Dynamic task fusion for a block-structured finite volume solver over a dynamically adaptive mesh with local time stepping [⋆]

Baojiu Li[1][0000−0002−1098−9188], Holger Schulz[2][0000−0003−3428−8922], Tobias Weinzierl[2,3][0000−0002−6208−1841], and Han Zhang[1][0000−0002−7651−1179]

[1] Institute for Computational Cosmology, Durham University, DH1 3FE Durham
[2] Department of Computer Science, Durham University, DH1 3FE Durham
[3] Institute for Data Science, Large-Scale Computing, Durham University, DH1 3FE Durham
`tobias.weinzierl@durham.ac.uk`

Abstract. Load balancing of generic wave equation solvers over dynamically adaptive meshes with local time stepping is difficult, as the load changes with every time step. Task-based programming promises to mitigate the load balancing problem. We study a Finite Volume code over dynamically adaptive block-structured meshes for two astrophysics simulations, where the patches (blocks) define tasks. They are classified into urgent and low priority tasks. Urgent tasks are algorithmically latency-sensitive. They are processed directly as part of our bulk-synchronous mesh traversals. Non-urgent tasks are held back in an additional task queue on top of the task runtime system. If they lack global side-effects, i.e. do not alter the global solver state, we can generate optimised compute kernels for these tasks. Furthermore, we propose to use the additional queue to merge tasks without side-effects into task assemblies, and to balance out imbalanced bulk synchronous processing phases.

Keywords: Task-based programming · Block-structured dynamic adaptive mesh refinement · Local time stepping · Wave equation solvers

1 Introduction

Dynamic adaptive mesh refinement (AMR) and local time stepping (LTS) are algorithmic key ingredients behind many efficient simulation codes, i.e. codes that

⋆ The authors acknowledge the support through the embedded CSE programme of the ARCHER2 UK National Supercomputing Service (http://www.archer2.ac.uk) under grant no ARCHER2-eCSE04-2, Durham's oneAPI Academic Centre of Excellence made by Intel, ExCALIBUR's Phase Ia grant ExaClaw (EP/V00154X/1) and ExCALIBUR's cross-cutting project EX20-9 Exposing Parallelism: Task Parallelism (grant ESA 10 CDEL). They furthermore received support through the European Research Council via grant ERC-StG-716532-PUNCA, the STFC Consolidated Grants ST/T000244/1 and ST/P000541/1, and the China Scholarship Council (CSC) studentship at Durham University. This work has made use of the Hamilton HPC Service of Durham University.

invest compute resources where they pay off most. We focus on the simulation of waves, i.e. hyperbolic partial differential equations (PDEs), from astrophysics which are solved via block-structured Finite Volumes with explicit time stepping. The combination of AMR plus LTS makes it challenging to load balance, as the workload per subdomain changes per time step. These changes are often difficult to predict and are not spatially clustered. Any geometric data decomposition thus has to yield (temporal) imbalances. Modern supercomputer architectures however exhibit and will exhibit unprecedented hardware parallelism [9], which we have to harness in every single simulation step.

Task-based programming promises to ride to the programmers' rescue. Tasks [3,8] allow the programmer to oversubscribe the system logically, i.e. to write software with a significantly higher concurrency compared to what the hardware offers. A tasking runtime can then efficiently schedule the work modelled via tasks, i.e. map the computations onto the machine. The task runtime and its task stealing eliminate imbalances per rank [8]. In practice, tasking as exclusive overarching parallelisation paradigm however often yields inferior performance compared to traditional domain decomposition with bulk-synchronous processing (BSP). There are three main reasons for this: First, tasking introduces significant administrative overhead if tasks become too fine grained. Yet, if programmers reduce the task granularity, they sacrifice potential concurrency. This is problematic in our context: If too few Finite Volumes are clustered into a task, the task becomes too cheap. If too many Finite Volumes are clustered, we lose concurrency and constrain the AMR and LTS. The dilemma becomes particularly pressing for non-linear PDEs where eigenvalues and, hence, admissible time step sizes per task change—we hence can not cluster multiple tasks that are always triggered together [25]. Second, assembling the task graph quickly becomes prohibitively expensive and introduces algorithmic latency if the graphs are not reused, change at runtime, or are massive. Yet, if codes issue exclusively ready tasks, we deny the runtime information and any scheduling decision thus might, a posteriori, be sub-optimal. Finally, generic task code is often not machine-portable. Indeed, we find that different machines require different task granularities. CUDA or KNL-inspired architectures for example prefer larger tasks with limited data transfer compared to a mainstream CPU [17,22,27,29] and they penalise too many kernel launches [29].

We study a generic Finite Volume code for wave equations that can handle different PDEs via functors: It spans a mesh and provides generic numerical kernels, while the actual PDE terms are injected via callback functions [26]. Our code is based upon the second generation of the ExaHyPE engine which inherits all fundamental principles from the first generation [18]. It is similar to others solver in the field such as SpECTRE [15] as it relies on generic Riemann solvers. The code features unconstrained dynamic AMR which can change the mesh in each and every time step, patch-local LTS, and a combination of traditional non-overlapping domain decomposition and a task formalism.

Our novel idea is that the tasks are classified into urgent, i.e. potentially along the critical path, and non-urgent. The latter are called enclave tasks [6].

The urgent tasks are embedded into the actual BSP code, i.e. the mesh traversal of the non-overlapping domains. They are executed immediately. Enclave tasks can be held back in a user-defined task queue and are not instantaneously passed on to the tasking runtime. Instead, we analyse where BSP imbalances arise and just take as many tasks from the user queue as we need to compensate for them. Alternatively, we can spawn them as native tasks on top of the BSP parallelism and rely on the runtime to compensate for imbalances. Furthermore, we label the enclave tasks that have no global side-effects, i.e. do not alter the global solver's state. These tasks are processed by specialised, optimised compute kernels and can be batched into task sets which are processed en bloc. Different to other approaches, our code does not geometrically cluster or combine tasks, and it does not rely on a priori knowledge of the task creation pattern. We retain all flexibility and yet obtain compute-intense batches of tasks, while we do not assemble any task graph at all.

To the best of our knowledge, such a flexible approach has not been proposed before. Our working hypothesis is that the batching allows us to work with small tasks and still get high performance compute kernels, that a task graph assembly is not necessary and actually inadequate for LTS plus AMR, and that task-based programming plus domain decomposition outperforms plain BSP. We study the impact of our ideas for two astrophysical problems, though the principles apply to a lot of mesh-based PDE solvers and even Lagrangian methods which typically suffer from too fine-granular tasks.

The remainder of the paper is organised as follows: We briefly sketch our application area of interest and our benchmarks (Sect. 2) before we introduce our numerical ingredients and the resulting software architecture (Sect. 3). In Sect. 4, we discuss the four different levels of parallelisation applied, highlight the rationale behind these levels and the arising challenges. The key ideas of the present paper are sketched in the introductory paragraph of Sect. 5 before we provide details on their realisation. Section 6 hosts some numerical experiments. A brief outlook and summary (Sect. 7) close the discussion.

2 Applications

Our equations of interest are hyperbolic PDEs given in first-order formulation:

$$\frac{\partial Q}{\partial t} + \nabla \cdot F(Q) + \sum_{i=1}^{d} B_i(Q) \frac{\partial Q}{\partial_{x_i}} = S(Q) \qquad \text{with } Q : \mathbb{R}^{3+1} \mapsto \mathbb{R}^N. \tag{1}$$

They describe time-dependent wave equations. Conservative PDE formulations comprise a term $\nabla \cdot F(Q)$, otherwise non-conservative product (ncp) terms enter the equation. These B_is act on the directional derivatives. We implement two astrophysics scenarios on top of (1):

2.1 Modified Euler equations: Secondary Infall

Our first simulation is a secondary infall test in a hydrodynamic system on an expanding (cosmological) background, driven by Newtonian gravity. It studies

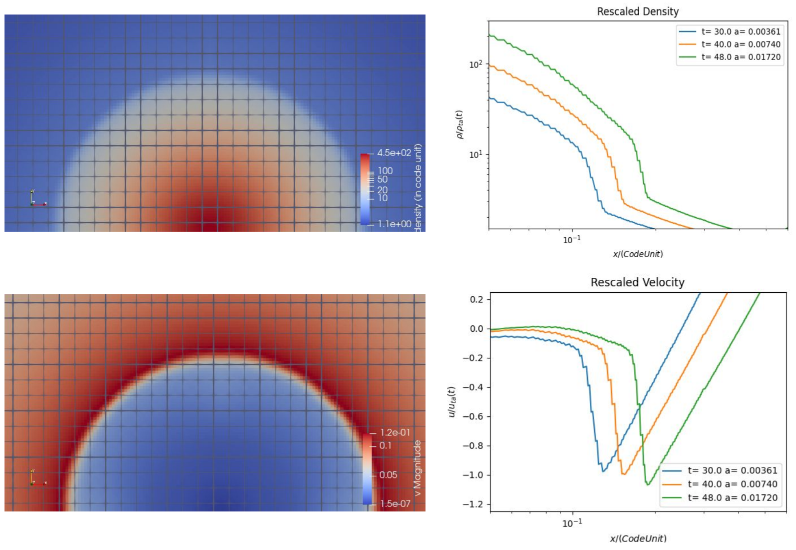

Fig. 1. Left: Characteristic density and velocity fields from the collisional secondary infall simulation. Due to the gravitational infall of mass, an outer-propagating shock appears. Right: Rescaled density and velocity profiles in radial direction.

spherically-distributed mass (collisional baryonic or collisionless dark matter) collapsing under self-gravity, which is an important testing scenario in galaxy formation. One property of interest is the theoretically predicted self-similarity after rescaling in the solution [5]: The density, pressure and velocity profiles depend exclusively on time, i.e. preserve their shape (Fig. 1). Our own studies with the present software suggest that this property is even preserved (approximately) for a set of non-standard graviational models [28]. The underlying equations are modified compressible Euler equations over five unknowns $\rho, E \in \mathbb{R}$ and $\mathbf{j} \in \mathbb{R}^3$. ρ, $\mathbf{j}$, E represent the density of mass, momentum and energy respectively. The self-gravitation enters the equations via a source term $S(Q)$ which depends upon $\rho\mathbf{g}$. $\boldsymbol{g}$ is the gravitational acceleration, which is proportional to $m_{<r}/r^2$where $m_{<r}$ is the total (overdensity) mass within a given radius r around the infall's centre. This system is a flavour of (1) with $N = 5$, where $B_i = 0$.

The F term is independent of the global solution behaviour. This does not hold for the right-hand side: The value of the mass function $m_{<r}$ requires global information from all volumes within r. We construct a mass array $\{m_i\}_{0\leq i\leq \max}$ storing the total mass within radii $\{r_i\}_{0\leq i\leq \max}$. The m_i values are calculated by accumulating the mass within r_i per time step. During the subsequent time step, we feed the following interpolation into the source term $S(Q)$:

$$m_{<r} = \begin{cases} m_0 r^3/r_0^3, & r \leq r_0 \\ m_i \frac{r_{i+1}-r}{r_{i+1}-r_i} + m_{i+1}\frac{r-r_i}{r_{i+1}-r_i}, & r_i < r \leq r_{i+1} \\ m_{\max} + \frac{4\pi}{3}\rho(r_{\max})\left(r^3 - r_{\max}^3\right), & r > r_{\max} \end{cases} \tag{2}$$

The interpolation works with bucketing, i.e. the $m_{<r}$ is discretised, while the modifications in (2) near and far from center allow us to accumulate only the mass values within a finite area around the centre. We work with a finite number

of shells or spheres for which we calculate the actual mass. In a Finite Volume setting, volumes far away from the centre do not feed into the subsequent S evaluations. Their behaviour is local, i.e. the time stepping's update of such volumes has no global side-effect. As the solution Q evolves smoothly in time, we can evolve $\{r_i\}_{0\leq i\leq \max}$ smoothly in time, too, such that the error due to the discretisation into spheres remains bounded.

2.2 CCZ4 GR equations: Gauge Waves

Our second tested scenario solves a complete numerical relativity system based upon the CCZ4 formalism in vacuum [2]. The equations simulate strong gravity in regions where the Newtonian approximation is invalid. The textbook version of the vacuum CCZ4 system involves 25 variables and is second order in space. It also is subject to constraints. We follow [12] and recast the equations into first order by introducing auxiliary variables. This expands the whole system to $N = 59$ variables in (1). The auxiliary variables allow us to eliminate the second-order terms. They also map the constraints onto Lagrangian parameters, i.e. the Einstein constrain equations are enforced weakly.

Our code is a complete C++ rewrite of the equations (12a)-(12m) as published in [12] which absorbs the flux terms within the ncp, i.e. $F(Q) = 0$. Both the eigenvalue and ncp calculation are phrased as dense tensor products (like $2\alpha\tilde{\Gamma}^i_{jk}\tilde{A}^{jk}$), i.e. yield a high arithmetic intensity.

In this paper, we assess the code performance through the gauge wave test from [1]. This setup is derived from a time dependent coordinate transformation of the flat Minkowski metric: The spacetime is trivial, which is a property we do not exploit further. We expect a standing wave which travels through the domain and enters on the other side. The eigenvalues are invariant, and the wave's amplitude and frequency do not change over time subject to numerical dissipation. Different to the previous setup, we need periodic boundary conditions.

3 Software and solver architecture

We solve both challenges with first-order explicit time stepping (explicit Euler) on block-structured adaptive meshes. Finite Volumes with a generic Rusanov solver for the arising Riemann problema deliver our spatial discretisation.

3.1 Dynamically adaptive AMR

We discretise the computational domain Ω_h through an octree or spacetree formalism, as it is state-of-the-art in the field [7,10,11,14,21,23,26]. Let the cubic domain Ω_h represent the root of a tree data structure. Equidistant, axis-aligned cuts through the cube along each Cartesian coordinate axis yield children of the tree and span a mesh. For each of these cubes, we decide independently whether we continue to refine recursively or not. We end up with an adaptive Cartesian mesh of *cells*, where each cell is a cube. The cells can have different sizes and hanging vertices are allowed, i.e. the mesh is non-conformal. We do not impose any balancing [14,21].

Each unrefined cell, i.e. cell which is not refined further, hosts a $p \times p \times p$ Cartesian grid. $p \geq 2$ is globally fixed. The elements within this embedded grid are called *volumes*. All the volumes within one cell form a *patch*. The resulting computational mesh of volumes is adaptive and block-structured. The patch formalism can also be read as a non-overlapping domain decomposition, where the patches of different sizes tessellate the computational domain.

The terminology around block-structured adaptive mesh refinement (AMR) is ambiguous. Different to other papers in the field [11], we employ the phrase "block-structured" but make all patches within our mesh host the same number of volumes. Different cardinalities for the subdivision within the tree are popular: Splitting into two parts along each coordinate axis yields the classic quadree or octree scheme [14,21], while splitting into $k > 2$ parts yields shallower trees. For $p = k$, our overall formalism results in a tree with fixed branching factor. For $p > k$, we can read our mesh as inhomogeneous tree using the same branching factor of k^d everywhere besides the very last level, where the branching factor suddenly becomes p^d. It becomes a hybrid combination of the ideas behind cell-by-cell refinement and patch-based AMR [23]. To match a given spatial accuracy, different choices of k and p can yield volumes of sufficiently small size yet with different topologies. Our code uses $k = 3$ [26] but leaves p as free parameter.

3.2 Finite Volumes

Let every volume carry a constant shape function. This defines our *finite volumes*. We apply an explicit Euler discretisation with time step size T, i.e. assume that all shape functions per patch remain constant over a time interval of length T. A weak formulation of (1) allows an integration by parts and the Gauss divergence theorem to yield a textbook Finite Volume scheme. Along the volume interfaces, our implementation uses the Rusanov flux

$$\begin{aligned}\text{flux}_n^{\pm}(Q)|_{\partial v} = \frac{1}{2}F_n(Q^+) + \frac{1}{2}F_n(Q^-) \mp \frac{1}{2}(Q^+ - Q^-)B_n(\frac{Q^+ + Q^-}{2}) \\ - \max(\lambda_{\max}(Q^+), \lambda_{\max}(Q^-))(Q^+ - Q^-)\end{aligned}$$

with left/right values $Q^{\pm}$ and maximum eigenvalues over the Jacobian.

The generic numerical scheme is tailored towards a particular application by specifying the number of unknowns N in (1) and by providing implementations of a source S and the maximum eigenvalue. As we work in a Cartesian world, we furthermore inject a flux F and the B_i for each of the three coordinate axes.

3.3 Compute kernels

Let each patch be surrounded by a halo layer of width one. If two face-connected patches host volumes of exactly the same size, such a halo layer is a copy of the volumes from the face-connected neighbouring patch. If a neighbouring patch hosts a coarser mesh, the halo volumes are filled with a linear interpolation of coarser data. If a neighbouring patch hosts a finer mesh, the halo volumes host the averaged data of finer meshes.

With the notion of a halo layer, we can phrase the Finite Volume update of a patch as a *compute kernel*: It accepts the patch data plus the halo and yields the next time step's solution via the PDE term realisations passed in by the user. The kernel requires an epilogue code which updates the kernel's halo layer for the subsequent time step. This is a functional programming approach: We have some generic numerical code which is passed functors that realise the actual physics. Our code uses virtual function calls for the functors. These virtual functions belong to a global solver object which hosts all global solver attributes.

3.4 Time stepping variants

We realise four different time stepping schemes. The simplest scheme is *fixed time stepping* which consists of two logical steps per time step and works with a given, time-invariant ΔT: (i) Update all Finite Volumes. For this, we run over all cells aka patches that host them. (ii) Analyse adaptive mesh refinement (AMR) criteria, i.e. refine and coarsen the mesh, and update all halo layers with the correct information from adjacent patches.

Our *adaptive time stepping* adds an additional epilogue to the kernel invocation and reduces ΔT: (i.a) Update all Finite Volumes. (i.b) Determine the maximum eigenvalue $\lambda_{\max}$ over all volumes within the patch and compute a patch-local admissible next time step size $\Delta T^{(\text{adm})} = Ch/\lambda_{\max}$. (ii.a) Realise the AMR and initialise the halo layers. (ii.b) Reduce a global admissible time step size and use this one for the next time step. Due to the global reduction, the adaptive time stepping ensures that the CFL condition is never violated even if the wave propagation speed changes over time.

Subcycling accepts that volumes of different size are subject to different (local) CFL conditions: (i.a) Augment each patch with a patch-local time stamp $T(v)$. (i.b) Run over all patches. Update the Finite Volumes within a patch if and only if all face-connected patches either carry the same time stamp or are ahead in time. Otherwise skip the patch. Use the time step size $\Delta T = \Delta \hat{T} \cdot h/h_{\max}$. $h_{\max}$ is the global coarsest volume size and h is the local patch's volume size. (i.c) Determine the normalised admissible patch-local time step size $\Delta \hat{T}^{(\text{adm})}(v) = Ch_{\max}/\lambda_{\max}$ (ii.a) Realise the AMR and initialise the halo layers. (ii.b) Reduce a global normalised admissible time step size $\Delta \hat{T}(v)$.

The *local time stepping* eliminates the strict coupling between neighbouring patches: (i.a) Augment each patch with a patch-local time stamp $T(v)$ and time step size $\Delta T(v)$. (i.b) Run over all patches. Update the Finite Volumes within a patch if and only if all face-connected patches either carry the same time stamp or are ahead in time. Use the local time step size $\Delta T(v)$. (i.c) Determine a new local time step size if the patch has been updated. (ii) Realise the AMR and initialise the halo layers.

Subcycling requires linear interpolation in time along mesh resolution boundaries. Local time stepping requires linear interpolation for each halo layer update.

For linear PDEs, it would yield subcycling. In our case, we solve non-linear PDEs. There is no clear 1:k^i time step relation between patches of different resolution, and even patches of the same mesh size are allowed to advance in time with different speed if their eigenvalues differ.

Some literature in the field [23] uses the term adaptive for subcycling. This is a reasonable decision once we observe that too small time step sizes introduce numerical diffusion and thus pollute the simulation result. In this context, one can argue that both our fixed time stepping and adaptive time stepping are inappropriate for dynamically adaptive meshes over non-linear PDEs. Larger volumes or volumes with small eigenvalues should advance in time faster.

3.5 Concurrency analysis

Once we have made a decision to update a patch, we are able to process it independently of other patches as we supplement each patch with a halo layer. The patch update becomes a task. In a regular mesh with $P \times P \times P$ volumes, we obtain a best-case concurrency level of $(P/p)^3$. Within our tree formalism, we know $\exists \ell : k^\ell \cdot p = P$ and we consequently could obtain the same mesh with patch sizes $\ldots, k^2 p, kp, p, p/k, p/k^2, \ldots$, too. Larger p reduce the concurrency of the patch updates yet increase the arithmetic weight per task. Smaller p yield higher concurrency yet increase the cost per Finite Volume update due to (time step) bookkeeping per patch and data movements for the halos, even though the arithmetic intensity per volume is independent of the choice of p.

Larger p also reduce the AMR cardinality. If shocks arise, smaller values of p allow us to resolve these features with fewer volumes. Finally, large p constrain the time step adaptivity, since all Finite Volumes within a patch advance in time with the same time step size. In the context of subcycling and the local time stepping, we see that both schemes yield permanently changing concurrency levels, as the number of patches that are updated per time step changes per step. This even holds if the AMR grid remains static or we have a regular mesh in the local time stepping scheme.

The halo data exchange between patches of the same size is embarrassingly parallel. In a distributed memory context, it however requires MPI data exchange. The halo data projections from coarse to fine impose no concurrency constraints either. The restriction, i.e. the averaging, requires some local synchronisation, as no two neighbouring patches may restrict into the same halo volume at the same time.

4 Parallelisation

4.1 Domain decomposition

In our code, the computational domain's adaptive mesh of cells is split along a space-filling curve (SFC) [4,13]: Curve segments of similar cell count are first distributed over the compute nodes (ranks), before we split each rank's subpartition once again along the SFC into one chunk per thread. Fewer chunks are used if the subpartitions would become too small. We end up with two levels of data parallelism: A distributed memory one and a shared memory one. Both realise the same SFC decomposition paradigm yet differ in the way they exchange data: The top level sends MPI messages around, while the shared memory domain decomposition copies memory segments.

Rationale 1. *The two-level domain decomposition maps directly onto plain SPMD and BSP programming: All MPI ranks run through the same code, spawn one large parallel region (parallel for) and make each thread run through one chunk of the domain.*

The two-level balancing yields a chains-on-chains problem [13]: The SFC linearises all patches into one long sequence of patches which we cut first into chunks per MPI rank and then into chunks per thread. We employ a uniform cost model which is a strong simplification for subcycling. More sophisticated chains-on-chains approaches tackle this aspect [16]. However, they intrinsically fail for local time stepping if the underlying PDE is non-linear. A uniform cost model also fails to reflect that adaptive mesh refinement with non-conformal meshes requires inter-resolution transfer operators at patch boundaries where the mesh resolution changes. These induce additional cost.

Challenge 1. *The BSP programming model applied to a geometric domain decomposition is inherently ill-suited to handle local time stepping codes with dynamic AMR for non-linear hyperbolic PDEs.*

4.2 Task decomposition

The update of the individual patches is embarrassingly parallel. This statement however ignores data dependencies along mesh refinement transitions, ignores that dynamic AMR is expensive due to memory allocations which do not arbitrarily scale up, and it ignores that the halo data exchange via MPI requires deterministic messaging, while periodic boundary conditions typically involve some sorting or mapping. Some patch update tasks might belong to the critical path of the overall task graph as they are more expensive than others or feed into data transfer.

We therefore realise the concept of enclave tasking [6] and map each time step onto two mesh sweeps: Let a skeleton cell be a cell which is flagged to be refined or coarsened, is adjacent to a coarser cell, adjacent to a subdomain boundary or adjacent to a periodic boundary. The code sweeps through the domain. Per skeleton cell, it invokes the compute kernel, triggers all halo data exchange along the domain boundaries and realises the adaptive mesh refinement. In a second sweep, the code receives this boundary data, and updates all halo data.

The non-skeleton cells are treated differently: The first, primary sweep spawns a task for the corresponding patch and memorises the task's number within the cell. In the secondary sweep, each thread waits for the corresponding task to terminate once it hits a non-skeleton cell. It subsequently works in the task's result into the mesh before it updates the halo layers.

Rationale 2. *We do not explicitly assemble a task graph. Instead,we spawn only ready tasks. These tasks should overlap with boundary data transfer, adaptive mesh refinement operations or mesh administration.*

Challenge 2. *The scheme bursts large numbers of ready tasks. These have to be held back until they can be used to compensate for BSP imbalances.*

Challenge 3. *To gain maximum flexibility to compensate imbalances, the patches (tasks) have to be small.*

4.3 Intra-patch concurrency

Finite Volume schemes spend the majority of their runtime on patch updates. Each patch update is a sequence of nested for loops with known cardinality. In the innermost loop, we invoke the user functions. Our code refrains from exploiting any patch-internal concurrency for the cores. A patch is an atomic task which is not subdivided further.

Rationale 3. *The patch updates have to yield the MFLOP/s, i.e. aggressive vectorisation has to result from the patch updates unless the user function itself is costly and vectorised.*

Challenge 4. *The compiler cannot vectorise over virtual function calls.*

5 Performance Optimisation

5.1 Optimised C++ kernels

Our implementation hides complexity from the domain code, as domain experts only write code for the PDE terms. They feed into generic compute kernels. Our per-patch optimisation efforts focus on those non-skeleton tasks which have no side-effects. A user-defined flag per cell indicates whether the patch kernel evaluation alters global variables, i.e. the m_i values, or not. For the remaining tasks, we use our generic compute kernel.

Exploit lack of side-effects We extract our Rusanov patch update into a C++ template function over the solver type and replicate all PDE terms in the user code by specialisations: For every PDE term within a solver class, we establish a second variant which is static, i.e. has class-bindings instead of object-bindings. This static flavour is stripped off all global data access routines, aka the $\{m_i\}$ accumulation. For (2), we hence optimise only patches outside of $r_{\max}$. Our patch update for side-effect free cells calls the static PDE term variant, i.e. we eliminate virtual function calls (cmp. Challenge 4).

Next, we remove all C++ expressions from the template kernels which prohibit the compiler to inline. This involves the elimination of initialisation lists which are internally mapped onto for or while loops by C++. As we know the exact loop ranges for all constructs through the template arguments, we can replace such loops, i.e. manually unroll them. It also implies moving the PDE term definitions into headers, annotating them with `__attribute__((always_inline))` and removing explicit template instantiations. Explicit template instantiations enable the compiler to generate object files and thus speed up the translation while the library sizes are reduced. However, once machine code is "hidden away" within object files, our compiler struggles to inline. Aggressive inlining allows the compiler to unroll the loops, to vectorise, and it enables us to store all temporary data on the call stack rather than the heap which we use in our generic kernel implementations accepting functors and any data cardinality.

Finally, we realise a patch update kernel which computes all the fluxes per volume, all eigenvalues and all ncp terms before it updates the image, as well as a kernel which updates all volumes in-situ. The former variant requires temporary arrays which can be realised in different storage formats (AoS or SoA). The in-situ approach runs over the faces, computes the flux/ncp contribution per face and immediately updates both adjacent volumes. Similar to a red-black Gauss-Seidel, we update the columns, rows and layers of the patch in an interleaved fashion such that no race conditions arise. As the Rusanov flux requires averaging between two cell fluxes plus left and right eigenvalues, many terms are calculated twice. Yet, we need no massive temporary data structures.

Batched kernels So far, we have two semantically different patch update routines: The standard routine is a generic numerical scheme which accepts N and p as parameters and is given one functor per PDE term plus the eigenvalue invocation. Each functor can alter the state of the underlying PDE solver. It can, for example, aggregate global data views. A second routine is used for cells hosting patches that do not alter the global solver state.

This second flavour can be extended into a third, batched variant: Assume we have $\hat{P}$ patches not causing side-effects, each with its own time stamp and time step size yet the same patch dimensions and PDE terms. We can then update all $\hat{P}$ patches in parallel. The lack of side-effects implies that there are no race conditions. This adds an additional loop over the patches to our compute kernels. The batched kernel variant takes the $\hat{P}$ patches and first computes the flux, e.g., of one volume of the first patch, then the second patch, and so forth, before it continues with the second volume. Despite small patches, the vectorisation now operates over a large iteration space (cmp. Challenge 3) which can be collapsed.

5.2 Task queues

We propose an optimised threading architecture, where each rank issues one OpenMP thread per rank-local subpartition. This is BSP. To compensate for the intrinsic imbalances (cmp. Challenge 1), we employ our task-based parallelism on top: When a traversal encounters a non-skeleton cell, i.e. a patch update that is not algorithmically latency-sensitive, it creates a task.

This task is not passed into the underlying task runtime. Instead, we store it in an application queue. Once ready tasks are spawned into a task system, they are typically "lost" to the application code, i.e. it is difficult to manipulate them anymore. Furthermore, a fundamental principle of task systems is that the application code hands the responsibility for when a task is executed over to the runtime. With the application queue, we keep control over the tasks: If tasks within the queue are required in a subsequent BSP traversal, i.e. if there's a task dependency, and if this task has not yet completed, our BSP code runs through the user queue, processes one task, and checks again for completion. It polls the queue. If the queue becomes empty, the underlying BSP thread yields. This architecture allows us to realise the following tasking schemes:

- *BSP tasking* All tasks in the application queue remain there until they are polled by a subsequent BSP section. This realises a lazy evaluation scheme, where tasks are evaluated upon demand. We postpone their execution.
- *Native tasking* Our native tasking scheme maps all tasks that hit the application queue directly onto OpenMP tasks. They are not buffered.
- *Fill* This extension of BSP uses an atomic counter within the BSP section. It is initialised with the number of SFC partitions per rank and decremented once a BSP thread has finished its thread-local mesh traversal. Each BSP thread thus sees how many companion threads are still working. As long as some other BSP threads are still active, a thread does not close the BSP section but grabs a task from the user queue and executes it directly. It "releases" tasks if and only if we spot imbalances on the BSP side (cmp. Challenge 2).
- *Specialise late.* In this extension of the fill variant, a thread waiting for companion BSP threads first checks if the next task in the queue causes no side-effects. If so, it invokes the optimised kernel. Otherwise, it calls the generic kernel.
- *Batch(#t) late.* If a BSP thread recognises that companion BSP sections run longer, it takes up to #t consecutive tasks from the queue if they all have the same task type and lack side-effects. It then invokes the batched, optimised kernel over this set of #t tasks. Tasks are fused into one large task (cmp. Challenge 3).
- *Specialise immediately or batch immediately.* In this variant of the fill strategy, we immediately check if we can invoke a specialised kernel on one task or over a sequence of tasks, respectively, whenever a task is spawned.

6 Results

We run our experiments on Durham's Hamilton 8 cluster. It hosts AMD EPYC 7702 64-Core processor, i.e. the AMD K17 (Zen2) architecture. The 2×64 cores are spread over two sockets, but they are organised in 8 NUMA domains. Each core has access to 32 kB exclusive L1 cache, and 512 kB L2 cache. The L3 cache is (physically) split into chunks of 16 MB associated with four cores.

We use the Intel oneAPI software stack with icpx 2021.4.0. Intel's MPI (version 2021.4) is used for the distributed memory parallelisation, though we disable MPI for the single-node measurements. The tasking and vectorisation are realised through OpenMP. We use the most aggressive generic compiler optimisation level (`-Ofast`) and native code generation (`-mtune=native -march=native`).

6.1 Single task (kernel) optimisations

Table 1. Single patch kernel performance for the modified Euler equations (top) and CCZ4 (bottom) for various patch sizes $p \times p \times p$. We compare the plain kernel with virtual functions (baseline) against kernels without side-effects which either run through the patches one by one or batch the operations. Three different realisations w.r.t. the data organisation and computation orchestration are studied. Per measurement, we present the time ($[t] = s$) per Finite Volume update incl. the subsequent CFL computation, while the fastest configuration and those slower than the baseline run are set in bold.

		Patch-wise			Batched		
p	Baseline	AoS	SoA	in-situ	AoS	SoA	in-situ
3	$1.12 \cdot 10^{-6}$	$8.28 \cdot 10^{-7}$	$8.81 \cdot 10^{-7}$	$\mathbf{4.27 \cdot 10^{-7}}$	$7.85 \cdot 10^{-7}$	$7.91 \cdot 10^{-7}$	$6.52 \cdot 10^{-7}$
4	$9.11 \cdot 10^{-7}$	$8.07 \cdot 10^{-7}$	$8.10 \cdot 10^{-7}$	$\mathbf{3.93 \cdot 10^{-7}}$	$7.35 \cdot 10^{-7}$	$7.34 \cdot 10^{-7}$	$6.21 \cdot 10^{-7}$
7	$7.91 \cdot 10^{-7}$	$7.43 \cdot 10^{-7}$	$7.85 \cdot 10^{-7}$	$\mathbf{3.54 \cdot 10^{-7}}$	$7.00 \cdot 10^{-7}$	$7.09 \cdot 10^{-7}$	$5.62 \cdot 10^{-7}$
8	$7.84 \cdot 10^{-7}$	$7.67 \cdot 10^{-7}$	$7.70 \cdot 10^{-7}$	$\mathbf{3.52 \cdot 10^{-7}}$	$6.90 \cdot 10^{-7}$	$6.87 \cdot 10^{-7}$	$5.63 \cdot 10^{-7}$
15	$7.99 \cdot 10^{-7}$	$7.48 \cdot 10^{-7}$	$7.72 \cdot 10^{-7}$	$\mathbf{3.44 \cdot 10^{-7}}$	$6.92 \cdot 10^{-7}$	$6.80 \cdot 10^{-7}$	$5.47 \cdot 10^{-7}$
16	$7.95 \cdot 10^{-7}$	$7.41 \cdot 10^{-7}$	$7.62 \cdot 10^{-7}$	$\mathbf{3.45 \cdot 10^{-7}}$	$6.85 \cdot 10^{-7}$	$6.87 \cdot 10^{-7}$	$5.47 \cdot 10^{-7}$
3	$1.84 \cdot 10^{-5}$	$1.73 \cdot 10^{-5}$	$1.70 \cdot 10^{-5}$	$\mathbf{1.17 \cdot 10^{-5}}$	$1.60 \cdot 10^{-5}$	$1.60 \cdot 10^{-5}$	$1.26 \cdot 10^{-5}$
4	$1.68 \cdot 10^{-5}$	$1.65 \cdot 10^{-5}$	$1.65 \cdot 10^{-5}$	$\mathbf{1.12 \cdot 10^{-5}}$	$1.50 \cdot 10^{-5}$	$1.53 \cdot 10^{-5}$	$1.22 \cdot 10^{-5}$
7	$1.56 \cdot 10^{-5}$	$\mathbf{1.57 \cdot 10^{-5}}$	$1.56 \cdot 10^{-5}$	$\mathbf{1.02 \cdot 10^{-5}}$	$1.45 \cdot 10^{-5}$	$1.45 \cdot 10^{-5}$	$1.10 \cdot 10^{-5}$
8	$1.55 \cdot 10^{-5}$	$\mathbf{1.70 \cdot 10^{-5}}$	$\mathbf{1.68 \cdot 10^{-5}}$	$\mathbf{1.03 \cdot 10^{-5}}$	$1.44 \cdot 10^{-5}$	$1.44 \cdot 10^{-5}$	$1.11 \cdot 10^{-5}$

We first assess the impact of our kernel optimisations in isolation. For this, we focus only on the runtime of the patch update routines without any time stepping, mesh administration, halo updates, and so forth, and compare the generic FV patch update implementation using functors with the optimised versions which are used for tasks without side-effects. Our results present data for various patch sizes on one core. The compiler optimisation report clarifies that the translator succeeds in vectorising the user functions themselves (baseline) or vectorises over the patch entries or the batch of patches respectively.

The inlining pays off for the Euler equations (Table 1). A tuning of the intermediate data formats (AoS vs. AoS) however is irrelevant. Instead, avoiding the storage of intermediate results and computing the eigenvalues and fluxes twice gives the best performance. For CCZ4, the evaluation of a single source, ncp or eigenvalue is already expensive and utilises all vector capabilities according to the compiler feedback. The aggressive inlining, collapsing and vectorisation of the patch loops harms the performance unless we use the in-situ updates or batching. In-situ without batching continues to yield the best performance overall. We have not been able to identify significant runtime differences between AoS, SoA and in-situ for the baseline code using virtual function calls.

Lessons learned 1. *With a distinction of patch updates with side effects from kernels without them as well aggressive inlining and vectorisation, the small patches manage to match the throughput of large regular patches of volumes.*

If we first compute all fluxes in one direction, then in the other, then all ncps, then all eigenvalues, until we eventually combine these ingredients to update the outcome, the compiler vectorises all calculations once we inline the function calls. Yet, it suffers from many memory accesses. It is better to compute the eigenvalues, flux and ncp feeding into the Rusanov flux redundantly and to update all unknowns in-situ, i.e. to minimise the memory accesses. With this update scheme plus the exploitation of no side effects, we speed up our code by a factor of two for a memory-bound, simplistic PDE. For complex PDE terms, the approach still reduce the runtime robustly by roughly 30%. It remains an open question if a switch of the intermediate data structures or the elimination of redundant calculations would be the method of choice for other architectures or PDE choices. Furthermore, we do not exploit the fact that the arising equation system is sparse due to the flat Minkowski metric. Our generic kernel evaluates many terms which degenerate to zero. It is unclear how such domain knowledge can help to improve the performance further.

6.2 Single node studies on regular grid

We next study the scaling behaviour on one node. Our first subject of interest is the performance of the code on a regular grid. Here, subcycling equals the global adaptive scheme. Our benchmarks use a patch size of $p = 4$, i.e. each patch hosts 64 volumes. The runs span a total time of $(0, 10^{-3})$ which allows the solution to unfold reasonably such that we find a whole spectrum of eigenvalues over the domain for the modified Euler equations, while the regular grid hosts 34,012,224 (Euler) or 6,751,269 (CCZ4) volumes.

For Euler, the adaptive time stepping outperforms a fixed time stepping, as it dynamically adopts the time step size to the actual solution's needs while the fixed time stepping has to be conservative (Fig. 2): We have to set the fixed time step to the minimum of admissible step sizes over the whole simulation time span. Due to the absence of any clustering [25] in our local time stepping, i.e. the opportunity for every patch to march with its very own ΔT, the individual patches get "out of sync" and only few patches can advance in time per grid sweep. Typically, we update around 0.2% of all patches at one point in time. Local time stepping hence does not pay off. Adaptive time stepping equals the fixed time stepping for CCZ4 once we pick the same, time-invariant time step size for both. We typically do not know the admissible time step sizes for such complex PDEs a priori and thus have to build in some slack. It manifests in translated speedup curves. Due to the time invariance, the local time stepping yields the same performance patterns, too.

All schemes scale reasonably up to half a socket. After that, their scaling deteriorates. The runtime eventually increases if we use too many cores. Even though the mesh is perfectly balanced and does not change, we have slight runtime imbalances between the BSP sections—we use one SFC subpartition per core—which the native tasking on top of BSP can mitigate. Holding back tasks in an application queue on top of the OpenMP runtime introduces overhead. The fill strategy is thus slower than native tasking. Our CCZ4 setups are smaller than the Euler experiments and thus run into the strong scaling earlier, while the more expensive patch updates imply that any BSP imbalance—due to periodic boundary conditions, e.g.—manifests in stronger performance differences. Consequently, any fill-up decision runs risk to delay the termination

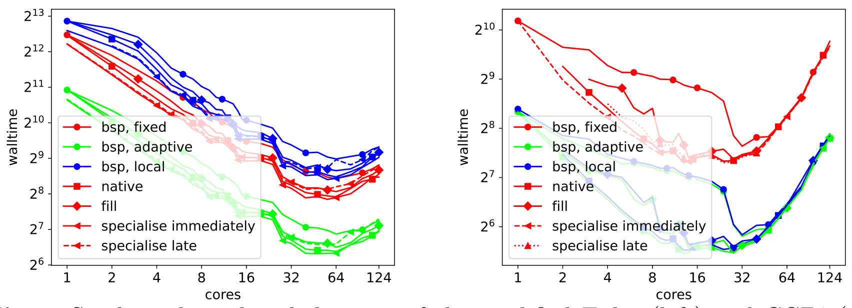

Fig. 2. Single node scaling behaviour of the modified Euler (left) and CCZ4 (right) with different time stepping and task orchestration schemes.

of the BSP section further. Once we identify tasks without side-effects and invoke specialised, optimised kernels, we get the better performance than for all other strategies. For reasonably small core counts, both the immediately spawning of specialised kernels and the holding back of kernels and (late) specialising upon-demand yield comparable performance.

Lessons learned 2. *A task parallelisation on top of classic BSP re-stabilises the code suffering from non-uniform load per BSP partition.*

We manage to preserve the positive impact of the kernel specialisations, though the tasking is responsible for the major runtime improvement. In total, we speed up the code by a factor of two. Our development experience suggests that this is possible if and only if we work with one additional user queue per core into which enclave tasks are "parked". This queue is merged into the global task queue once the corresponding BSP section producing tasks has terminated. If one global application queue is used right from the start, too many threads hit this queue and synchronise each other.

While the batching does not yield improved performance, it does not penalise the performance either as long as the ratio of work to cores remains high. On an EPYC, we have limited vectorisation potential. Furthermore, OpenMP tasks are tied to one core. It is thus not a surprise that we do not benefit from the batching, but the observation is encouraging for other systems, future runtimes and accelerators where individual tasks have to span multiple cores. This is facilitated by fused, large tasks.

6.3 Single node studies for adaptive grids

We continue with the Euler equations and 4^3 patches and let the AMR refinement follow the outer-propagating shock (Fig. 1). This is a challenging setup given the low arithmetic cost and high communication demands. All of our tests use a domain decomposition which is 90% accurate when the simulation starts, i.e. we compute the optimal load balancing and ensure that no rank is more than $\pm$ 10% off. We do not rebalance subpartitions throughout the test.

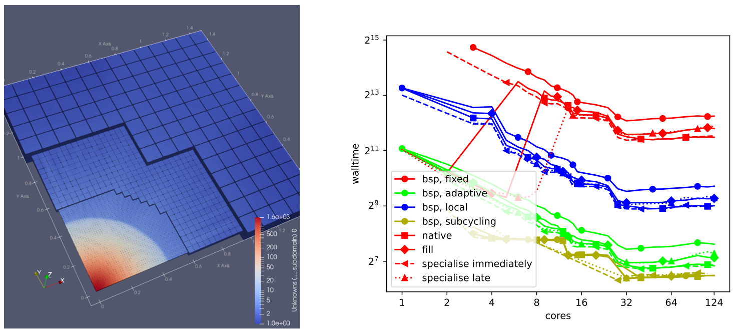

Fig. 3. Left: Typical adaptive mesh resulting from the secondary infall setup. Right: Single node scaling behaviour of the modified Euler over an adaptive grid. The mesh has three levels of adaptive refinement.

For an adaptive mesh, the task-based approach on top of BSP continues to outperform its BSP counterpart (Fig. 3). The latter suffers from AMR: As long as we use up to four cores and hence four partitions only, the BSP partitions might be imbalanced, but they produce many tasks which help to keep the cores busy. For more subdomains, fewer tasks are labelled as enclaves. The code fails to overlap expensive, "unpredictable" calculations for the mesh resolution transitions with plain compute tasks corresponding to regular grid subregions.

It continues to be advantageous to check if we can use a specialised compute kernel for a task. Compared to the kernel benchmarking or the regular grid calculations, the specialisation however does not yield the full factor of two in runtime improvements anymore. Finally, subcycling pays off performance-wisely, while local time stepping is slower than its adaptive algorithmic cousins.

Lessons learned 3. *Our manual task specialisation and scheduling allows us to preserve reasonable on-node scaling even if we use adaptive mesh refinement. It can also handle subcycling and local time stepping.*

6.4 Multi-node runs

Our techniques focus on the single node performance of PDE solvers with AMR and LTS. Yet, they have impact on MPI: The idea to add tasks on top of BSP implies that we also have tasks which can be used to overlap data exchange which typically is triggered after a rank's BSP sections have terminated. This overlap is the more powerful the more tasks we have relative to any urgent rank work and data exchange. Therefore, AMR and many MPI ranks challenge the scheme. At the same time, the balancing with tasks runs risk to suffer from NUMA effects.

We stick to setups from the previous sections. They have reached a strong scaling saturation already. As we use one BSP section per core, all experiments rely on the same geometric partitioning yet distribute the SFC segments differently between threads and ranks. Our benchmarks suggest that it is advantageous to stick to SPMD/BSP parallelism for regular grids with up to 32 MPI

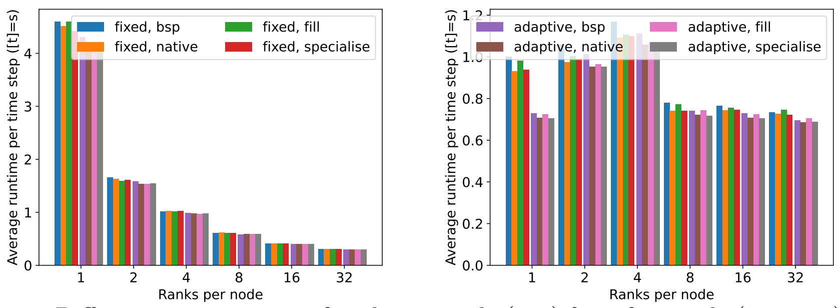

Fig. 4. Different arrangements of ranks-per-node (rpn) for a four-node (512 core) simulation run with the modified Euler equations over a regular (left) or adaptive (right) grid. The setups equal those of the previous sections.

ranks per node (Fig. 4). It is not clear if the good performance is due to NUMA effects or other machine characteristics. We however observe that the specialisation gains are diminished. For adaptive meshes, it is better to work with smaller rank counts per node.

Lessons learned 4. *Splitting up NUMA nodes into logically separated processors is usually advantageous from a performance point of view, yet reduces the number of non-urgent tasks which we can reschedule, optimise or batch. It hence makes the code performance less robust w.r.t. AMR.*

We emphasise that application task queues sitting on top of the tasking system have to be used carefully in an MPI context if they prevent tasks from overlapping with MPI data exchange. As soon as the problem size relative to the core count becomes larger again, we retain the qualitative statements of the single node studies and the specialisation starts to pay of again (not shown).

7 Conclusion and outlook

Task-based programming is not omnipresent in large-scale PDE solvers. Our data suggests that it can significantly improve the performance of classic BSP-style parallelism realised via parallel fors, if tasks can step in to compensate for BSP imbalances. We do not propose to use tasks as alternative to classic BSP parallelism but as a complementing technique. For the realisation of this concept, we avoid assembling any task graph, i.e. focus on the production of ready tasks, and add an additional task queue on top of the OpenMP task queue. It allows us to release work into the computation such that BSP imbalances are mitigated. We get the best of both worlds, i.e. BSP and tasking. While the tasking on top of BSP helps us to scale better, we also propose to distinguish tasks with global side-effects from low-priority tasks without side-effects and to handle the latter with specialised compute kernels that exploit their intrinsically higher internal concurrency level. In many cases, this doubles the code performance, even though we do not impose any constraints on the LTS or AMR or require any a priori analysis of the patch behaviour.

While a batched handling of assemblies of tasks without side-effects is an appealing concept, it does not robustly translate into an improved performance in our applications. Such a task fusion is conceptionally close to approaches that work with different p values for different blocks to keep the administration overhead low for grid regions where we can use regular submeshes, and it is similar to geometric clustering [25] or geometric blocking [27] as proposed by other authors. Different to these techniques or a flexible p selection, our approach does not hard-wire or pre-determine different p choices or make assumptions on the geometric distribution of blocks with the same p values or time stepping behaviour. While the task fusion does not translate into improved performance, the compiler feedback reports excellent vectorisation characteristics and hence suggests that the automatic batching is an interesting candidate for GPU offloading, e.g., as long as we batch early. It is future work to combine the technique with a performance model and to generate performance-portable code which exploits the batching similar to dynamic choices of p for different components of a supercomputer.

Our manual postponing of task spawns through the application task queue can be interpreted as an approach which implicitly prioritises BSP code segments over tasks. The hierarchy of parallelisation approaches in our code translates into a task prioritisation. It is hence reasonable to assume that carefully chosen task priorities would allow us to obtain similar runtime characteristics within an exclusively task-based programming model. Overall, the data suggests that a task graph assembly is not required as long as the prioritisation works. To meet our goal, our implementation replicates some OpenMP features such as queues on the user level. If future OpenMP generations provided options such as automatic task fusion or a careful prioritisation of tasks, it would be reasonable to eliminate our added tasking code layers to reduce the code complexity.

Next steps of our work comprise the analysis of the impact of the time stepping scheme on the quality of the solution. Our studies suggest that local time stepping is slower than global adaptive time stepping or subcycling. However, these measurements do not assess the impact of these schemes on the solution quality. In theory, local time stepping should introduce less numerical dissipation than the other schemes and hence yield better results. In this context, we also have to study more more sophisticated Riemann solvers [15]. This improved quality has to be assessed against the proposed runtime improvements. Finally, the generality of our concepts implies that a our approach might be advantageous for very flexible, task-based Lagrangian codes such as [19], too, as well as other solver components such as linear algebra subsystems.

References

1. Alcubierre, M., Allen, G., Bona, C., Fiske, D., Goodale, T., Guzmán, F., Hawke, I., Hawley, S., Husa, S., Koppitz, M., Lechner, C., Pollney, D., Rideout, D., Salgado,

M., Schnetter, E., Seidel, E., Shinkai, H.a., Shoemaker, D., Szilágyi, B., Takahashi, R., Winicour, J.: Towards standard testbeds for numerical relativity. Classical and Quantum Gravity **21**(2), 589–613 (2004)

2. Alic, D., Bona-Casas, C., Bona, C., Rezzolla, L., Palenzuela, C.: Conformal and covariant formulation of the Z4 system with constraint-violation damping. **85**(6), 064040 (2012)
3. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems **20**(3), 404–418 (2009)
4. Bader, M.: Space-filling Curves—An Introduction with Applications in Scientific Computing, Texts in Computational Science and Engineering, vol. 9. Springer (2013)
5. Bertschinger, E.: Self-similar secondary infall and accretion in an Einstein-de Sitter universe. **58**, 39–65 (1985)
6. Charrier, D., Hazelwood, B., Weinzierl, T.: Enclave tasking for dg methods on dynamically adaptive meshes. SIAM Journal on Scientific Computing **42**(3), C69–C96 (2020)
7. Daszuta, B., Zappa, F., Cook, W., Radice, D., Bernuzzi, S., Morozova, V.: GR-athena: Puncture evolutions on vertex-centered oct-tree adaptive mesh refinement. The Astrophysical Journal Supplement Series **257**(2), 25 (2021)
8. Demeshko, I., Diehl, P., Adelstein-Lelbach, B., Buch, R., Kaiser, H., Kale, L., (Sanjay)Khatami, Z., Koniges, A., Shirzad, S.: TBAA20: Task Based Algorithms and Applications. DOE Report LA-UR-21-20928 (2021)
9. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.C., Barkai, D., Berthou, J.Y., Boku, T., Braunschweig, B., Cappello, F., Chapman, B., Chi, X., Choudhary, A., Dosanjh, S., Dunning, T., Fiore, S., Geist, A., Gropp, B., Yelick, K.: The international exascale software project roadmap. IJHPCA **25**, 3–60 (02 2011)
10. Dubey, A., Almgren, A.S., Bell, J.B., Berzins, M., Brandt, S.R., Bryan, G., Colella, P., Graves, D.T., Lijewski, M., Löffler, F., O'Shea, B., Schnetter, E., van Straalen, B., Weide, K.: A survey of high level frameworks in block-structured adaptive mesh refinement packages. Journal of Parallel and Distributed Computing **74**(12), 3217–3227 (2016)
11. Dubey, A., Berzins, M., Burstedde, C., Norman, M.L., Unat, D., Wahib, M.: Structured adaptive mesh refinement adaptations to retain performance portability with increasing heterogeneity. Computing in Science Engineering **23**(05), 62–66 (2021)
12. Dumbser, M., Guercilena, F., Köppel, S., Rezzolla, L., Zanotti, O.: Conformal and covariant z4 formulation of the einstein equations: Strongly hyperbolic first-order reduction and solution with discontinuous galerkin schemes. Phys. Rev. D **97**, 084053 (2018)
13. Harlacher, D.F., Klimach, H., Roller, S., Siebert, C., Wolf, F.: Dynamic load balancing for unstructured meshes on space-filling curves. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS. pp. 1661–1669. IEEE Computer Society (2012)
14. Isaac, T., Burstedde, C., Ghattas, O.: Low-cost parallel algorithms for 2:1 octree balance. In: IEEE 26th International Parallel and Distributed Processing Symposium. pp. 426–437 (2012)
15. Kidder, E., Field, S., Foucart, F., Schnetter, E., Teukolsky, S., Bohn, A., Deppe, N., Diener, P., Hebert, F., Lippuner, J., Miller, J., Ott, C., Scheel, M., Vincent, T.: Spectre: A task-based discontinuous galerkin code for relativistic astrophysics. Journal of Computational Physics **335**, 84–114 (2017)

16. Meister, O., Rahnema, K., Bader, M.: Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells. ACM Trans. Math. Softw. **43**(3) (2016)
17. Peterson, B., Humphrey, A., Sunderland, D., Sutherland, J., Saad, T., Dasari, H., Berzins, M.: Automatic halo management for the uintah gpu-heterogeneous asynchronous many-task runtime. International Journal of Parallel Programming **47**(5–6) (2018)
18. Reinarz, A., Charrier, D., Bader, M., Bovard, L., Dumbser, M., Duru, K., Fambri, F., Gabriel, A.A., Gallard, J.M., Köppel, S., Krenz, L., Rannabauer, L., Rezzolla, L., Samfass, P., Tavelli, M., Weinzierl, T.: ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems. Computer Physics Communications **254**, 107251 (2020)
19. Schaller, M., Gonnet, P., Chalk, A., Draper, P.: Swift: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores. In: Proceedings of the Platform for Advanced Scientific Computing Conference. PASC '16, Association for Computing Machinery (2016)
20. Schulz, H., Gadeschi, G., Rudyy, O., Weinzierl, T.: Task inefficiency patterns for a wave equation solver. In: OpenMP: Enabling Massive Node-Level Parallelism. pp. 111–124 (2021)
21. Sundar, H., Sampath, R.S., Biros, G.: Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. SIAM J. Sci. Comput. **30**(5), 2675–2708 (2008)
22. Sundar, H., Ghattas, O.: A nested partitioning algorithm for adaptive meshes on heterogeneous clusters. In: Proceedings of the 29th ACM on International Conference on Supercomputing. p. 319–328. ICS '15 (2015)
23. Teyssier, R.: Cosmological hydrodynamics with adaptive mesh refinement—a new high resolution code called ramses (2002)
24. Treibig, J., Hager, G., Wellein, G.: LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In: Proceedings of the 2010 39th International Conference on Parallel Processing Workshops. pp. 207–216. ICPPW '10, IEEE Computer Society (2010)
25. Uphoff, C., Rettenberger, S., Bader, M., Madden, E., Ulrich, T., Wollherr, S., Gabriel, A.A.: Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17 (2017)
26. Weinzierl, T.: The peano software—parallel, automaton-based, dynamically adaptive grid traversals. ACM Transactions on Mathematical Software **45**(2), 14 (2019)
27. Weinzierl, T., Wittmann, R., Unterweger, K., Bader, M., Breuer, A., Rettenberger, S.: Hardware-aware block size tailoring on adaptive spacetree grids for shallow water waves pp. 57–64 (2014)
28. Zhang, H., Weinzierl, T., Schulz, H., Li, B.: 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 p. (submitted) (2022)
29. Zhang, W., Myers, A., Gott, K., Almgren, A., Bell, J.: Amrex: Block-structured adaptive mesh refinement for multiphysics applications. The International Journal of High Performance Computing Applications **35**(6), 508–526 (2021)