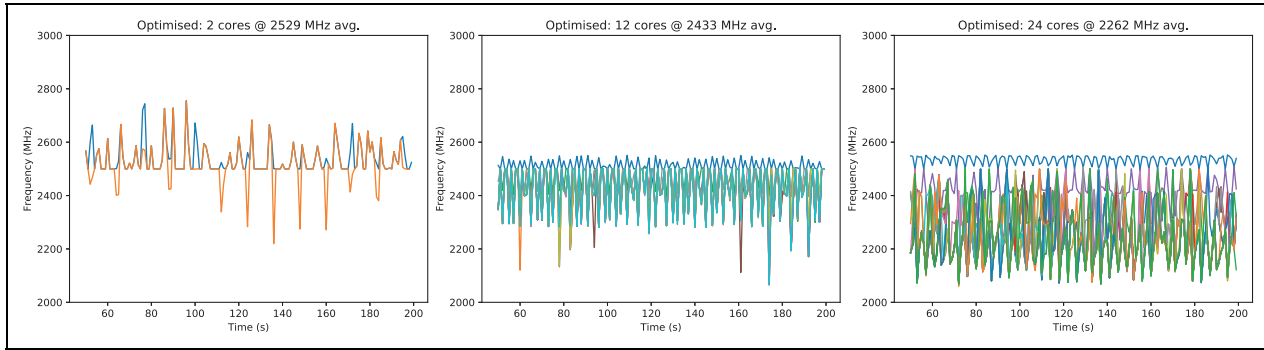


Figure 4. Broadwell's frequency choice per core for a dual-core (left), one-socket (middle) and a 24-core run (right). The caption gives the time-averaged frequency. All setups rely on code translated with AVX. Without the manual AVX optimization (not shown), the average frequency is 2616 GHz for two cores, drops to 2454 GHz on a socket and finally to 2409 GHz, while the AVX-optimized code base, as shown, yields 2529, 2433, 2262 MHz, respectively.

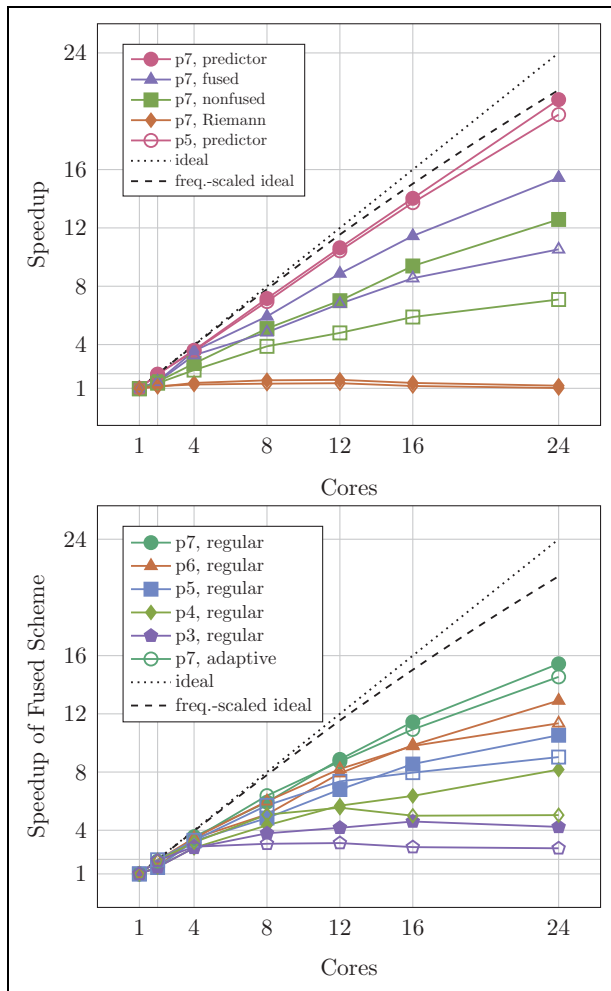


Figure 5. Code scalability on Broadwell for various orders and a $27 \times 27 \times 27$ grid. Top: Scalability of the isolated predictor and Riemann phase of the nonfused scheme plus overall scalability of the nonfused (straightforward) and the fused schemes for orders 5 and 7. Both an ideal linear speed-up and a speed-up calibrated with the observed frequency reduction are given. Bottom: Scalability of the fused scheme for orders 3–7. Regular grid runs are compared to adaptive grids where a dynamic refinement criterion is allowed to add additional grid levels to the regular base grid.

of the cell-wise operator (Figure 5) top. While the predictor scales perfectly for both $p = 5$ and $p = 7$ once we accept that the cores clock down, the Riemann solvers scale hardly at all. They are heavy on data movements, move many small chunks of data and suffer from the AMR administration overhead which we did not remove from any plot. Furthermore, we exploit the first-touch policy to ensure that data are allocated following their cell associativity: all cell data plus all faces of this cell are allocated en bloc by the allocating thread. The Riemann solves however bring together data from two cells and thus suffer from NUMA effects. In the end, the combination of the three task types ends up in-between these two extreme cases. Fusion is robustly faster than the nonfused scheme. The higher the order, the more dominating the predictor steps. The arithmetic intensity of the Riemann solves in contrast is close to p -invariant. Similar to an Amdahl law, we obtain better scaling overall when we increase the order.

Our experiments clarify that our code scales reasonably on one socket. The predictor ‘saves’ the overall scalability. If the other 12 cores are also used, the parallel efficiency deteriorates.

5.2.1. Observation 1. Strongly dynamic AMR codes with heterogeneous tasks suffer from multi-socket architectures. It might be reasonable to deploy one process/rank per socket, that is, to give up on the idea of a larger shared memory system, to reduce NUMA effects.

Higher orders mitigate this effect, while dynamic adaptivity makes it slightly worse. Dynamic adaptivity is not for free. Further increases of the mesh resolution n (growing the mesh with $\mathcal{O}(n^d)$ for regular grids) or polynomial order (increase in $\mathcal{O}(n^p)$) are impossible due to memory limits on Broadwell. Our benchmark quickly is caught in a strong scaling regime. This observation is typical for many HPC codes working with dynamically adaptive meshes. They do not exhibit arbitrary concurrency.

5.2.2. Observation 2. To improve code scalability, increases of the polynomial order or mesh size are required. Such

Table 1. Hardware counters on Broadwell (24 cores) for a 27^3 grid.^a

	Regular						Dyn. adaptive					
	Scalar			AVX2			Scalar			AVX2		
	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$
Avg. time (s)	0.34	0.84	2.29	0.31	0.52	1.3	2.57	4.09	11.11	2.58	2.88	6.79
Gflop/s	12.3	32	41.7	12.8	47.9	74.3	3.7	18.4	37.1	5.7	23	58.1
L2 request rate (%)	6.55	9.27	8.80	8.52	16.34	19.78	5.09	7.56	7.99	6.79	14.93	18.16
L2 miss ratio (%)	13.76	14.43	18.95	10.37	15.28	19.96	17.12	16.91	18.26	11.92	13.63	17.59
L3 request rate (%)	0.09	0.04	0.04	0.11	0.20	0.28	0.05	0.05	0.04	0.09	0.17	0.28
L3 miss ratio (%)	29.38	7.43	8.85	60.69	20.91	13.40	28.23	14.15	9.86	44.55	24.26	11.57
Mem. bandwidth (GB/s)	7.9	6.5	5.4	6.4	12	9.7	4.5	6.8	5.7	6.1	8.2	9.4
Avg. time (s)	0.3	0.51	1.78	0.28	0.38	0.94	1.95	2.56	9.04	1.97	2.32	4.59
Gflop/s	11.8	42	54.3	11.8	26.9	93.1	5.5	22.3	51.8	4	24.5	66.1
L2 request rate (%)	6.49	8.69	8.92	9.21	17.17	18.17	5.08	8.43	8.63	7.14	14.24	18.32
L2 miss ratio (%)	9.58	16.07	18.79	8.26	15.56	22.49	13.37	15.35	15.77	8.90	17.68	20.48
L3 request rate (%)	0.02	0.04	0.04	0.05	0.19	0.28	0.03	0.04	0.03	0.06	0.16	0.25
L3 miss ratio (%)	25.48	4.64	6.84	41.80	2.99	3.96	27.42	11.22	10.98	29.54	8.06	5.54
Mem. bandwidth (GB/s)	4.85	6.34	4.09	6.05	11.74	6.91	3.17	4.34	3.25	4.49	5.47	5.86
Avg. time (s)	0.11	0.1	0.11	0.11	0.11	0.1	0.95	0.99	1.04	0.92	0.94	0.96
Gflop/s	0.6	3	9.4	0.6	3	8.6	0.5	3.8	15.6	0.2	1.1	2.8
L2 request rate (%)	9.62	9.91	8.32	8.32	14.37	18.76	3.61	3.03	6.45	3.37	5.32	8.07
L2 miss ratio (%)	17.18	16.42	17.15	16.65	16.07	19.71	20.43	20.08	5.29	23.54	20.26	21.85
L3 request rate (%)	0.10	0.05	0.05	0.17	0.30	0.28	0.08	0.05	0.02	0.09	0.10	0.16
L3 miss ratio (%)	30.97	21.59	12.72	33.23	18.92	9.96	24.76	23.86	19.27	28.34	26.17	14.61
Mem. bandwidth (GB/s)	2	2.9	4.3	1.66	2.4	3.31	3.7	3.32	2.71	3.36	3.51	3.62
Avg. time (s)	0.11	0.14	0.18	0.14	0.14	0.18	1.13	1.55	2.1	1.1	1.27	1.72
Gflop/s	2	6.6	12.5	1.8	6.4	13	1.2	5.3	15.6	0.6	2.1	5
L2 request rate (%)	5.61	6.51	8.10	16.06	19.20	25.79	3.69	3.53	7.22	5.59	9.01	13.64
L2 miss ratio (%)	22.32	19.64	14.67	26.43	20.56	17.73	25.49	21.58	7.54	26.93	23.27	21.36
L3 request rate (%)	0.10	0.05	0.05	0.32	0.34	0.42	0.08	0.08	0.04	0.13	0.24	0.26
L3 miss ratio (%)	39.82	34.55	33.17	61.23	46.50	41.84	35.39	38.22	32.61	37.58	43.81	39.93
Mem. bandwidth (GB/s)	7.2	15.9	21.7	8.4	18	26.8	4.6	7.3	9.9	3.8	7.2	11.6

^aIn the columns, scalar denotes no vectorization (-no-vec -no-simd) and no generation of vectorized inline assembler code. The table is vertically split into four blocks. The top block presents the whole code characteristics if all three different task types are merged into each other. Three blocks follow which break down all measurements into the phases prediction, Riemann solves and solution correction.

increases however are constrained by the memory available.

It does not come as a surprise that it is desirable to have more memory to be able to increase either the resolution, that is, to shift the strong scaling regime, or p , that is, to increase the arithmetic intensity (Hutchinson et al., 2016). While this favours the introduction of novel large-scale memory as provided through the Intel Optane technology, our observations suggest that any architectural extension that increases NUMA penalties affects the overall performance negatively.

5.3. Code characteristics and optimizations

If we rewrite our code into a fused variant where a task is immediately triggered once its input data become available, we obtain faster code. It robustly pays off to issue compute tasks as soon as their input data are available and thus to overlap computationally demanding with memory-intensive tasks. This observation is in line with implicit data access blocking as we find it in Intel's TBB (Reinders, 2007),

where the task graph/tree is processed depth-first. We thus focus solely on the fused scheme from hereon.

Our benchmarking continues with performance counter measurements. Each test is done without any vectorization (disabled at compile time) and with full AVX2 vectorization. We observe a robust speed improvement through vectorization (Table 1). The reduction of the time-to-solution follows an increase of the gigaflop per second (Gflop/s). The positive vectorization impact is solely due to the high-order prediction tasks which make up for the majority of the runtime. The higher the polynomial order, the higher the fraction of the runtime plus the higher the Gflop/s.

While the predictors yield the Gflop/s, the solution update, which is fused with the predictor, delivers the memory throughput. It reaches around 25% of Stream TRIAD on Broadwell. Correcting the solution reduces the effective Gflop/s of this fused, cell-aligned task type. The vectorization success is diminished further by the fact that arithmetically intense tasks take turns with cheap Riemann solves. The runtime of the latter benefits insignificantly from vectorization. Riemann solves process face by face. Each face

Table 2. Measurements from Table 1 for the Intel Xeon Scalable gold running with all 36 cores.^a

	Regular						Dyn. adaptive					
	Scalar			AVX2			Scalar			AVX2		
	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$
Avg. time (s)	0.28	0.38	1.15	0.27	0.32	0.95	2.32	3.01	6.48	2.52	3.19	5.36
Gflop/s	14.6	62.5	86.7	22	100.9	156.4	7.3	32	74.6	8	32.8	95.3
L2 request rate (%)	4.82	5.55	7.64	9.31	15.64	21.68	5.81	5.66	8.76	11.06	18.78	22.00
L2 miss ratio (%)	8.25	4.10	11.46	9.25	7.35	12.27	12.86	4.31	9.27	11.41	3.70	12.44
L3 request rate (%)	0.03	0.02	0.03	0.11	0.06	0.16	0.06	0.01	0.03	0.15	0.08	0.17
L3 miss ratio (%)	55.04	60.71	14.22	85.55	75.49	59.23	37.52	45.68	11.69	56.32	63.04	47.92
Mem. bandwidth (GB/s)	9.7	13.7	20.4	13.5	23.5	89.7	6.6	6.7	12.1	7.4	9.7	64.2

^aWe show only data for the fused scheme without a breakdown into individual phases.

is of small memory footprint, Riemann solves are not arithmetically intense, and our AMR data structures associate face data with the cells. This scatters the Riemann input/output data in memory.

We have not been able to observe any significant impact of the task character homogenization on the AVX downclocking in our experiments. One might expect that intermixing computationally heavy with cheap tasks implies that not all cores run AVX at the same time, and the node thus does not throttle the speed as significantly. This however seems not to happen significantly. Figure 4 remains representative.

To characterize the cache usage, we measure per cache level the request and the miss rate (Table 1). Request rate means number of requests divided by number of instructions. Miss rate means number of requests not served by a cache divided by number of instructions. From both rates, we can derive the miss ratio which is the ratio of cache accesses which have not been served by a particular cache.

The request rate of both L2 and L3 increases with increasing polynomial order. Furthermore, it is significantly higher for AVX-enabled code, while the request rate decreases rapidly over the cache levels. Our code's aggressive cache blocking renders the dominating predictor cache-efficient. The predictor's data do not fit into L1, but barely any misses hit through the last-level cache (LLC).

For all programme phases and both for the vectorized code base and without AVX, our miss ratio is high: Every time a piece of data is not found in the L2, the LLC cannot serve this request either with high probability. Although the miss ratio decreases with increasing polynomial order, high ratios imply that we are neither bandwidth- nor compute-bound. This problematic behaviour stems from the Riemann solves. They pollute the caches through their low arithmetic intensity, small input data cardinality and NUMA effects and thus both are cache-inefficient themselves and pollute the following correctors. It is the re-filling of the caches with small chunks of data for the corrector/predictor steps which slows down the code. It is dominated by memory latency. The two steps themselves are memory-efficient.

The Intel Xeon Scalable chip amplifies all observed trends. The chip delivers higher performance – also due to the increased core count – and benefits from a decreased L2 miss ratio (Table 2), as the L2 cache per core is increased. However, changing from inclusive to noninclusive caches and reducing the cache-per-core size makes the code yield an even higher L3 miss ratio. This results in a significantly increased memory bandwidth.

5.3.1. Observation 3. The code suffers from memory latency.

We obtain a reasonably high percentage of peak performance for a dynamically adaptive grid through the high-order space-time predictor. The necessity to inter-wave it with cheap tasks however implies that we are overall neither compute- nor bandwidth-bound. We are latency-bound.

6. Frequency and energy analysis

Modern chips regulate their frequency actively: Notably AVX operations induce a frequency reduction if the chip exceeds its energy/temperature thresholds (Figure 4). Our results (Table 3) validate that the reduction in time-to-solution compensates for problematic impact: With AVX, the executable delivers the results faster at lower energy footprint. The higher the polynomial order, that is, the higher the arithmetic intensity of the heavy compute tasks, the stronger this effect.

We continue with experiments on our Xeon test bed and manually modify the frequency of the chip. Frequency alterations affect all standard components' maximum speed, while intense AVX usage still might reduce the frequency. Our data track the time-to-solution and the energy consumption per simulation run (Figure 6).

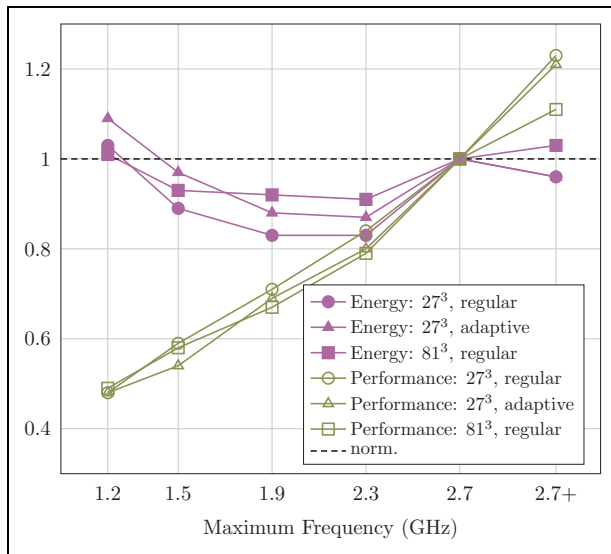
6.1. Observation 4

Running a chip at maximum frequency and high polynomial degree is best in terms of time-to-solution. If energy per simulation however is the optimality condition, a significant reduction of the frequency is advantageous.

Table 3. Energy consumption per DoF on Broadwell (24 cores) and Xeon Scalable (36 cores) for a typical run with a regular and a dynamic grid.^a

	Regular						Dyn. adaptive					
	Scalar			AVX2			Scalar			AVX2		
	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$	$p = 3$	$p = 5$	$p = 7$
Total energy (kJ)	1.46	5.24	15.54	1.29	3.02	9.01	27.57	41	96.84	22.24	28.42	53.7
Total energy per DoF (mj)	1.16	1.23	1.54	1.02	0.71	0.89	3.16	1.39	1.39	2.55	0.97	0.77
Energy DRAM (kJ)	0.49	1.25	3.06	0.43	0.78	1.79	6.96	9.03	19.12	5.42	6.62	10.64
Energy DRAM per DoF (mj)	0.389	0.294	0.304	0.341	0.183	0.178	0.799	0.307	0.274	0.622	0.225	0.153
Total energy (kJ)	3.24	6.41	17.77	2.85	4.63	13.32	36.46	46.68	123.74	51.75	46.49	91.48
Total energy per DoF (mj)	2.57	1.51	1.76	2.26	1.09	1.32	4.19	1.59	1.78	5.94	1.58	1.31
Energy DRAM (kJ)	0.17	0.25	0.69	0.17	0.25	0.87	1.83	2.04	4.95	2.72	2.42	4.9
Energy DRAM per DoF (mj)	0.135	0.059	0.068	0.135	0.059	0.086	0.210	0.069	0.071	0.312	0.082	0.070

DoF: degree of freedom.

^aThe total energy plus the energy spent on the memory are given.**Figure 6.** Filled symbols: Energy results for various CPU frequencies on the Intel Xeon Scalable machine for $p = 6$. Empty symbols: Time-to-solution results. We normalize the results against the default frequency of 2.7 GHz. The label 2.7+ denotes a base clock of 2.7 GHz with 3.7 GHz TurboBoost enabled. The memory frequency is determined by the chip (auto modus). Larger y values denote higher energy hunger or faster code, respectively.

Our results are in line with reports on the best-case efficiency for Linpack if the total energy consumption has to be minimized (Glessner, 2016). They also agree with ADER-DG experiments on tetrahedral meshes (Breuer et al., 2015). More detailed studies however uncover three more insights: (i) a frequency alteration does not change the character of our latency challenge. We observe no flattening of the speed curve when we reduce the frequency. Notably, we have not been able to observe any statistically significant impact on the cache counters and thus latency effects when we did alter the memory speed against the CPU (not shown). The memory's auto mode

choosing an appropriate memory speed is not outperformed by any manual memory frequency tuning. (ii) For our heterogeneous task graphs, turbo boost techniques which allow cores to temporarily upclock yield significantly improved performance at a limited increase of energy hunger. (iii) As cache capacity grows, as caches become noninclusive and as the core count rises, our cache-optimized algorithm also makes the chip spend more energy on the cores rather than the memory, and the total energy cost per degree of freedom (DoF) grows (Table 3).

6.2. Observation 5

Core frequency reductions – whether manually imposed or triggered through AVX – are insufficient to mitigate latency effects.

6.3. Observation 6

Larger caches and higher core counts do not automatically improve the energy efficiency.

The latter observation results, to some degree, from a saturated scalability of the benchmark code.

7. Pinning, hyperthreading and latency hiding

Many HPC codes report pinning to be essential to achieve reasonable performance and to avoid NUMA pollution. We have not been able to confirm that TBB's task pinning pays off for our code. No data are presented here, as no statistically pinning impact could be observed:

7.1. Observation 7

Runtimes with and without thread pinning can hardly be distinguished.

As our code is extremely cache efficient, data reside in the cache close to the core. If the system should decide to

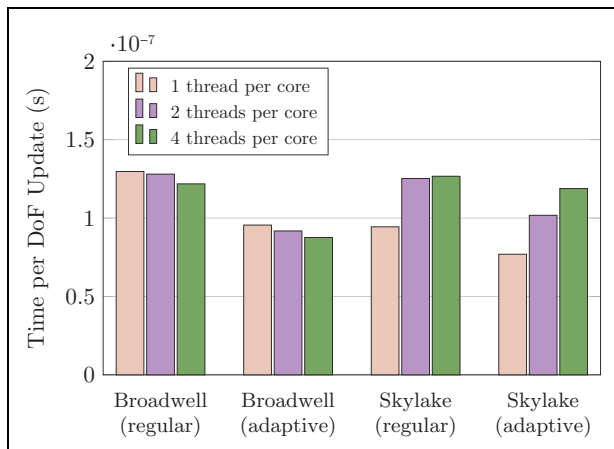


Figure 7. Characteristic $p = 7$ runs for both regular and adaptive meshes on the 36 cores of the Intel Xeon Scalable. We benchmark a one thread per core setup to oversubscribing with two (hyperthreading) or even four threads. The latter overbooks each hyperthread with two logical threads. The label Skylake identifies the Intel Xeon Scalable processor.

migrate a running task, the cache content has to be moved, too. However, a code with such extremely localized data access usually does not run into traditional cache conflicts and false sharing. In contrast, it might be reasonable to improve the affinity-awareness of the thread scheduler. This option is not explored further here.

Hyperthreading and oversubscription are a popular technique for deep memory hierarchies (Figure 2) or systems where floating point units are shared between physical threads: Whenever a data request cannot be served, the system issues a cache transfer. At the same time, the system swaps this thread with another compute thread until the data eventually have arrived. The cores thus do not idle. This is similar to the streaming/high throughput compute paradigm in CUDA. While one thread uses the floating point capabilities, other threads can fetch/prepare all data for the subsequent AVX usage and queue to continue once the first thread ‘releases’ the vector units.

On Broadwell, the techniques do yield performance improvements in our case (Figure 7). Starting from the Intel Xeon Scalable (Skylake) hardware generation however, they are counterproductive. They decrease the performance. This holds for all problem sizes.

7.2. Observation 8

Our code’s performance suffers from hyperthreading and thread oversubscription on systems with noninclusive caches. Oversubscription is *not* a way forward to mitigate latency effects here.

The result is not a surprise once we take into account that the code relies heavily on data access localization and that individual tasks with their high arithmetic intensity already fill the close caches. Within one task, the code streams data through the AVX components. Switching tasks is expensive: It interrupts the AVX usage pattern of

the current thread and induces further capacity misses on the close-by caches. If the data then still reside in a next-level cache, these runtime penalties are eventually compensated by the gain in vector facility utilization. If swapped-out data however are not contained in a close-by cache – a situation likely with noninclusive caching – swapping out logical threads becomes too expensive to be compensated. The effect is amplified by overhead necessary to administer the task queues and sequentialization and synchronization effects stemming from work stealing, for example.

8. Additional deep memory (Intel Optane technology)

Finally, we scale up our problem size such that it does not fit into our conventional memory anymore. We use Intel Optane technology to accommodate this larger memory footprint. Hence, the conventional memory becomes a fully associative L4 cache. Intel’s hardware is responsible for bringing data into and out of the cache. Agnostic of the particular data movement strategy, we may assume that high temporal and spatial locality (Kowarschik and Weiß, 2003) in the memory accesses continues to be advantageous. Agnostic of the specific hardware properties, we may assume that ‘main memory cache misses’ suffer from higher latency.

For the benchmarking, we start from two regular grids: 27^3 and 81^3 . These regular grids are denoted by $\Delta\ell = 0$, as we add 0 levels of dynamic adaptivity. Different to previous setups where, for reasonably large setups, memory constrains the dynamic adaptivity criterion and allows it to add at most one level of grid refinement, we now also conduct experiments with up to two additional resolution levels of dynamical adaptive meshes ($\Delta\ell \in \{1, 2\}$). All data are normalized against the real DoFs and the number of time steps, as explicit hyperbolic equation solvers require the time step size to scale with the (adaptive) mesh size. We measure the cost per DoF updates per time step. As we found the large setups to crash with multicore support – the per-thread call stack was exceeded – we made each thread outsource its significant temporary local data structures to the heap. This is less efficient than on-stack storage and reduces the scalability, yet has to be done for all experiments here to obtain consistent data. Future versions of TBB will fix this stack size problem.

We spot a v-shaped cost profile for the setups fitting completely into memory (Figure 8). The cost per DoF update increases with p , an effect in practice more than compensated through the higher order of the approximation. Furthermore, we see that this increase is more than made up as long as $p \leq 6$. Vector units are used more efficiently (Tables 1 and 2). Once $p \geq 7$, the increase in cost also materializes in increased runtimes. The memory of the individual compute steps exceeds close caches (cf. L2 Request Rate in 2). We start to suffer from L2 or L3 cache misses. The v-pattern translates into energy per DoF update, too.

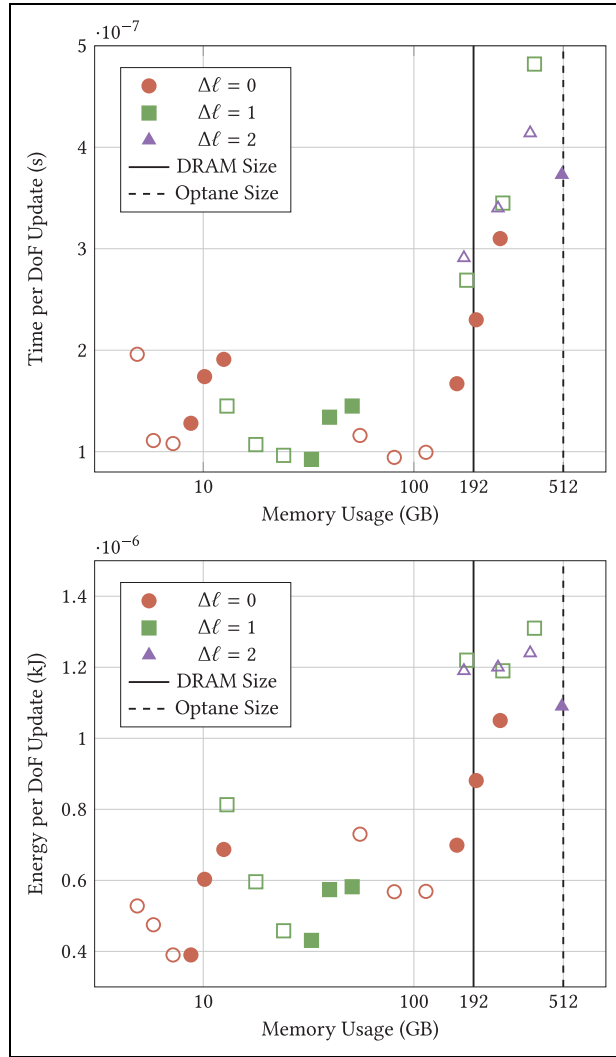


Figure 8. Intel Xeon Scalable experiments for various problem sizes. Setups left of the dotted line use no Intel Optane technology as everything fits into the DRAM, so we switch it off. Top: Time per DoF update. Bottom: Total energy usage. Each setup is, as long as it fits into the memory, computed multiple times (from left to right): We start from a base grid of $27 \times 27 \times 27$ (circles) and test six polynomial orders $p \in \{4, 5, 6, 7, 8, 9\}$ with $p \leq 6$ denoted through empty symbols. We then add one level of adaptivity ($\Delta\ell = 1$, squares). We next rerun the regular grid experiment with a base grid of $81 \times 81 \times 81$ (circles) and one level of adaptivity (squares). A base grid of $27 \times 27 \times 27$ with two levels of adaptivity (triangles) already requires Intel Optane technology unless we choose $p = 3$. DoF: degree of freedom.

Once our regular grid setup exceeds main memory, we experience a runtime penalty. For the low-order experiments, this penalty is significantly below a factor of three which would mirror the fact that the hardware has higher latency, too. With increasing orders, the penalty increases.

8.1. Observation 9

Trading bandwidth for latency does *not* work for our code. We suffer directly from increased latency.

With the Intel Optane technology, the v-pattern is distorted. The higher our bandwidth demands, the higher also the LLC misses and the higher the runtime penalty of the Intel Optane technology. We observe that the most aggressive adaptivity pattern $\Delta\ell = 2$ now yields a better cost per DoF update ratio than the more regular discretizations. The higher the polynomial order, the more significant this effect. As the dynamically adaptive mesh intermixes memory-intensive and arithmetically demanding tasks stronger, the runtime penalty induced by the Intel Optane technology is more significant for the other grid setups. For $\Delta\ell = 1$, we have not been able to reproduce this effect which might be due to the fact that we ran into memory limits.

8.2. Observation 10

We find the simulation for dynamically adaptive meshes being better suited to Intel Optane technology than a regular grid/fixed mesh setup.

We consider it to be a pattern of many important HPC codes: data access exhibits stream access behaviour close to the compute core – here notably for high orders. On a higher abstraction level, codes however rely on flexible, dynamic tasks and thus do not fit to hardware tailored to stream access. Yet, with many cache levels, the arising non-local data accesses do not hammer the last level memory. As long as not too many codes access the memory concurrently, the latency penalty remains under control. Runtimes should thus intermix computationally heavy- and memory-demanding tasks.

While our code exhibits no clear correlation of adaptivity pattern and polynomial order to energy cost in the main memory, we do observe that the usage of Intel Optane technology increases the energy footprint. Future work will have to analyse whether persistent memory modes can bring down these increased energy cost.

9. Summary and conclusion

Our manuscript studies a non-trivial solver for PDEs. We consider it to be characteristic for many upcoming simulation codes: it relies on many tasks of different compute character; the runtime of the tasks and the task composition are hard to predict – AMR plays a major role here and the situation might become more severe once non-linear equations are solved which require localized Newton or Picard iterations; finally, the efficiency of the solve hinges on the opportunity to use high polynomial orders and fine meshes. The solver requires massive memory.

Computer memory designers operate in a magic triangle of size, bandwidth and latency. Under given energy and cost constraints, not all three of these characteristics can be improved. While caches optimize for bandwidth and latency, the new Intel memory optimizes for size and bandwidth. At the same time, a core increase amplifies NUMA effects for low-order and cheap (Riemann) tasks. As we find our code suffers from memory latency in general, we hypothesized that it may pay off to reduce the core

frequency relative to the memory frequency, to use core oversubscription, to hide latency penalties and to regularize and homogenize all computations, that is, to work with as regular data structures and task graphs as possible. We have not been able to confirm these hypotheses in general. However, we have found or confirmed attractive alternative solutions or solution proposals per diversity axis.

An increasing flexibility and heterogeneity of clock frequencies allows chips to alter the frequencies for individual system parts. We have not been able to exploit this feature to soothe the impact of latency, although we have confirmed the well-known insight that drastically decreased frequencies improve the energy efficiency of the simulation. Yet, our data suggest that the turbo boost feature of modern chips is of use for very heterogeneous task graphs. It significantly improves the runtime while the energy demands remain under control. It might be reasonable to downclock chips overall, but to allow the runtime to increase the frequency of particular cores starting from the reduced baseline up to the turbo boost frequency. Those cores producing further tasks and running computationally cheap tasks would benefit from such a feature. Such a fine-granular frequency alteration feature – likely coupled with a task runtime – seems to be promising.

An increasing core count and thus NUMA heterogeneity amplify NUMA effects which we label as growing horizontal diversity. Our data suggest that it might be reasonable to subdivide large shared memory chips into logically distributed memory systems. ExaHyPE is written as MPI + TBB code relying on tasks. Here, it makes sense to have at least one rank per node per socket to keep NUMA effects under control. Our benchmarks furthermore clarify that existing cache optimization techniques – notably a high data access localization – continue to pay off. They help to soothe the impact of massively increased latency. In return, however, overbooking is not an option to hide latency/NUMA effects as vendors give up on inclusive caching. It is future work to study whether runtimes explicitly copying data from persistent/large-scale memory into the ‘right’ part of the main memory – which effectively becomes the LLC – can help to eliminate NUMA effects (Kudryavtsev, 2018). In this case, future runtimes have to be equipped with the opportunity to predict the task execution pattern and to replace classic prefetching with explicit memory moves.

In the case of ExaHyPE, a homogenized task parallelism which mixes tasks of different compute characteristics allowed us to hide some latency of the Intel Optane technology. Our code has been able to cope with the increased vertical memory diversity. We show that it is absolutely essential to equip tasking systems and algorithms with the opportunity to run memory-intense and compute-bound tasks concurrently, while the majority of compute-intense jobs has to exhibit data access locality. If we get the balance between bandwidth and compute demands right, latency effects remain under control. The access pattern has to be homogenized. Future task systems should internally be

sensitive to the compute character of the tasks. They have to mix compute-intense jobs with memory-intense jobs to avoid that a whole node waits for slow deep memory. This naturally can be mapped onto job priorities and mechanisms ensuring that not too many jobs of one priority are launched. To the best of our knowledge, current runtimes as found with OpenMP, TBB or C++11 lack mature support for such priorities or constraints.

Machines equipped with Intel Optane technology provide ample memory. It is an appealing alternative to classic ‘fat nodes’; also in terms of procurement cost. Once the exascale era makes the total power budget of computers grow to tens of megawatts, it is an option to trade, to some degree, the DRAM for an energy-modest extra layer of memory. This article’s experiments navigate at the edge of ‘fits into the memory’ and thus provide too few experimental samples to support claims through frequently observed patterns. We need to run more experiments with more hardware configurations and more applications. Yet, our results suggest that the way forward into the massive-memory age might not be a naive rendering of the main memory into an additional cache layer; at least not for non-trivial/non-streaming codes. Instead, we ask for three architectural or software extensions: fine-granular frequency control, runtimes with explicit data prefetching and runtimes with mature task priorities; the latter perhaps even guided by the availability of task data in close caches.

Acknowledgements

The authors appreciate support received from the European Union Horizon 2020 research and innovation programme. This work made use of the facilities of the Hamilton HPC Service of Durham University. Particular thanks are due to Henk Slim for supporting us with Hamilton. Thanks are due to all members of the ExaHyPE consortium who made this research possible; notably J.-M. Gallard for integrating aggressively optimized compute kernels into ExaHyPE and K. Duru, A.-A. Gabriel as well as L. Rannabauer for realizing the seismic benchmark on top of ExaHyPE. The authors are particular thankful to L. Rannabauer for the support on running the seismic benchmarks. All underlying software is open source (Bader et al., 2014–2019).

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the European Union Horizon 2020 research and innovation programme under grant agreement no. 671698 (ExaHyPE).

ORCID iD

Tobias Weinzierl  <https://orcid.org/0000-0002-6208-1841>

References

- Bader M, Dumbser M, Rezzolla L, et al. (2014–2019) ExaHyPE—an exascale hyperbolic PDE solver engine. Available at: <http://www.exahype.eu> (accessed 2 April 2019).
- Boyandin K (2018) Guest post: Intel Optane and in-memory databases. Available at: <https://blog.selectel.com/guest-post-intel-optane-and-in-memory-databases> (accessed 29 August 2018).
- Breuer A, Heinecke A, Rannabauer L, et al. (2015) High-Order ADER-DG Minimizes Energy- and Time-to-Solution of SeisSol. In: Kunkel J and Ludwig T (eds.) *High Performance Computing. ISC High Performance 2015, Lecture Notes in Computer Science*. Cham: Springer, pp. 340–357.
- Charrier D, Hazelwood B and Weinzierl T (2018) Enclave tasking for discontinuous Galerkin methods on dynamically adaptive meshes (arXiv:1806.07984).
- Charrier D and Weinzierl T (2018) Stop talking to me—a communication-avoiding ADER-DG realisation (arXiv:1801.08682).
- Day S and Bradley C (2001) Memory-efficient simulation of anelastic wave propagation. *Bulletin of the Seismological Society of America* 91(3): 520–531.
- Dubey A, Almgren A, Bella J, et al. (2016) A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing* 74(12): 3217–3227.
- Dumbser M and Käser M (2006) An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case. *Geophysical Journal International* 167(1): 319–336.
- Glesser D (2016) Road to exascale: improving scheduling performances and reducing energy consumption with the help of end-users. PhD Thesis, Grenoble Alpes.
- Hutchinson M, Heinecke A, Pabst H, et al. (2016) Efficiency of high order spectral element methods on petascale architectures. In: Kunkel Julian M., Balaji Pavan and Dongarra Jack (eds.) *High Performance Computing. ISC High Performance 2016, Lecture Notes in Computer Science*. Vol 9697. pp. 449–466. Cham: Springer International Publishing.
- Kowarschik M and Weiß C (2003) An overview of cache optimization techniques and cache-aware numerical algorithms. In: Meyer U, Sanders P and Sibeyn JF (eds.) *Algorithms for Memory Hierarchies 2002, Lecture Notes in Computer Science*. Vol 2625. pp. 213–232. Cham: Springer.
- Kudryavtsev A (2018) Optane and Intel memory drive technology, big surprise. Available at: <https://itpeernetwork.intel.com/optane-intel-memory-drive-technology> (accessed 10 December 2018).
- McCalpin J (1995) Memory bandwidth and machine balance in current high performance computers. In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*. pp. 19–25.
- Microway (2018) Detailed specifications of the Intel Xeon E5-2600v4 Broadwell-EP processors. Available at: <https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-intel-xeon-e5-2600v4-broadwell-ep-processors> (accessed 14 December 2018).
- Reinders J (2007) Intel threading building blocks. O'Reilly. The SPICE Code Validation (2006) Problem wp2_loh1. Available at: http://www.sismowine.org/model/WP2_LOH1.pdf (accessed 2 April 2019).
- Treibig J, Hager G and Wellein G (2010) LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: *Proceedings of the 2010 39th international conference on parallel processing workshops, ICPPW '10* (eds Lee WC and Yuan X), San Diego, California, USA, 13–16 September 2010, pp. 207–216. IEEE Computer Society.
- Weinzierl T (2018) The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. *ACM Transactions on Mathematical Software* (accepted; arXiv:1506.04496).
- Weinzierl T and Mehl M (2011) Peano—A traversal and storage scheme for octree-like adaptive Cartesian multiscale grids. *SIAM Journal on Scientific Computing* 33(5): 2732–2760.
- WikiChip (2018) Intel Xeon Gold 6150. Available at: https://en.wikichip.org/wiki/intel/xeon_gold/6150 (accessed 10 December 2018).

Author biographies

Dominic E Charrier is a PhD student at Durham University's Department of Computer Science, where he studies ADER-DG under the direction of Tobias Weinzierl. He is predominantly interested in high-performance computing aspects and adaptive mesh refinement for this family of high-order finite element methods. His research is funded by the EU Horizon 2020 project ExaHyPE.

Benjamin Hazelwood received a masters by research degree from Durham University under the direction of Tobias Weinzierl. His research orbits around redundancy in MPI codes to support resiliency and classic performance engineering of the ADER-DG method. His research is funded by the EU Horizon 2020 project ExaHyPE.

Ekaterina Tutlyaeva is an engineer programmer at the Scientific and Application Research Department of the company RSC Technologies in Russia. She started a professional high-performance computing career in 2006 in the Research Center for Multiprocessor Systems of the Program System Institute of the Russian Academy of Science and obtained a Specialist Degree in applied mathematics and informatics at the Pereslavl University in 2009. The RSC Group is an industry partner in the EU Horizon 2020 project ExaHyPE.

Michael Bader is an associate professor at the Department of Informatics at the Technical University of Munich. He works on hardware-aware algorithms in computational science and engineering and high-performance computing. In particular, he focuses on challenges imposed by latest supercomputing platforms, and the development of suitable efficient and scalable algorithms and software for simulation tasks in science and engineering. He is the PI of ExaHyPE.

Michael Dumbser became an associate professor for numerical analysis at the University of Trento in 2011. Since November 2018, he is a full professor of numerical analysis. His current research interests are adaptive mesh refinement with time-accurate local time stepping, high-order DG finite element schemes with a posteriori limiters for hyperbolic partial differential equations, Lagrangian schemes on moving unstructured meshes and novel structure-preserving semi-implicit schemes for continuum mechanics. He collaborates with Michael Bader and Tobias Weinzierl on the ExaHyPE project.

Andrey Kudryavtsev is an SSD Solution Architect at Intel and has more than 15 years of total server experience. He holds a Computer Science degree from Nizhny Novgorod

State University in Russia and is currently based in Folsom, CA, USA, where he is a member of the Solution Architecture team at the Non-Volatile Memory Solutions Group at Intel.

Alexander Moskovsky is the CEO and co-founder of RSC Technologies, part of the RSC Group. RSC is the leading Russian innovative HPC solution provider and developer. He received an MSc and PhD degrees from Moscow State University in 1997 and 2001, respectively. Prior to RSC, he held research positions at the Russian Academy of Sciences and worked in software engineering at VDI (now EPAM) and the Digital Equipment Corporation. The RSC Group is an industry partner in the EU Horizon 2020 project ExaHyPE.

Tobias Weinzierl is an associate professor for high performance and scientific computing at Durham University. He holds a PhD and habilitation from the Technical University of Munich, where he also served as Scientific Programme Manager for the Munich Centre of Advanced Computing. He works primarily on efficient multiscale methods, adaptive mesh refinement and parallelization approaches for large-scale partial differential equation solvers. He collaborates with Michael Bader and Michael Dumbser on the ExaHyPE project.