Received: 11 September 2019 | Revised: 14 April 2020 | Accepted: 18 May 2020

DOI: 10.1002/cpe.5916

WILEY

RESEARCH ARTICLE

Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement

Philipp Samfass[1] | **Tobias Weinzierl**[2] | **Dominic E. Charrier**[2] | **Michael Bader**[1]

[1]Department of Informatics, Technical University of Munich, Garching, Germany

[2]Computer Science, Durham University, Durham, Great Britain

Correspondence
Philipp Samfass, Department of Informatics, Technical University of Munich, Garching, Germany.
Email: samfass@in.tum.de

Funding information
EPSRC's Excalibur programme, Grant/Award Number: EP/V00154X/1 (ExaClaw); Horizon 2020 Framework Programme, Grant/Award Number: 671698 (ExaHyPE); Leibniz Supercomputing Centre, Grant/Award Number: pr48ma

Summary

Balancing the workload of sophisticated simulations is inherently difficult, since we have to balance both computational workload and memory footprint over meshes that can change any time or yield unpredictable cost per mesh entity, while modern supercomputers and their interconnects start to exhibit fluctuating performance. We propose a novel lightweight balancing technique for MPI+X to accompany traditional, prediction-based load balancing. It is a reactive diffusion approach that uses online measurements of MPI idle time to migrate tasks *temporarily* from overloaded to underemployed ranks. Tasks are deployed to ranks which otherwise would wait, processed with high priority, and made available to the overloaded ranks again. This migration is nonpersistent. Our approach hijacks idle time to do meaningful work and is totally nonblocking, asynchronous and distributed without a global data view. Tests with a seismic simulation code developed in the ExaHyPE engine uncover the method's potential. We found speed-ups of up to 2-3 for ill-balanced scenarios without logical modifications of the code base and show that the strategy is capable to react quickly to temporarily changing workload or node performance.

KEYWORDS

adaptive mesh refinement, MPI+X, reactive load balancing, task-based parallelism

1 | INTRODUCTION

Load balancing that decomposes work prior to a certain compute phase—a time step or iteration of an equation system solver—is doomed to underperform in many sophisticated simulation codes. There are multiple reasons for this: The clock frequency of processors changes over runtime,[1-3] the network speed is subject to noise due to other applications[4,5] or IO, and task-based multicore parallelization (MPI+X) tends to yield fluttering throughput due to effects of the memory hierarchy,[6] work stealing and nondeterminism in the MPI progression. While this list is not comprehensive, notably modern numerics drive the nonpredictability: They build atop of dynamic adaptive mesh refinement (AMR) that changes the mesh throughout a time step or mesh sweep,[7] combine different physical models,[7-9] or solve nonlinear equation systems with iterative solvers in substeps.[10] It becomes hard or even impossible to predict a step's computational load. As adjusting parallel partitions and respective data migration is often costly, many AMR codes consequently repartition only every 10th or 100th time step and tolerate certain load imbalances in-between.

We propose a novel, lightweight load redistribution scheme that acts on top of traditional load balancing. It, first, assumes that parts of the underlying simulation code are phrased in terms of many expensive tasks. It, second, assumes that good AMR codes manage to hide data exchange behind computations yet cannot keep all cores busy all the time. In every solver step, some cores on some ranks have to wait for MPI data to drop in. Our idea is to offload tasks from overbooked to waiting ranks to make these work productively rather than being idle.[11] The code plugs into the

MPI operations searching for the late sender pattern,[12] which yields a wait graph. Ranks that find out that they are critical to the walltime search for "optimal victims," that is, ranks that can take up further work without slowing down the overall computation, and then actively offload tasks to victim ranks. Third, we assume that neither load distribution nor imbalances change radically in-between algorithm steps. We therefore update the wait graph on-the-fly, using concepts from reinforcement learning,[13] and let the wait graph guide a diffusion of tasks to follow load alterations. Finally, we keep local copies of all offloaded tasks. This allows us to urgently recompute them if the temporary outsourcing does not come back with results fast enough. We overaggressively distribute tasks to build up a load balancing slack, and thus can react quickly to unforeseen load imbalances.

Task-based parallelization between MPI ranks is not new. The UIntah framework,[14,15] for example, uses a centralized data/task warehouse from which ranks are served. Tasks therefore are not tied to a particular rank and the ownership is (logically) with the warehouse. The Swift project[16] as another example phrases a whole SPH simulation in terms of tasks and applies graph partitioning to derive task decomposition and task migration patterns over the whole machine, that is, both shared and distributed memory domains. This is a wholistic, fine-granular, proactive load balancing approach. Charm++[17] features tasks that can be migrated and a runtime which tracks task dependencies in-between ranks dynamically. Dependencies thus pose no constraint on the task placement. Other task-based approaches such as HPX[18,19] feature task migration between different processes via a global address space. The AMR framework sam(oa)^2 finally introduces task stealing driven by the application[20] in-between bulk-synchronous processing. This list is not comprehensive.

An established alternative to a functional decomposition—typically realized through tasks—is a data, that is, domain decomposition. In an AMR world or situations where the load per cell is hard to predict, it has to be combined with frequent rebalancing. Efficacious load rebalancing strategies relying on space-filling curve cuts, diffusion processes or graph algorithms, for example, are known. Several properties determine whether they yield effective, that is, fast, code: First, an appropriate geometric cost model has to exist. If energy constraints compromise the compute nodes' performance,[1] if numerical schemes yield unpredictable workload per mesh entity, or if different physical models are applied to the same mesh set, deriving a cost model becomes nontrivial. Second, memory constrains the balancing. If very cheap and very expensive grid areas coexist, situations can arise where a load balancer cannot fit a big enough (cheap) subpartition to one resource. Third, data transfer cost constrains rebalancing. Even once a good domain decomposition is determined, the cost of moving toward this good decomposition from a given partitioning can outweight the gain if the partitioning remains advantageous only for few compute steps. Finally, spatial redistribution is an algorithmic step which synchronizes resources and stresses the communication subsystem. If many nodes rebalance at the same time, the communication subsystem is heavily used though there might have been periods of underutilization throughout compute phases.

While our approach starts from existing load balancing and takes up ideas and extends upon existing work, it introduces new capabilities: (i) It does not target load distribution per se but determines MPI waiting times to improve upon existing load balancing. This improvement is a reactive rather than a predictive add-on to load balancing and notably does not require an a priori cost model.[9] (ii) It is very fine-grained as it acts on the level of individual (compute-intense) tasks. Yet, no task dependencies are tracked. We work nonpersistently. Tasks are offloaded to other ranks, processed there, and the results are immediately sent back. (iii) It is a lightweight approach since the task migration is realized through a set of tasks itself. Therefore, we plug seamlessly into the tasking system and the overhead is small. We do not need a dedicated load balancing or MPI progression thread.[21] To our knowledge, this is the first approach that abandons the attempt to perfectly balance work in a predictive way but rather explicitly determines and hijacks MPI wait times to guide a lightweight task distribution while it remains reactive without the latency penalty introduced by classic task stealing, that is, it can react to quickly changing performance and load balancing characteristics.

Its properties render our approach promising for many applications which are already phrased in tasks. We assess it by means of an earthquake simulation benchmark. The underlying code base ExaHyPE[10,22] relies on an explicit time-stepping scheme, which works on dynamically adaptive meshes. The setup poses a challenge to our approach as it is not dominated by few compute-intense tasks.

We benchmark the reactive scheme against sole geometric domain decomposition and against a task distribution which is derived from chains-on-chains partitioning (CCP).[23]

Our article is organized as follows: We introduce the benchmark code in Section 2 before we phrase our vision (Section 3). Some terminology (Section 4) allows us to introduce a set of load balancing strategies in Section 5. This core contribution starts from a point-to-point diffusion approach which is augmented and accelerated by various techniques. In Section 6, we elaborate on the technical details of our implementation. Some experiments in Section 7 highlight the potential of the approach. We close the discussion with an interpretation of the scheme's characteristics (Section 8), before we identify further application areas of the proposed methodology plus future work in Section 9.

2 | A PARALLEL ADER-DG SEISMIC SOLVER ON ADAPTIVE MESHES

Our benchmark code implements an explicit high-order discontinuous Galerkin solver for the linear elastic wave equations, which may be written as (cf. Reference 24)

$$\frac{\delta \sigma}{\delta t} - E(\lambda, \mu) \cdot \nabla v = S_\sigma,$$
$$\frac{\delta v}{\delta t} - \frac{1}{\rho} \nabla \cdot \sigma = S_v,$$

with a velocity field v and a stress tensor σ in the first equation of the system. It results from Hooke's law and evolves both quantities through a stiffness tensor E depending on the Lamé constants λ and μ (ie, material parameters). The second equation describes Newton's second law. ρ here is the density of the material.

As simulation setup, we use the established layer over halfspace 1 (LOH.1) benchmark.[25] It mimics an earthquake via a simplified setting that assumes a point source in a cubic domain that consists of two material layers: a thin sediment layer (with slower wave speeds) over a rock layer (with higher wave speeds). LOH.1 is part of a widely used collection of benchmark scenarios to validate codes and compare results with other simulation software.

2.1 | ADER-DG: High-order discontinuous Galerkin

Our solver realizes an arbitrary high-order derivative discontinuous Galerkin (ADER-DG) method[26] on tree-structured Cartesian grids. It is implemented within the ExaHyPE engine to solve hyperbolic PDE systems.[10] In the following, we summarize the main computational steps of the scheme, whereas we describe full details of the scheme in previous work.[27]

ADER-DG is an explicit time-stepping scheme that decomposes each time step into three phases, thus computing $(\sigma, v)(t + \Delta T) = (\mathcal{C} \circ \mathcal{R} \circ \mathcal{P})(\sigma, v)(t)$. Each grid cell approximates the solution locally via a tensor product of polynomials of degree p (orthogonal polynomials constructed on Gauss-Legendre points), following a classic DG spectral element method approach.[28] In the *predictor* step $\mathcal{P}$, the algorithm first extrapolates the solution in time, ignoring the influence of neighboring cells and evolves (σ, v). This step follows the Cauchy-Kovaleskaya procedure.[8] The arising discontinuities in the predicted solution (σ, v) along the cell faces are next subject to a space-time Riemann solver $\mathcal{R}$. Finally, we bring the Riemann solution and the predicted value together, that is, correct ($\mathcal{C}$) the predicted value.

Our code discretizes our computational domain through a spacetree[29] and thus solves the problem on an adaptive Cartesian grid where the individual cells are cubes. Each cell may be transformed according to a curvi-linear transformation[28] to align to geometry features: Each cell carries a transformation matrix which fits it to the actual topology, allowing simulation of seismic wave propagation in complex topographies. For the LOH.1 benchmark, the transformation matrix is simple (but causes the same computational load), as it only aligns the material discontinuity in the LOH.1 geometry to our Cartesian grid. To reduce the discretization error further, we adaptively refine the mesh in the top sediment layer and around the point source.

2.2 | Parallel implementation of ADER-DG

Per time step, the $\mathcal{C} \circ \mathcal{R} \circ \mathcal{P}$ sequence of cell/face/cell operations is applied to the adaptive grid which is geometrically partitioned. We use a nonoverlapping domain decomposition where the Riemann tasks along the domain boundaries are computed redundantly by each adjacent rank.

The three ADER-DG phases translate into three types of tasks. Prediction tasks correspond to cells, Riemann tasks to faces, and correction tasks again to cells. Out of the three task types, the predictions $\mathcal{P}$ are the computationally dominant ones. They make up more than 97% of the runtime for our experiments with polynomial order $p = 7$. $p = 7$ is the order we observed the best time-to-solution per accuracy for our experiments. While they are expensive, they work per cell, that is, have well-defined memory needs, and they are totally independent of each other. Nevertheless, they decompose into two categories[7] of $\mathcal{P}$ tasks. One category are tasks/cells whose faces are adjacent to a resolution transition—that is, neighboring cells have a different resolution—or cells which are adjacent to a domain decomposition boundary.

The other category of $\mathcal{P}$ tasks is formed by all the remaining $\mathcal{P}$s. Each $\mathcal{P}$ task feeds its output into $2d$ Riemann tasks $\mathcal{R}$. Obviously, a Riemann task $\mathcal{R}$ depends on more than two prediction tasks if it corresponds to a face along a resolution transition. If an $\mathcal{R}$ task corresponds to a face along the MPI boundary, it furthermore requires input data running through the network. Our first category of prediction tasks—the same applies to corrections—all have dependencies with such "sophisticated" Riemann tasks. All Riemann tasks are computationally lightweight. After completion of all $2d$ Riemann solves which surround one cell, the time step's final correction task $\mathcal{C}$ is triggered. $\mathcal{C}$ is comparably cheap, too.

As we work with a task-based formalism, our code can work with fully nonblocking boundary data exchange. Upon the completion of a prediction job which is adjacent to a partition boundary, we send out the output immediately. As our steps are phrased in tasks, we then continue with further prediction tasks or postpone Riemann tasks which are not ready yet due to missing incoming data. This yields a classic MPI+X parallelization where the boundary exchanges do not synchronize the individual ranks (Figure 1).

The task formalism intermixes the three compute steps $\mathcal{C}, \mathcal{R}, \mathcal{P}$.[7] While $\mathcal{R}$ and $\mathcal{P}$ are very cheap and thus stress the memory system, the scheduler typically runs them parallel to some $\mathcal{C}$ tasks. The node's memory controllers consequently can deliver all data on time. This melange of different activities is interwoven with MPI data transfer in the background.[7] If bandwidth restrictions arise, they arise as bursts toward the end of each time step when the majority of tasks has finished and all MPI communication is triggered.[30] The code is MPI bandwidth-demanding yet not bandwidth-bound always.

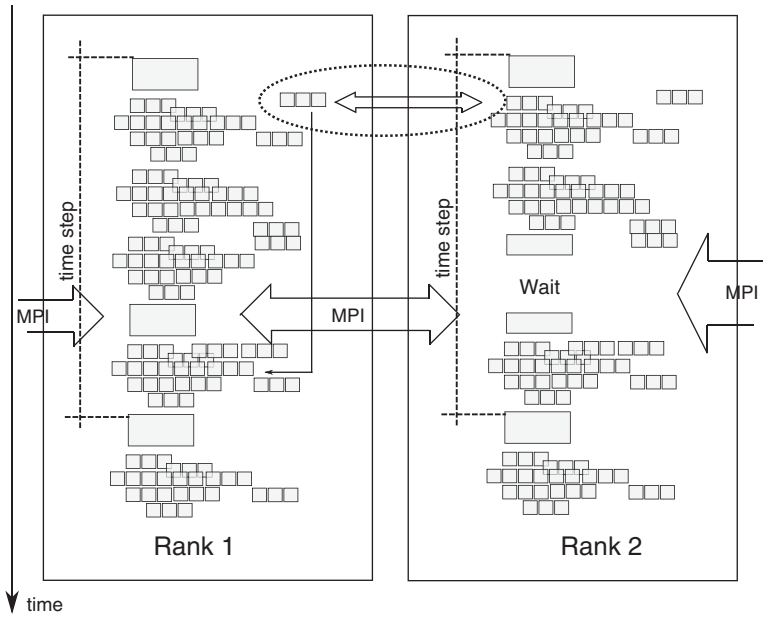


FIGURE 1 Schematic program execution: Ranks decompose into tasks. Some tasks' outcomes are required only late throughout the computation or even after the boundary data exchange or in the next time step. High bandwidth demands arise toward the end of the time step, that is, we are not consistently bandwidth-bound. Our scheme offloads nonurgent tasks to MPI ranks that tend to wait and immediately transfer the outcome back (tasks within the dotted circle belong to rank 1 but are computed on rank 2). The remote completion is almost hidden from the local rank's workflow

2.3 | Optimistic time stepping with weakened temporal and spatial constraints

Generic explicit time stepping for hyperbolic equations suffers from strong synchronization: The outcome of one time step has to be globally reduced, as we have to determine the admissible time step size from the CFL condition. This is an allreduce. Once we however assume that time step size is known or that our code can reliably estimate the evolution of the admissible time step size a priori—for our linear PDE with simplistic initial conditions, this holds trivially since the admissible time step size is invariant—we can eliminate the strict global synchronization of the ranks.[27] While a rank waits for an exchange of global information, incoming Riemann data or AMR information, it can already process $\mathcal{P}$ tasks of the subsequent time step. Performance analysis thus has to be done carefully: While a rank waits to complete its time step, and, hence, cannot logically kick off the next time step, it might still have work to do which logically belongs into this very next time step.

The $\mathcal{R}$ and $\mathcal{C}$ tasks have to run close to the memory. They are cheap and have outgoing dependencies into the compute-heavy $\mathcal{P}$ tasks, as they couple cells with each other or precede subsequent time steps. $\mathcal{P}$ tasks in contrast are candidates to be deployed to remote compute devices: they cause the primary computational load and they are typically not immediately time-critical, at least not in the moment they are spawned. Other predictions are in the task queue, and there is a high probability that further Riemann and correction steps of the previous time step still have to be processed. Furthermore, they are compact: They are atomic work units whose costly computations require input of limited memory footprint and yield output of limited footprint. Their arithmetic intensity is high while their input/output demands are small. We therefore call prediction tasks offloadable:

Definition 1. An **offloadable** task is a task with high arithmetic intensity and small input and output data, which is furthermore not time-critical in most situations, that is, is typically accompanied by many other ready tasks.

3 | METHODOLOGICAL VISION

We assume that the work in our code is already reasonably distributed via a distribution of data (grid cells and so on) to MPI ranks. We expect, however, that this distribution cannot lead to perfectly balanced execution times, because of impredictable computational load or fluctuations in system performance. In MPI+X codes, imbalance eventually manifests in MPI waits. We therefore determine approximate waiting times—the measured "wait" is reduced by the time a rank could spend on dangling tasks that are not critical for progress—and build a wait graph that allows us to determine bottleneck ranks. However, it is too late to react once ranks become idle, as we would essentially create further waiting times to move around tasks. Instead, we implement proactive task offloading in the sense that a *critical rank* (identified as being too slow) will offload tasks to under-employed *victim ranks* ahead of time (ie, *proactively*) and based on knowledge from previous time steps. Where this offloading is too ambitious, that is, results do not come back on time, the rank reduces offloading in subsequent compute steps and "urgently" recomputes the result itself. It is reactive. Our task offloading teams up with traditional data decomposition and migration, and helps to improve load balancing. It finally is hidden away from the code, that is, it is a lightweight extension.

We further exploit that our AMR code runs phases which are dominated by computations and phases where communication is critical. Despite bandwidth access peaks, we have bandwidth available in-between these peaks. *We propose to hijack MPI wait times and available bandwidth on "too fast" ranks to process tasks that are "stolen" from "too slow" ranks.*

4 | TERMINOLOGY

Our algorithm is constructed around simple terminology and a few definitions. Let $N^{(\text{ranks})}$ denote the number of MPI ranks. $0 \leq i, j < N^{(\text{ranks})}$ always holds for indices i, j. Each rank employs $N^{(\text{cores})}$ cores. They realize the time-stepping algorithm, that is, process $\mathcal{C}, \mathcal{R}, \mathcal{P}$. They notably also process all remote $\mathcal{P}$ tasks, that is, tasks sent in by another rank, processed locally, but then sent back. We denote $N_i^{(\text{tasks})}(t)$ to be the number of these tasks at a certain time t on rank i. As tasks are migrated, spawned throughout the time step, and completed, $N_i^{(\text{tasks})}(t)$ changes all the time. Finally let $t^{(\text{task})}$ be the time one core requires to complete one of the offloadable tasks. We assume they are atomic, that is, run exclusively on one core at a time. $t^{(\text{task})}$ quantifies the cost of $\mathcal{P}$. Sampling determines it introspectively: we use a moving average to determine $t^{(\text{task})}$, which implies that we assume all tasks on a given rank to be similarly costly on average. Yet, the time window of the moving average renders this cost model adaptive at run time, that is, it is reactive.

Per time step, our code exchanges boundary data with neighboring ranks as well as a global time step size. Our reactive load balancing plugs into these data exchanges. We found it sufficient to track the global exchange only, but the concept could be applied to the boundary exchange, too. It thus holds also in the absence of global synchronization.

Definition 2. Our code runs into situations where a rank i (logically) stops and cannot continue until a message from rank j arrives. Let the **waiting time** $t_{i,j}^{(\text{wait})}$ be the core time that elapses in-between.

In a BSP-type environment (bulk synchronous processing) where a rank forks threads, joins these again, and then finishes all data exchange, $t_{i,j}^{(\text{wait})}$ is a simple online measurement quantity: $t_{i,j}^{(\text{wait})} = N^{(\text{cores})}(T_{i,j}^{(\text{start})} - T_{i,j}^{(\text{end})})$, where $T_{i,j}^{(\text{start})}$ is the time stamp when rank i receives the kick-off message of the subsequent time step from rank j and $T_{i,j}^{(\text{end})}$ is the time stamp when data exchange between i and j ends (these are typically sends). $t_{i,j}^{(\text{wait})}$ sums up all core wait times (which are equal) and thus scales with $N^{(\text{cores})}$.

In an asynchronous task environment tasks of a time step n that are not critical to the progress of the rank may overlap with computations of time step $n+1$.

We therefore reduce the wait time $t_{i,j}^{(\text{wait})}$ by an additional term:

$$t_{i,j}^{(\text{wait})} = \max\left(0, N^{(\text{cores})}\left(T_{i,j}^{(\text{start})} - T_{i,j}^{(\text{end})}\right) - N_i^{(\text{tasks})} t^{(\text{task})}\right). \tag{1}$$

$N_i^{(\text{tasks})} t^{(\text{task})}$ quantifies how much of the wait time can be spent productively on handling ready tasks. It is a crude estimate as $N_i^{(\text{tasks})}$ might change dramatically throughout this time. Consequently, we use the max function to avoid negative wait times.

Definition 3. Rank i is called a **critical rank** if $\forall j : t_{i,j}^{(\text{wait})} = 0$ and $\exists j : t_{j,i}^{(\text{wait})} > 0$.

A critical rank is a rank that does not wait for any other rank but delays at least another one. While there may be more than one critical rank, we usually identify the most critical one to offload tasks from it to underloaded victim ranks:

Definition 4. Let $t_{max}^{(\text{wait})} = \max_{i,j} t_{i,j}^{(\text{wait})}$. A rank i is an **optimal victim** if $\nexists j : t_{j,i}^{(\text{wait})} > 0$ and $\exists j : t_{i,j}^{(\text{wait})} = t_{\max}^{(\text{wait})}$.

An optimal victim is the rank in the system that could take up the biggest chunk of further work without decreasing the performance, since it idles the longest. Our goal is to make critical ranks deploy more and more tasks to optimal victims until they cease to be critical. For this, we introduce a quantity $N_{i,j}^{(\text{offload})}$ per rank which clarifies how many tasks from rank i should be deployed to rank j. Rank i then plugs into the task spawn mechanism. We outsource the first $N_{i,j}^{(\text{offload})}$ offloadable tasks that become ready throughout a time step to rank j. In a task-based environment the "first" is to be read weakly, as the runtime might reorder them.

As we work in a distributed environment with changing meshes, nonconstant numeric cost, and hardware noise, this type of nonpersistent load balancing can fail:

Definition 5. An **emergency** arises for rank j if $\exists i : N_{i,j}^{(\text{offload})} > 0$ and $t_{i,j}^{(\text{wait})} > 0$.

Emergency means that a rank both deploys data to a victim rank and is delayed by this very rank. This may happen when the victim rank is overloaded, if we suffer from network congestion or if too many messages (remote tasks) stress the MPI subsystem such that results are not sent back fast enough to the deploying rank. As soon as we spot such an emergency, we add a rank to a black list.

Definition 6. The **blacklist** is the set of ranks that may not take up more work. We hold one blacklist per rank.

Our terminology circumscribes a greedy graph optimization algorithm. We establish a *wait graph* over all ranks. $t_{i,j}^{(\text{wait})}$ serves as edge weight in this directed graph. If we mask out zero weights, the graph is sparse. The critical rank is the last rank along a critical path through the set of ranks. The "last" edge points to the critical rank. Multiple critical ranks may exist. Our goal is to remove the head from the critical path and then to continue iteratively.

To achieve this goal, we label those ranks in the graph which are origins of wait paths with the biggest wait time as optimal victims. They can take up further work without slowing down the overall computation. The determined numbers of task offloads $N_{i,j}^{(\text{offload})}$ establish a *task distribution graph* on top of our rank vertices. It connects sinks of the wait graph with sources of critical paths. Finally, we allow ranks to compute local task outcomes even though they tried to offload work:

Definition 7. An **urgent local compute** is the computation of a task despite the fact that this task has been given to another rank. If we urgently recompute a task outcome, we neglect this task's results when they eventually drop in.

5 | LIGHTWEIGHT REACTIVE LOAD BALANCING

Once the MPI wait times are identified, each rank i maintains statistics of $N_i^{(\text{tasks})}(t)$. It measures all $t_{i,j}^{(\text{wait})}$ and it samples execution times to determine $t^{(\text{task})}$. Furthermore each rank has a blacklist of ranks that return remote tasks too slowly. All statistics are sampled over time spans through

$$\bar{x} = \frac{\sum_{l=0}^{S} \left(\omega^{(\text{avg})}\right)^l x_l}{\sum_{l=0}^{S} \left(\omega^{(\text{avg})}\right)^l}, \quad \text{with a fixed } \omega^{(\text{avg})} \in (0, 1]. \tag{2}$$

$x_0, \ldots, x_S$ are the measurements from the $S+1$ most recent time steps (x being a placeholder for our quantities of interest). We drop older measurements as further quantities alter the moving average by less than 10% for $\omega^{(\text{avg})} \approx 0.9$.

Our global statistics allow us to introduce various algorithms to determine $N_{i,j}^{(\text{offload})}$, that is, how many tasks each rank i has to deploy to rank j. We update $N_{i,j}^{(\text{offload})}$ prior to each time step with the most recent statistics at hand. From hereon, newly spawned offloadable tasks on i can be offloaded to another rank j as long as they haven't exceeded our quota $N_{i,j}^{(\text{offload})}$. This definition implies that we never delegate stolen tasks further, that is, only tasks produced locally are "stolen" by another rank.

In order to improve parallel performance in the presence of critical ranks, we propose different strategies.

5.1 | Reactive load balancing

Each rank can determine its optimal number of tasks $N_{i,j}^{(\text{opt})}$ that it has to deploy to other ranks from the global data view (Algorithm 1). The iterative approach identifies the unique critical rank, computes how much it could "fill up" the optimal victim, adopts the load distribution, and then waits for the next time step's measurements.

Algorithm 1. Blueprint of reactive load balancing.

```
function REACTIVELB(rank i)
    ∀k ≠ i exchange t^(wait) (nonblocking allgather)
    Compute critical rank m
    if m = i then
        Compute optimal victim n
        N_{i,n}^(opt) ← 0.5 t_max^(wait) / t^(task)
    end if
end function
```

The algorithm's use of the term optimal in $N^{(\text{opt})}$ is misleading for several reasons: First, it is a backward-looking optimum which derives an optimal task distribution for the passed time step. With AMR, the grid however might change in the present step. Second, it relies on a weak consistency model for its input quantities, as we use nonblocking allgather. Some data used in the computation thus might be outdated. Third, the quantities themselves rely on $N_i^{(\text{tasks})}(t)$ which is a snapshot of the local runtime's state. Fourth, the formula is based upon a real-time measurement of $t^{(\text{task})}$ which we determine through a weighted averaging over multiple probes. If a core downclocks due to high energy consumptions[1] or failures, this does not immediately reflect in the timings. Finally, though our formalism sticks to unique critical workers and optimal victims, it can happen that the asynchronous balancing makes multiple ranks consider themselves to be critical.

5.2 | Diffusion

There is limited sense in using $N^{(\text{opt})}_{i,j}$ as it is only a guideline which tends to rebalance aggressively. It grabs one victim rank's MPI time completely in one rush. We therefore introduce per rank a relaxation factor $0.1 \leq \omega^{(\text{diff})}_i \leq 1$ and determine a task distribution from the optimal distribution plus the current state:

$$N_{i,j}(k+1) = \omega^{(\text{diff})}_i N^{(\text{opt})}_{i,j}(k) + (1 - \omega^{(\text{diff})}_i) N_{i,j}(k).$$

$\omega^{(\text{diff})} \approx 1$ makes the diffusion adopt the "optimal" task distribution quickly, while a small $\omega^{(\text{diff})}$ yields a moving average. The actual distribution is incrementally fitted to the optimal distribution.

We may consider our overall optimization problem to be strongly nonconvex and subject to fluctuations. To reduce the risk to run into local minima with small $\omega^{(\text{diff})}$, but also to reduce the risk to introduce massive distribution fluctuations, we increment $\omega^{(\text{diff})} \leftarrow \min(\omega^{(\text{diff})} + 0.1, 1)$, if

$$\forall i : \frac{\sum_j |N^{(\text{opt})}_{i,j}(k+1) - N^{(\text{offload})}_{i,j}(k)|}{\sum_j |N^{(\text{opt})}_{i,j}(k) - N^{(\text{offload})}_{i,j}(k-1)|} \geq \omega^{(\text{reinf})}. \tag{3}$$

Otherwise, $\omega^{(\text{diff})} \leftarrow \max(0.9\omega^{(\text{diff})}, 0.1)$. $\omega^{(\text{reinf})} \in (0, 1]$ is fixed.

While a decrease of $\omega^{(\text{diff})}$ by 10% ensures that our diffusion updates usually become smaller and smaller, we increase the relaxation if two subsequent iterations drag the update with a certain intensity. The latter typically happens if ranks enter the blacklist:

5.3 | Blacklisting

Our load balancing strategies can run into situations where they overbook ranks and thus slow down the overall computation—despite the damping of the updates through $\omega^{(\text{diff})}$. The paragraph following Definition 5 enlists reasons for this and introduces blacklists that accommodate this problem.

Whenever a victim rank does not deliver the result of a stolen task back fast enough, the origin rank identifies this emergency and adds the victim rank to its local blacklist. Blacklists are subject to our nonblocking all-gather communication and thus shared globally. The update of the local load distribution sets $N^{(\text{opt})} = 0$ for any blacklisted communication partner. In a diffusive world, this triggers a gradual retreat from overbooked ranks.

We found it valuable to use an annotated blacklist set where each entry holds a weight. As long as emergencies arise for a particular rank, its blacklist value is incremented by one. After each rebalancing round, we decrement the weight by 10%. Blacklist entries with a weight below 0.5 are eventually removed from the blacklist. We avoid oscillations: If a rank has entered the blacklist, it remains on this list for a while to avoid that it is immediately rebooked after enough tasks have been retreated.

5.4 | (Reduced) Chains-on-chains partitioning

Diffusion yields a slow process. This is especially true at startup if we start from an ill-suited domain decomposition. Furthermore, no iterative technique is safe from running into local minima. It is hence reasonable to benchmark against an "optimal" task distribution that is computed for a given grid setup. The term optimal however is to be chosen carefully, as any precomputation relies on an a-priori cost model which can only approximate the actual machine behavior. We use a uniform cost model for $\mathcal{P}$ which neglects data transfer cost.

CCP[23] is one approach to determine good task distributions. It can be defined as partitioning of a 1D chain of $\sum_i N^{(\text{tasks})}_i$ tasks into $N^{(\text{ranks})}$ partitions such that the bottleneck load (maximum load assigned to a rank) is minimized. With uniform cost per task, the CCP problem reduces to a much simpler problem: we only need to "cut the chain" of tasks into $N^{(\text{ranks})}$ equally sized pieces, that is, the bottleneck load will then be equal to the average load over all ranks (±1 task). The number of tasks per rank is known after the initial mesh was built on every rank. We use a single collective allgather step to distribute this information among all MPI ranks. Every rank then solves the reduced CCP problem using a simple search algorithm. This results in a new unique partitioning that defines how many tasks every rank needs to give to other ranks such that the new load on every rank is rendered equal to the average load.

5.5 | Urgent local compute

We offload solely ready tasks. If results of offloaded tasks come back too late, blacklisting becomes active. This is a reactive strategy to accommodate unexpected performance breakdowns. However, it remains a proactive mitigation and does not moderate the immediate performance penalty

arising from a lack of task results. If a rank experiences a performance drop, blacklisting and task reassignment react to this change of performance two or three time steps later.

We therefore propose an extension of our scheme that tackles sudden performance drops: Whenever our algorithm offloads a task to another rank, a local copy of this very task is stored and kept on the origin rank as well. We call this task a *local recompute task*. If we run into an emergency, we continue to blacklist. Instead of an idling wait, we however compute the outcome of the task we are waiting for locally. We handle the local recompute task. We eventually can proceed even though task results still have not come back. The underlying offloaded task is internally marked as recomputed. When its result comes back, we throw it away, as we have already determined the task outcome locally.

Urgent local recomputes are accompanied by some local overhead, as we have to realize some additional bookkeeping. Its most important implication is that it changes the blacklisting behavior: Whenever a rank waits for offloaded task results from ranks i and j, a realization without urgent recomputes blacklists first i, waits for the result of i and then checks j. This gives j more time to get its results back. With urgent recomputes, there is a higher probability that both i and j are blacklisted. We found it thus advantageous to explicitly mask out such situations, that is, to stop any blacklisting after one emergency until the underlying emergency's rank has finally got its results back.

6 | IMPLEMENTATION

The success of our reactive, lightweight load balancing hinges upon an efficient, low-overhead realization. In particular, we rely on fast task migration for an irregular, a-priori unknown dynamic communication pattern. We found that prioritized task processing and full overlap of task communication are essential. The latter requires dedicated attention from a technical standpoint, as sufficient "progression" of MPI messages needs to be ensured.

Our implementation is based on Intel's Threading Building Blocks (TBB)[31] which we extended by a custom priority mechanism: We employ as many real TBB tasks as we have cores per rank. These TBB tasks process (consume) our own, logical tasks managed through TBB's priority queue. We found this solution to outperform the native TBB priorities.

6.1 | Task lifecycle and decision making

Our runtime distinguishes three types of tasks: High priority tasks, low priority tasks and offloadable tasks. Low priority is the default. The offloading hooks into the actual creation of offloadable tasks.

Algorithm 2. Spawn process of a ready task on rank i. At the start of each time step $\hat{N}_{i,j}^{(\text{offload})} \leftarrow N_{i,j}^{(\text{offload})}$.

```
function SPAWNTASK(rank i, task x)
    notStarved = N^(tasks) > C                          ▷ Avoid rank starvation
    if notStarved ∧ canOffload(x) then
        j = pickarg_k{N̂_{i,k} > 0}                      ▷ Round robin
        if j ≠ ⊥ then
            N̂_{i,j}^(offload) ← N̂_{i,j}^(offload) − 1   ▷ Atomic
            Send x to rank j
        end if
    else
        Enqueue x with low priority
    end if
end function
```

The hook makes the decision whether a task is enqueued locally or can be offloaded. For this, it combines three criteria (Algorithm 2): The task has to be offloadable, there have to be more than C tasks in the local task queue, and there has to be a victim rank. We store an atomic counter for each target rank j of i. It counts how many tasks still might be offloaded to rank j. It is updated per time step by the load balancing. If a task is given away, the respective counter is decremented. As we may need to offload tasks to multiple victim ranks, victims are selected in a round-robin fashion. Round-robin ensures that victim ranks can start to process offloaded tasks as soon as possible.

We exploit that each task is ready when it is spawned. For codes with task dependencies, the offload decision would need to hook into the transition of a task into ready. Giving away tasks too aggressively can lead to starvation of rank-local task consumers. We face a classical consumer-producer challenge: The code spawns tasks only at a certain speed and puts them into the job queue. Besides the limited speed, not all tasks are offloadable. Hence, if we give away tasks too aggressively to other ranks—which act as additional consumers—the task job queue may run

out of tasks for local processing. To avoid this, we only offload a task if enough tasks remain available to keep the local task consumers busy. This guarantees optimal utilization of both local and remote resources.

Tasks that are offloaded logically split up into two tasks: While the actual task is sent away and computed remotely, we logically insert a single receive task for all offloaded tasks. The runtime will poll this receive task as part of the standard task processing. Once a remote task starts to send its results back, the receive task finalizes the corresponding MPI receives and cleans up all data structures. The task offloading itself is not visible to the application.

To ensure that offloaded tasks are sent back early, that is, to ensure that we make optimal use of the network, we issue offloaded tasks with high priority. They are thus computed prior to local tasks.

6.2 | Creating the reactive communication graph

Predictive load balancing algorithms, such as CCP, use a dedicated synchronization step where load balancing meta-information is exchanged. As a result, all communication partners are known prior to the actual computation and `MPI_Irecv`s can be posted at the time of the load migration. In our reactive scheme, we do not explicitly exchange meta-information, the communication pattern changes frequently and the round robin task distribution makes it impossible to predict the exact data flow as well as the number of messages to be transferred. Finally, tasks are to be sent out as soon as possible, that is, we may not aggregate tasks.

Our algorithm resembles a one-sided data exchange model where many small tasks are "put" to another rank and have to "trickle through" while the numerical algorithm is running. Without a mutual a-priori agreement on the communication pattern and the size of receive windows, that is, the data cardinality, we however issue one asynchronous data send per task that is to be offloaded, and we use `MPI_Iprobe` to detect tasks that are to be received.

This yields many small nonblocking data transfers. Their efficient realization, that is, the quick establishment of data flows—we may assume that they are large enough to prohibit eager buffering—is very important as we have to release critical ranks from work. We make an additional task realize the `MPI_Iprobe` pickups. It polls MPI, establishes incoming data connections, that is, launches receives, and eventually reschedules itself after all the other ready tasks. Therefore, the task's mean time between activation automatically depends on the load: The longer the task queue the longer it will take until the probing is executed again. In phases of high computational load, CPU time is mostly dedicated to computation. In phases of low computational load, in particular whenever a rank is underloaded and thus a potential victim, rescheduling ensure that tasks are received quickly. We busy-poll MPI. Once a remote task completes, its host rank, that is, the victim, triggers a send back. It is another nonblocking send which is eventually picked up by the probes on the task's origin.

6.3 | MPI progression

Our code stores all pending sends and receives, that is, the `MPI_Request` handles, in a central broker ("request manager"). They are held FIFO. A central difficulty with many nonblocking MPI messages and a dynamic exchange pattern, however, is progressing messages in the background. Issuing solely `MPI_Isend` does not ensure that the actual message transfer occurs fully in the background without any further CPU involvement.[21] We cannot be sure that MPI makes sufficient progress.

One possible remedy is to sacrifice a thread for asynchronous MPI progression. However, neither do all MPI implementations support dedicated progression threads, nor did we succeed to use them robustly on our test system, nor are we eager to sacrifice a whole thread. Even if it is pinned to a hyperthread, a progression thread tends to pollute the runtime characteristics and caches.

We therefore implemented a `progress` task (similar to Reference 32) which uses `MPI_Testsome` on the request manager's request queue to make progress on outstanding MPI requests. In line with the polling, the task is started prior to the first time step and reschedules itself. Its rescheduling policy is different to the polling:

Requeuing at the end of the ready queue turns out to be insufficient when a critical rank sends away tasks aggressively to victim ranks. A critical rank is per definition overloaded, that is, has a long task queue. Too little investments into MPI progress yield late receives on the victim side. The progression task therefore forks an additional very high priority task if there are outstanding send requests. This task is terminated once no more outstanding send requests are remaining. On the receiving side, that is, on an optimal victim rank, a very high priority progression task is spawned if there are outstanding receive requests. The latter is terminated once the receiver's set of active senders is empty.

Packing all tasks outsourced to one victim rank into one big message[9] could mitigate the need for aggressive, manual MPI progression, since fewer (larger) messages are exchanged. It can however delay the outsourcing on the sender side: If nonmigratable tasks "suddenly" are inserted into the local task graph, the assembly of a particular set of outsourced tasks can be significantly delayed. Such situations arise, if a mesh traversal has to realize dynamic adaptive mesh refinement early throughout a time step within its local domain. On the receiver/victim side, a collection of the task

outcomes that are to be returned can imply that the outsourcing rank recomputes outsourced data locally even though it would have been available on time. We reduce the communication to computation overlap.

6.4 | Data calibration

Our reactive load balancing algorithm relies on online performance measurements which is distributed using nonblocking collective communication (`MPI_Iallgather`). With real-time stamps, it is clear that an effective zero wait time does not manifest in a zero time span. We thus determine a threshold $t_{\min} = 0.95 \cdot \min_{i,j} t_{i,j}^{(\text{wait})} + 0.05 \cdot \max_{i,j} t_{i,j}^{(\text{wait})}$ for each rank, and drop all times below $t_{\min}$.

Nevertheless, some data remain biased: The wait time as defined in (1) notably suffers from snapshotting effects in MPI. Before we bookmark $N_i^{(\text{tasks})}(t)$, we run an additional instance of our polling task. It otherwise might happen that (1) assumes that no tasks were there even though they roam in MPI. This would eventually yield wrong timings and input into our algorithm.

We finally point out that (1) is a very idealised machine model: Our formula does anticipate that pending tasks can be performed while we wait for incoming MPI messages and cores thus do not idle, but the formula does not distinguish where these remaining ready tasks come from. If no tasks are stolen, it is reasonable to assume a fixed cost $t^{(\text{task})}$ for a homogeneous set of pending tasks. If some of these tasks however are stolen tasks, their cost is higher, as we eventually have to send these tasks back. The vanilla version of (1) underestimates the local load, thus yields too high wait times, and eventually traps the reactive load balancing in an overbooking of victim ranks. It is therefore reasonable to reduce the local load further by a penalty which correlates linearly to the number of received tasks (which is encoded in our request manager).

7 | RESULTS

We benchmark our code on SuperMUC phase 2 and SuperMUC-NG at the Leibniz Supercomputing Centre (LRZ). Each of phase 2's two-socket nodes contains two 14-core Intel Xeon E5-2687 v3 (Haswell) CPUs. Throughout the experiments, they have been clocked at 2.3 GHz. Infiniband FDR14 connects the individual nodes with a nonblocking pruned 4:1 tree. SuperMUC-NG hosts 2×24 cores of the Intel Xeon 8174 (Skylake) generation per node, which are clocked at 2.3 GHz and are connected through Intel Omni-Path. All shared memory parallelization relies on Intel's TBB[31] while Intel's C++ compiler translated all codes. We use the 2018 generation of both tools on SuperMUC phase 2 and the 2019 generation of both tools on SuperMUC-NG.

Benchmarking with the baseline code reveals that we achieve a high shared memory efficiency on one socket (Figure 2) for a regular grid. Performance deteriorates once we exceed 14 cores as NUMA effects kick in Reference 1.

We therefore typically run multiple-of-two ranks per node. For adaptive grids, our scalability is slightly worse. Our task parallelization exposes some freedom to move tasks around. With AMR, the task cost are more heterogeneous as interpolation and restriction tasks enter the system, too. This causes the slightly inferior scalability on one socket and an amplification of the NUMA effects. AMR's better cost per degree of freedom here is classic weak scaling effect. AMR management overhead is amortized by the higher degree of freedom count. The present setup uses a static adaptive mesh, that is, a mesh where we use a regular grid and then add one more level to some mesh cells. The results for dynamically adaptive grids do not differ qualitatively.[7]

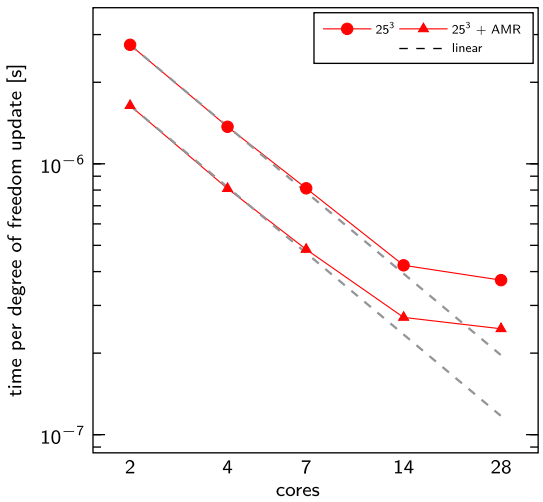


FIGURE 2 Shared memory parallel efficiency of our baseline code on one node without any offloading. We start from a $25 \times 25 \times 25$ grid (15 626 cells) and then add one additional level of (static) AMR (58 525 cells). AMR, adaptive mesh refinement

7.1 | Task and wait graph characterization

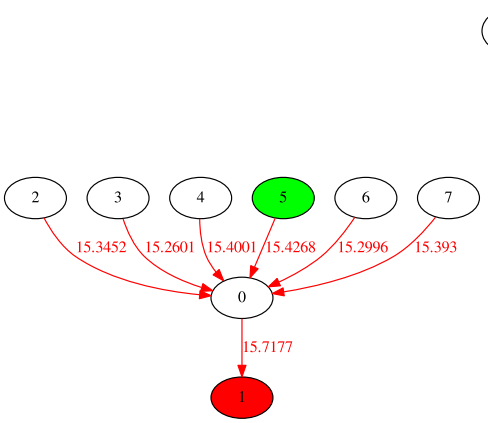
We kick off our work distribution experiments with a showcase to illustrate the algorithms' behavior for stationary grids. No urgent recomputes are employed so far. The setup uses a $25 \times 25 \times 25$ grid (leading to a problem size of 72 Mio degrees of freedom) on a single Haswell node hosting eight MPI ranks. The load decomposition with eight ranks has to be imbalanced. We make the code dump all task outsourcing and wait time information and use these data to extract the graphs underlying our algorithmic mindset.

The graphs (Figure 3) reveal that there is an overbooked rank 1 which delays our main time-stepping loop running on rank 0. As rank 0 has to wait for rank 1, it in turn throttles the remaining six ranks that wait for a kick-off of the next time step. Such knock-on effects explain that our wait graphs will always resemble tree or forest graphs. It is reasonable to address the tails of the wait graphs to bring down the runtime iteratively.

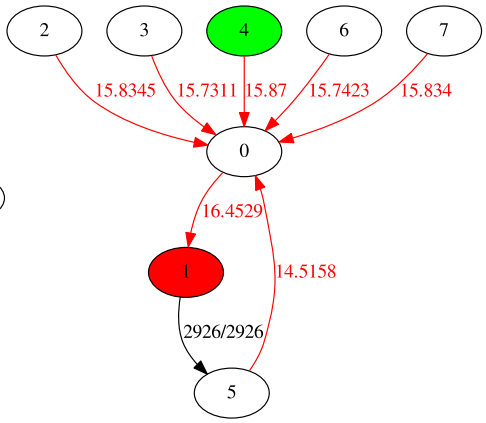
The diffusive scheme, here ran with fixed $\omega^{(\text{diff})} = 1$, starts to gradually outsource tasks from the overbooked rank to all other ranks. The optimal victim role is passed on from one rank to the other (compare Figure 3A,B) until all possible victims have been selected. The load distribution then stabilizes and is subsequently only altered by a small number of tasks. CCP yields a very similar task distribution scheme for the present setup. Our reactive diffusion thus is consistent in a numerical sense. Overall, CCP seems to balance more evenly across the ranks 2 to 7, while the number of offloaded tasks per rank is lower.

The task graphs' black labels lead to a further interesting observation. Both balancing schemes derive a maximum number of tasks $N^{(\text{opt})}$ per rank which determines how many of these tasks can be given away. As tasks however first have to be created—an effect that amplifies for AMR where the task graph is unknown prior to the time step and dynamic refinement and coarsening can delay the creation of some tasks as the grid first has to be adopted—not all ranks fully exploit their task quota.

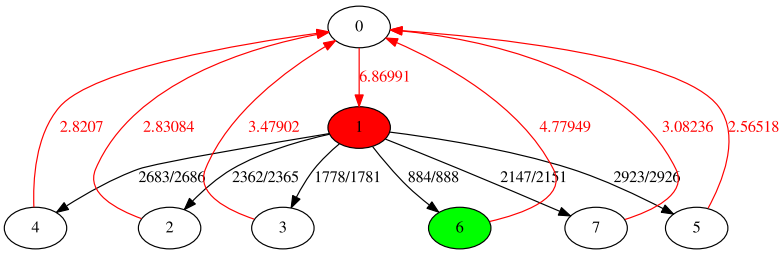
We continue to investigate this effect in further experiments where we use 16 MPI ranks distributed to 16 nodes and parametrize the number of cores available to each rank. We see ranks deploying the fewer tasks the more cores they have locally available (Figure 4). For many codes deployed



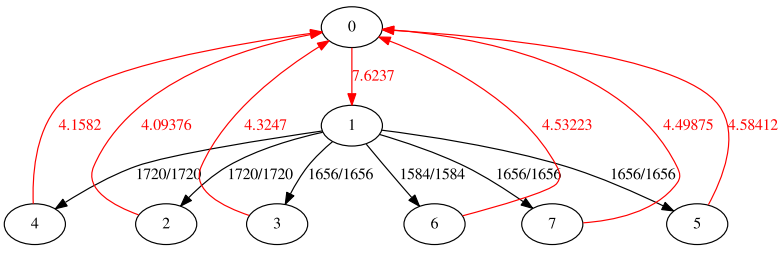
(A) Diffusion after 7 time steps.



(B) Diffusion after 10 time steps.



(C) Diffusion after 200 time steps.



(D) CCP after 200 time steps.

FIGURE 3 Wait and task distribution for the diffusive algorithm (A-C) and our task offloading using only the CCP guess (D) for eight ranks. The critical rank is highlighted in red, the optimal victim in green. Red edges are wait times in seconds, black edges illustrate task offloading. The two given task numbers denote offloaded tasks vs maximum tasks a rank would have been allowed to offload. CCP, chains-on-chains partitioning

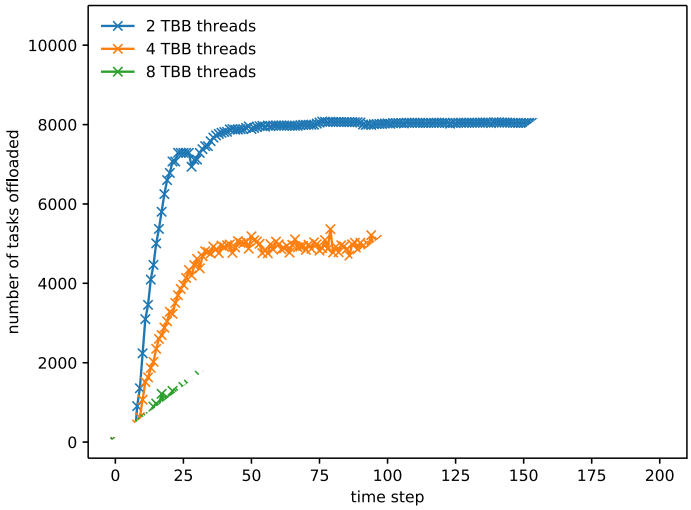


FIGURE 4 Run with 16 ranks (one MPI rank per node) where we vary the number of threads available to each rank (SuperMUC phase 2)

to multisocket systems, it is reasonable to use more than one rank per node. This reduces NUMA effects.[1] Our reactive load balancing supports such a strategy. Otherwise, too many local cores have to be kept busy. These two technical advocates for multiple ranks per node finally are supported by the observation that more ranks give the domain decomposition more degrees of freedom how to distribute the mesh.

7.2 | Comparison of baseline algorithms for an almost balanced mesh

We continue with a comparison of our two lightweight redistribution algorithms, CCP and reactive diffusion, to the baseline code performance. Again, the grid is fixed to $25 \times 25 \times 25$. We employ 28 ranks in total. Due to the dominance of the $\mathcal{P}$ tasks, it is reasonable to assess the balancing quality in terms of the distribution of the space-time predictors: The lightest eight ranks host 512 of these tasks, while the heaviest rank hosts 729 $\mathcal{P}$s.

The measurements reveal (Figure 5) how hard it is to balance and tune our baseline code—a property we consider to be prototypical for modern, task-based simulation codes: The runtimes per time step scatter significantly even though this is a regular grid setup without AMR. Closer inspection uncovers that the runtime does not randomly fluctuate but exhibits an oscillation-type pattern. Our code yields a task graph where individual tasks are optimistic. It is thus possible to bring tasks forward and to compute them in the (logically) previous iteration already. This leads to oscillating behavior: One iterate finishes quickly. Tasks of the follow-up time step are set ready but not processed before the iteration reports "done" to the other ranks and completes its boundary data exchange. The subsequent iterate now has to process all of its tasks. At the same time, its task processing already spawns tasks of the subsequent iteration. Some of them are processed straight away as they sit in the ready queue. Compared with the previous time step, the present time step thus lasts longer. The fact that it already computes (some of) the tasks of the next iteration in turn makes the subsequent iterate finish fast again. We end up with oscillations.

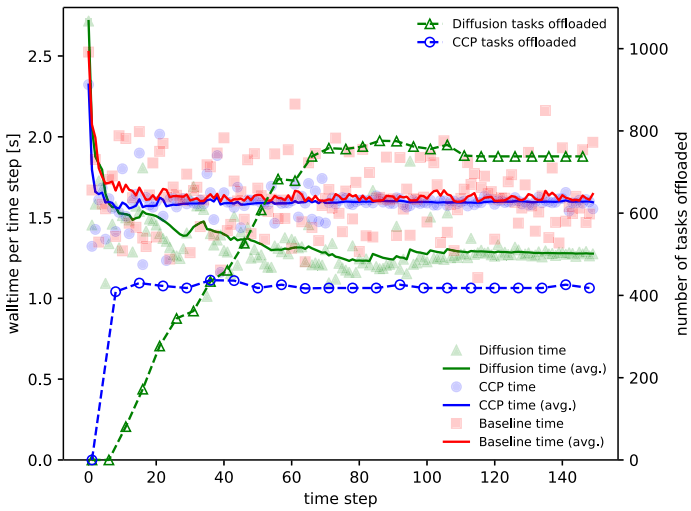


FIGURE 5 Comparison of CCP and diffusion with $\omega^{(\text{diff})} = 0.5$ to the baseline runtime. Per test, we present both the number of tasks that are offloaded and the runtime as gliding average as phrased by (2) (SuperMUC phase 2). CCP, chains-on-chains partitioning

Both of our balancing techniques reduce the oscillations. While is reduces the noise/scattering, CCP yields a time-averaged time per step which is hardly better than the runtime of the baseline code. CCP's quasi-static "re"-balancing or "on-top"-balancing fails to improve the performance. The reason is that CCP is agnostic of the real-time behavior of the multithreaded code. A positive insight is that the offloading's overhead is small, as we do not loose performance with CCP. CCP is not slower than the baseline.

The diffusive approach clearly outperforms CCP. It reduces the runtime almost monotonically and, once converged, brings the runtime per time step down from approximately 1.6 s per step to around 1.2 s. The measurements support our decision to use work with (5) for measurements, and we observe that the diffusion, anticipating real hardware behavior, calls for convergence acceleration techniques. The improvement of the runtime is slow. Most importantly, the diffusion is faster than the static-cost model of CCP. Taking real measurements into account is important.

7.3 | Convergence acceleration

Our work proposes to accelerate the damping update in (3) through a reinforcement technique. For all diffusion-based approaches, that is, for any choice of $\omega^{(\text{diff})}$ and with and without an adaption of this value according to (3), the runtimes per time step eventually converge toward a similar value. Measurements in Figure 6 show that our reactive approach tends to "over-balance," unless we reduce $\omega^{(\text{diff})}$ in each time step. Over-balancing manifests in a large number of over offloaded tasks which trigger an emergency and thus induce a steep decline of tasks afterward. It is only $\omega^{(\text{reinf})} = 1$, where no emergency is triggered and we thus do not observe a rapid decrease of offloaded tasks. With $\omega^{(\text{reinf})} = 1$, both overshooting and retreat are damped, as (3) triggers an almost monotonous decay of $\omega^{(\text{diff})}$. For $\omega^{(\text{reinf})} = 0.5$, the diffusion parameter is not immediately decreased. It even increases over the first few steps. And once a rank hits the blacklist, (3) increases $\omega^{(\text{diff})}$ of the rank which caused the blacklisting again. The rank consequently retreats quickly.

Both choices of a dynamic change of $\omega^{(\text{diff})}$ outperform a static diffusion constant. By means of a rapid reduction of runtime, a quick reduction of $\omega^{(\text{diff})}$ is the best choice after a massive rebalancing step which is induced here by the initial domain decomposition but also might result from dynamic AMR. We however do observe that quick reincreases of $\omega^{(\text{diff})}$ due to small $\omega^{(\text{reinf})}$ might be reasonable if a small number of redistributed tasks is an objective, too. The reinforcement acts as additional penalty to the underlying optimization problem which takes task offloading cost into account.

If we repeat our benchmark with 14 ranks ($\omega^{(\text{diff})} = 0.5$ and $\omega^{(\text{reinf})} = 1$), and benchmark our reactive scheme against CCP, we see CCP yield an aggressive initial task decomposition (Figure 7). This is qualitatively in line with Figure 5: CCP's time per timestep is reduced much faster compared with the diffusion-only run. Reactive diffusion however is superior to CCP in the end as it takes the real behavior of the machine into account. We emphasize that these experiments use CCP to determine the initial distribution but then let diffusion take over. CCP speeds up the initial distribution, but it also seems to steer the reactive approach into a local minimum, and diffusion fails then to improve upon this load balancing further.

7.4 | Sudden performance drops

All experiments so far employ stationary grids. As a result, the task distribution converges toward a steady state. However, one might argue that a proper (re)balancing of the workload would be more effective in this case. Yet, this holds if and only if the typical rebalancing cost including all data movements is significantly lower than 10 to 20 time steps, as this is the characteristic timescale of our reactive load balancing to yield good time to solution ratios.

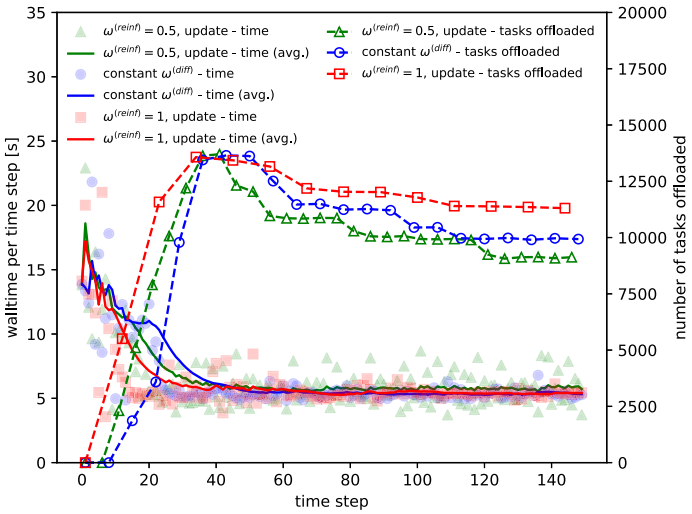


FIGURE 6 Runtime comparison of three executions of the diffusive algorithm. All executions start with $\omega^{(\text{diff})} = 1$. Two of them alter this diffusion parameter according to (3). We use a regular grid with $25 \times 25 \times 25$ cells on a single node hosting 14 ranks (SuperMUC phase 2)

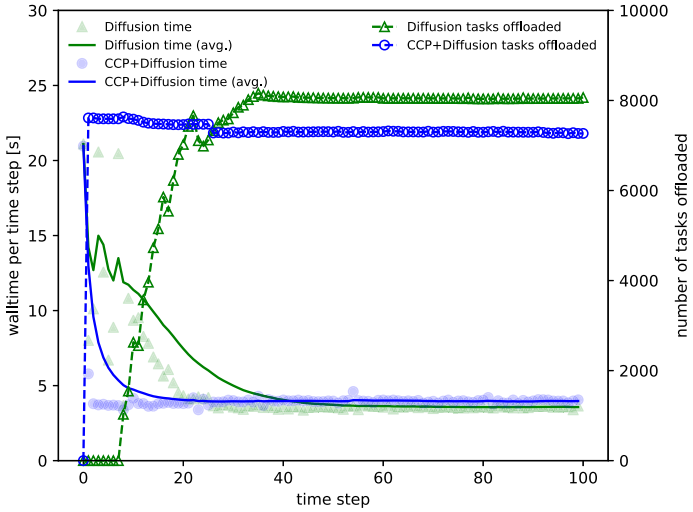


FIGURE 7 Runtime comparison of the diffusive algorithm with and without using CCP as an initial guess. We always initialize $\omega^{(\text{diff})} = 0.5$ (regular grid with $25 \times 25 \times 25$ cells on a single node of SuperMUC phase 2 hosting 14 ranks). CCP, chains-on-chains partitioning

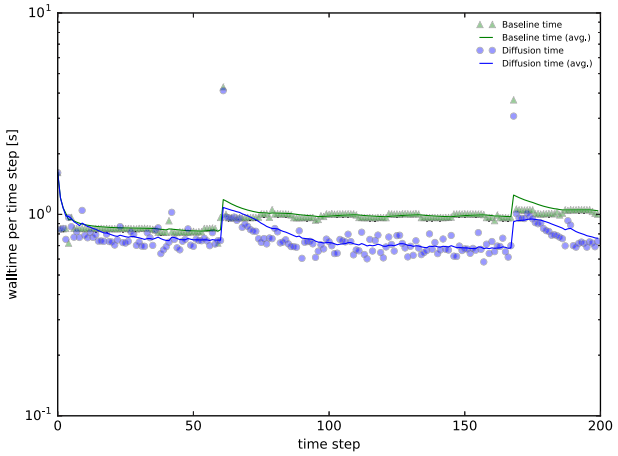


FIGURE 8 Time per timestep for a setup with dynamic AMR on 28 ranks on SuperMUC-NG. AMR, adaptive mesh refinement

Dynamic AMR yields peaks in the runtimes which are subsequently damped out by our diffusion (Figure 8). If the mesh changes dramatically and thus requires geometric rebalancing, the rebalancing cost amplifies the peak yet diminishes the subsequent tail, as it directly yields a relative balanced decomposition again. Domain repartitioning restarts our offloading yet with a good or even optimal initial guess of a partitioning. An alternative case of imbalancing results from the temporary degradation of node performance as it arises from temporary energy cuts due to overheating, hardware failures, coscheduling or congestion. It becomes extreme if the nodes recover quickly again. Performance runtime peaks, that is, efficiency break-downs, as discussed for the traditional balancing challenges here do arise, too, yet cannot be recovered and amortized due to task diffusion over the subsequent time steps. We thus focus on this last scenario from hereon: We (artificially) delay one rank in a 28 rank setup by 1 s every ten time steps.

The peaks every ten time steps make the time per time step flatter (Figure 9). The peaks are hard to spot, as temporary delays have both immediate effects—they delay classic boundary data exchange—as we well as knock-on effects due to a delayed delivery of outsourced tasks of the next time step as well as an impact on the diffusion metrics. Once we enable urgent local recomputes, the fluctuation of runtimes does not reduce dramatically; in particular the load diffusion continues to suffer from the strongly changing cost reported. However, the major peaks for an unbalanced task distribution are damped out and we improve the long-term time-to-solution by roughly 5%. The local urgent recomputes make the task distribution scheme really reactive and help to manage dynamically changing setups.

7.5 | Scaling studies

We continue our evaluation with some scaling studies. For this, we start with SuperMUC phase 2 and use the runtime per time step per degree of freedom on a single node as baseline. We normalize against the 28-core single node speed. Our data span 200 time steps, but we distinguish the runtimes within the first 50 iterations from the measurements within the remaining 150 steps. All following setups employ $\omega^{(\text{diff})} = 1$ and $\omega^{(\text{reinf})} = 1$.

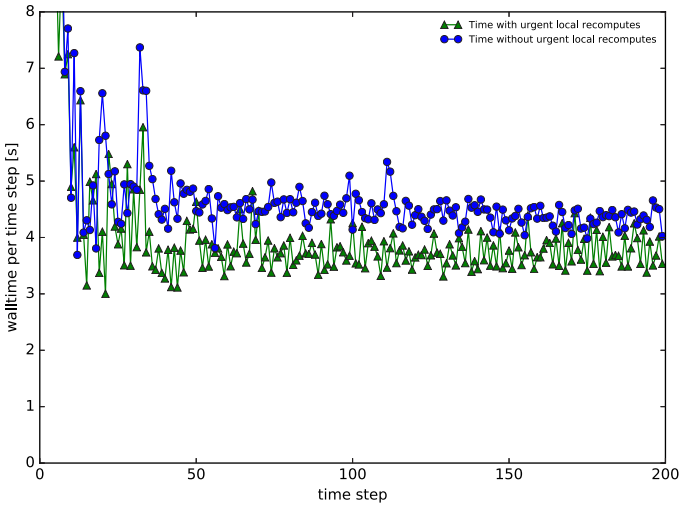


FIGURE 9 Runtime per time step for a 28 rank setup, where one rank is delayed by 1 s every 10 time steps. We compare our offloading with urgent recomputes to the offloading without them (SuperMUC NG)

Our first set of experiments (Table 1) study solely setups where we ensure that the geometric load balancing for the regular grid baseline is close to perfect. The baseline scaling thus is good, too. While the reactive diffusion improves upon the regular grid runtimes for the smaller node choices, its contribution is limited through the strong scaling regime: If the nodes' workload decreases, we eventually have enough cores available: It is cheaper to process tasks locally rather than to give them away—a decision encoded into our starvation check in Algorithm 2.

With AMR, reactive load balancing robustly improves the walltime for all experiments with limited node counts (Table 2). The improvement is very significant for static AMR. As we start from a regular grid, partition this grid perfectly, and then add the (static) refinement, our diffusion manages to compensate for any illbalancing that results from the AMR. For real-world setups, it might be more convenient to add an additional rebalancing step once the grid has become stationary, that is, to have both an initial partitioning to facilitate a geometric mesh construction plus a very good domain decomposition afterward. While our diffusion has been designed to act on top of such a load balancing, the data show that it can also replace the rebalancing step in some scenarios. For dynamic AMR, the mesh continues to change gradually over time. The load imbalances are typically small in the beginning but tend to increase in the long term. An example of this behavior is shown in Figure 8, where the dynamic AMR results in two prominent peaks in the time per time step due to the remeshing. There is a slight put persistent increase in time per timestep after each remeshing step. The diffusion adapts quickly to this small load imbalance without the need for an expensive global repartitioning step. For the other dynamic AMR setups on 20 nodes in Table 2, the overall observed load imbalance due to dynamic AMR is not large enough to justify a global rebalancing step. Yet, our reactive load balancing adapts and improves time to solution.

A major selling point of AMR is its capability to allow codes to scale up problem sizes in a fine granular way, while work is invested where it pays off most. Equidistant global refinement in contrast would make the degrees of freedom and, hence, the memory footprint explode. As load balancing

TABLE 1 Strong scaling speedups

Nodes	Regular grid	AMR
2	1.14	1.24
4	1.10	1.33
7	0.90	1.05
14	0.92	0.90
2	0.98	2.21
4	1.19	1.80
7	0.90	1.07
14	0.85	0.88

Note: For the regular grid experiments, we use a $25 \times 25 \times 25$ grid. AMR denotes that we add one level of AMR to this regular grid. The data columns show by which factor the baseline scalability (without task offloading) is improved. Entries smaller than 1 denote a slow-down, higher is better.
Note: We separate time steps 1 to 25 (top) from 26 to 200 (bottom).
Abbreviation: AMR, adaptive mesh refinement.

TABLE 2 Some typical reactive diffusion timings for various numbers of ranks

Machine	Ranks	Nodes	$\#\mathcal{P}$	AMR	Base [t]=s	Diffusion [t]=s
Phase 2	4	1	–	dyn	18.0	14.4
Phase 2	7	1	–	dyn	15.3	12.7
NG	28	2	–	dyn	4.2	3.8
NG	120	20	–	dyn	19.9	19.2
NG	480	20	–	dyn	8.9	8.5
Phase 2	40	20	33 201	stat	11.2	5.1
Phase 2	140	20	33 201	stat	8.1	4.6
Phase 2	280	20	33 201	stat	5.7	3.6

Note: The mean runtime per time step for 200 time steps is given. The label "dyn" stands for dynamic AMR whereas "stat" denotes static AMR. We use a single refinement level for both variants of AMR. For dynamic AMR, the number of tasks $\#\mathcal{P}$ is changing over time.
Abbreviation: ARM, adaptive mesh refinement.

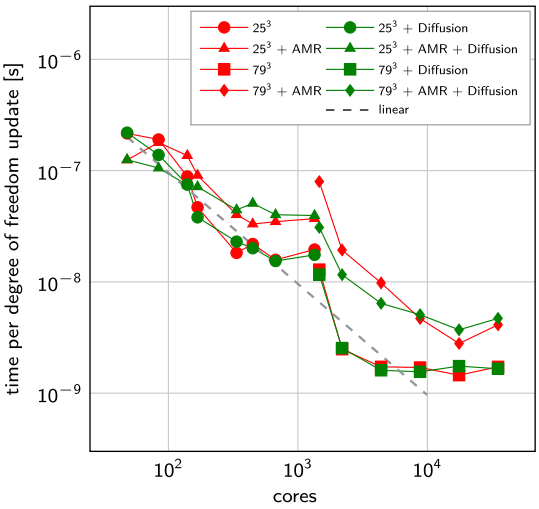


FIGURE 10 Strong scaling plots for various problem sizes with and without AMR on up to 731 ranks (SuperMUC-NG). AMR, adaptive mesh refinement

for varying grids is challenging, it is here where our approach helps most. It leverages the pressure to rebalance all the time and can compensate for slight ill-balancing.

We next benchmark our code systematically on multiple nodes of SuperMUC-NG on up to 731 ranks (Figure 10). Two regular grids of $25 \times 25 \times 25$ or $79 \times 79 \times 79$ serve as starting point. We validated that the chosen geometric load balancing approach balances the regular grid setups almost perfectly. Indeed, we observe reasonable strong scaling behavior for these regular grid configurations, that is, runtimes decrease close to linearly with increasing core counts before they enter a stagnation regime. Our reactive load balancing does not make a real difference for these reasonably balanced setups. The important observation is, however, that it also does not impose any significant runtime penalty. This is due to its totally nonblocking implementation.

We finally allow our adaptivity criterion to add further cells to the regular base grids. For 25^3 and 79^3 this yields around $39 \cdot 10^6$ or $833 \cdot 10^6$ degrees of freedom, respectively. The baseline balancing here struggles to yield perfect decompositions. Indeed, we observe that the performance curves suffer from some offset, while the increased number of degrees of freedom, compared with the regular baseline grid, ensures that we scale to slightly more cores. Our reactive load balancing manages to narrow this gap between cost per degree of freedom in a perfectly balanced world vs a world where we have to pay for the adaptivity and the resulting illbalancing. However, at large node counts, we run into the aforementioned issues (compare Figure 4), where the task offloading is limited in the number of tasks that it can offload due to possible starvation of local cores. Indeed, some offloading-related overhead becomes visible.

8 | DISCUSSION

We introduce a very lightweight task migration pattern—lightweight in a sense that the baseline implementation is hardly changed—which allows us to use time otherwise spent in MPI waits for actual work. Many task systems already can exploit MPI waits to process (local) tasks, and our

demonstrator realizes this feature, too. However, we hypothesize that—in almost all cases—such an eager processing of ready tasks introduces idle time later down the line. It is thus reasonable to lighweightly "fill up" wait time with remote tasks from ranks that are overbooked. As our scheme migrates tasks nonpersistently, this feature is particularly appealing for machines that suffer from speed fluctuations and for simulations where the load balancing is constrained due to the main memory available or load balancing overheads. Different to other approaches such as Reference 11 that translate the concept of task stealing into a distributed memory world, our scheme furthermore proactively outsources tasks, that is, we try to have the migrated tasks in place before the actual wait occurs. Otherwise, internode latency would become a challenge.

There are natural shortcomings of the present approach. First, our algorithms focus on ready tasks only. Tasks that have dependencies[16,33] are not supported. On the long term, it is interesting to migrate whole task assemblies if a task set as a whole requires less data per computation to exchange than its individual tasks. Migrating task subgraphs also helps in situations where the number of ready tasks alone is too small or just big enough to keep the local cores busy. That is, it helps whenever migrating load of ready tasks would compromise the local occupation. Second, our algorithms are not yet memory aware. There is no quota on the maximum number of stolen tasks hosted by a victim. Victims consequently might exceed their memory. Memory consumption could be another blacklisting criterion. Third, we have chosen several "magic" parameters for our experiments. While they yield meaningful results, we cannot claim that they are optimal. Autotuning here might improve the code's performance.[34] Finally, it might be reasonable to take the network topology as well as the logical rank topology into account when a rank selects its victim. Rather than choosing the most underbooked rank globally, we could offload tasks to nearby ranks. This constrains the task migration but avoids that offloading adds more edges to the logical MPI communication graph that many codes tailor toward a network architecture.

The weakest point we see in terms of methodology is the lack of an appropriate notion of criticalness. Our experiments run into situations where victim ranks are given too many remote tasks. This delays their actual delivery of information such as boundary data and eventually slows down critical ranks further. They however do not recognize this as they are not waiting for an outsourced task. Such complex causal dependencies cannot be tracked by our current notion of an emergency. We track overloading in a compute sense, but lack a detector for overloading in a bandwidth or MPI overhead (too many pending nonblocking messages) sense.

We have extensively invested into a scheme which ensures that reasonable progress is made on the asynchronous MPI transfers without sacrificing a thread.[21] We use aggressive polling. Yet, this cannot detect congestion. While the prioritization of MPI messages might mitigate this problem to some degree, we would appreciate if there were an MPI monitoring, that is, online performance analysis that can tell the application if the MPI subsystem enters a critical state. This could be realized via software[12] supervising the machine state. Alternatively, "intelligent" communication devices alike the SmartNIC technology could host the monitoring.

On the shared memory side, it remains open to which degree our choice of TBB as tasking base with manual tweaking of features alike prioritization affects the performance results. All proposed software building blocks currently are extracted into a standalone software package such that they can be used more easily with other codes.[35] As part of this roll out, we also explore the integration into OpenMP. On the long term an abstraction over various tasking paradigms[36] however might become necessary, such that we can systematically study the interplay of tasking approach and our balancing.

Our lightweight task migration realizes push semantics: Oversubscribed ranks deploy work to other ranks. This approach differs to strategies where ranks know their task workload prior to the computation—though they might permanently renegotiate, that is, balance such responsibilities—or codes with pull semantics, where ranks "grab" tasks from a (distributed) repository.[15] While all paradigms might yield comparable data distribution graphs, our code migrates tasks only temporarily, that is, sends results back. Our induced data flow graph is cyclic. It is thus lightweight as it does not redistribute data permanently. It is not lightweight by means of data moves, as every temporary task migration relies on a send forth and a send back.

9 | OUTLOOK

The exact interplay of our scheme with various dynamic load balancing schemes or more sophisticated numerics is beyond scope for the present article. We do however expect that our approach has beneficial knock-on effects: If load balancing is semistatic,[33] that is, rebalanced only every k steps, our approach allows us to migrate work less frequently (similar to our previous work[20]). This reduces AMR overhead. If load is balanced continuously in a diffusive style, we may assume that the diffusion rate, that is, the amount of data migration per step, can be chosen smaller with our approach. This reduces bandwidth requirements. On accelerator-driven machines, where bandwidth and local memory are notoriously short, we may assume that our approach offers an alternative to the difficult heterogeneous scheduling.[37] Our approach would make each accelerator a designated victim and thus hide the complexity of persistent data migration to balance load between accelerators.

On the numerics side, we will investigate nonlinear equation systems in the ADER-DG context. Such schemes require iterative Picard or Newton solves per $\mathcal{P}$ task.[26] This renders the cost per $\mathcal{P}$ evaluation very hard to predict. ADER-DG is often contrasted with standard Runge-Kutta (RK) methods. Indeed, our ideas should apply to RK as well, though their lack of a space-time evaluation might imply that the evolution of the cells is cheaper. In return, we might get away with a smaller memory footprint. This renders RK another interesting numerical scheme to study. ADER-DG's attractiveness is its inherent fit to adaptive, local time stepping. Again, such a time stepping renders the workload prediction very difficult. It thus

should benefit from our approach. Finally, we plan, on the long term, to study the interplay of our Eulerian mindset with Lagrangian techniques[38] which may be applied on tree-structured adaptive Cartesian grids.[39] While the load inhomogeneity resulting from these couplings is obvious, it is notably the fact that such setups have to balance both memory and compute load rigorously which makes it interesting for our approach.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). Tobias has received additional funding around ExaHyPE through EPSRC's Excalibur programme under grant number EP/V00154X/1 (ExaClaw). We also acknowledge support and computing resources provided by the Leibniz Supercomputing Centre (grant no pr48ma). Special thanks are due to all members of the ExaHyPE consortium who made this research possible—in particular to Leonhard Rannabauer for his work on the elastic wave equation solver and the LOH.1 setup. All underlying software is open source.[22]

ORCID

Philipp Samfass https://orcid.org/0000-0001-8052-2519
Tobias Weinzierl https://orcid.org/0000-0002-6208-1841

REFERENCES

1. Charrier DE, Hazelwood B, Tutlyaeva E, et al. Studies on the energy and deep memory behaviour of a cache-oblivious, task-based hyperbolic PDE solver. *Int J High Perf Comput Appl*. 2019;33(5).
2. Acun B, Miller P, Kale LV. Variation among processors under turbo boost in HPC systems. Paper presented at: Proceedings of the ICS '16; 2016:6:1-6:12; New York, NY, ACM.
3. Charles J, Jassi P, Ananth NS, Sadat A, Fedorova A. Evaluation of the intel core i7 turbo boost feature. Paper presented at: Proceedings of the IISWC '09; 2009:188-197; IEEE.
4. Jain N, Bhatele A, Howell LH, et al. Predicting the performance impact of different fat-tree configurations. Paper presented at: Proceedings of the SC '17; 2017:50:1-50:13; New York, NY, ACM.
5. Pollard SD, Jain N, Herbein S, Bhatele A. Evaluation of an interference-free node allocation policy on fat-tree clusters. Paper presented at: Proceedings of the SC '18; 2018:26:1-26:13; Piscataway, NJ, IEEE Press.
6. McCalpin JD. HPL and DGEMM performance variability on the Xeon platinum 8160 processor. Paper presented at: Proceedings of the SC '18; 2018:18:1-18:13; Piscataway, NJ, IEEE Press.
7. Charrier DE, Hazelwood B, Weinzierl T. Enclave tasking for DG methods on dynamically adaptive meshes. *SIAM J Sci Comput*. 2020;42(3):C69-C96.
8. Dumbser M, Käser M. An arbitrary high order discontinuous Galerkin method for elastic waves on unstructured meshes II: the three-dimensional isotropic case. *Geophys J Int*. 2006;167(1):319-336.
9. Jofre L, Borrell R, Lehmkuhl O, Oliva A. Parallel load balancing strategy for volume-of-fluid methods on 3-D unstructured meshes. *J Comput Phys*. 2015;282:269-288.
10. Reinarz A, Bader M, Charrier DE, et al. ExaHyPE: an engine for parallel dynamically adaptive simulations of wave problems. *Comput Phys Commun*. 2020;254:107251. https://www.sciencedirect.com/science/article/pii/S001046552030076X?via%3Dihub.
11. Garcia M, Corbalan J, Labarta J. LeWI: a runtime balancing algorithm for nested parallelism. Paper presented at: Proceedings of the ICPP '09; 2009:526-533.
12. Mao G, Böhme D, Hermanns MA, Geimer M, Lorenz D, Wolf F. Catching idlers with ease: a lightweight wait-state profiler for MPI programs. Paper presented at: Proceedings of the EuroMPI/ASIA '14; 2014:103:103-103:108; New York, NY, ACM.
13. Bottou L, Curtis F, Nocedal J. Optimization methods for large-scale machine learning. *SIAM Rev*. 2018;60(2):223-311.
14. Germain JDDS, McCorquodale J, Parker SG, Johnson CR. Uintah: a massively parallel problem solving environment. Paper presented at: Proceedings of the 9th International Symposium on High-Performance Distributed Computing; 2000:33-41.
15. Meng Q, Luitjens J, Berzins M. Dynamic task scheduling for the Uintah framework. Paper presented at: Proceedings of the MTAGS@SC '10; 2010:1-10; IEEE Computer Society.
16. Schaller M, Gonnet P, Chalk ABG, Draper PW. SWIFT: using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores. Paper presented at: Proceedings of the PASC '16; 2016:2:1-2:10; New York, NY, ACM.
17. Acun B, Gupta A, Jain N, et al. Parallel programming with migratable objects: Charm++ in practice. Paper presented at: Proceedings of the SC '14; 2014:647-658; IEEE Press.
18. Kaiser H, Heller T, Adelstein-Lelbach B, Serio A, Fey D. HPX: a task based programming model in a global address space. Paper presented at: Proceedings of the PGAS '14; 2014:6:1-6:11; New York, NY, ACM.
19. The Stellar Group HPX; 2019. http://stellar-group.org/tag/hpx.
20. Samfass P, Klinkenberg J, Bader M. Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework sam(oa)^2. In: Nikolopoulos DS, de Supinski BR, eds. *Paper presented at: Proceedings of the 2018 IEEE International Conference on Cluster Computing (CLUSTER)*. Piscataway, New Jersey: IEEE; 2018:337-347.
21. Hoefler T, Lumsdaine A. Message progression in parallel computing—to thread or not to thread? In: Ishikawa Y, ed. *Paper presented at: Proceedings of the 2008 IEEE International Conference on Cluster Computing*. Piscataway, New Jersey: IEEE; 2008:213-222.
22. Bader M, Dumbser M, Gabriel AA, Igel H, Rezzolla L, Weinzierl T. *ExaHyPE—An Exascale Hyperbolic PDE Solver Engine*. 2019. http://www.exahype.org.
23. Pinar A, Aykanat C. Fast optimal load balancing algorithms for 1D partitioning. *J Parall Distrib Comput*. 2004;64(8):974-996.
24. Igel H. *Computational Seismology: A Practical Introduction*. 1st ed. Oxford, UK: Oxford University Press; 2016.
25. Day SM, Bielak J, Dreger D, et al. *Tests of 3D Elastodynamics Codes: Final Report for Lifelines Program Task 1A02. Technical Report*. San Diego State University; 2003.

26. Zanotti O, Fambri F, Dumbser M, Hidalgo A. Space-time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting. *Comput Fluids*. 2015;118:204-224.
27. Charrier DE, Weinzierl T. A communication-avoiding ADER-DG realisation; 2019. arXiv:1801.08682 (submitted).
28. Hesthaven JS, Warburton T. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Texts in Applied Mathematics. Vol 54. New York, NY: Springer; 2008.
29. Weinzierl T. The Peano software – parallel, automaton-based, dynamically adaptive grid traversals. *ACM Trans Math Softw*. 2019;45(2):14:1-14:41.
30. Samfass P, Weinzierl T, Hazelwood B, Bader M. teaMPI—replication-based resilience without the (performance) pain. Paper presented at: Proceedings of the ISC 2020; 2020. (in print).
31. Reinders J. *Intel Threading Building Blocks*. 1st ed. Sebastopol: O'Reilly & Associates, Inc; 2007.
32. Buettner D, Acquaviva JT, Weidendorfer J. Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks. Paper presented at: Proceedings of the ICPADS '13. IEEE Computer Society; 2013:208-215; Washington, DC.
33. Bremer MH, Bachan JD, Chan CP. Semi-static and dynamic load balancing for asynchronous hurricane storm surge simulations. Paper presented at: Proceedings of the 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI, PAW-ATM@SC 2018; November 16, 2018:44-56; Dallas, TX.
34. Eckhardt W, Glas R, Korzh D, Wallner S, Weinzierl T. On-the-fly memory compression for multibody algorithms. In: Joubert GR, Leather H, Parsons M, Peters F, Sawyer M, eds. *Advances in Parallel Computing*. Vol 27. Amsterdam: IOS Press; 2015:421-430.
35. Klinkenberg J, Samfass P, Bader M, Terboven C, Müller MS. CHAMELEON: reactive load balancing for hybrid MPI+OpenMP task-parallel applications. *J Parall Distrib Comput*. 2020;138:55-64.
36. Alomairy R, Ltaief H, Abduljabbar M, Keyes D. *Abstraction Layer for Standardizing APIs of Task-Based Engines. Technical Report*. : , King Abdullah University of Science and Technology; 2019. http://hdl.handle.net/10754/656693.
37. Sundar H, Ghattas O. A nested partitioning algorithm for adaptive meshes on heterogeneous clusters. Paper presented at: Proceedings of the 29th ACM on International Conference on Supercomputing ICS '15; 2015:319-328; ACM.
38. Dubey A, Antypas K, Daley C. Parallel algorithms for moving Lagrangian data on block structured Eulerian meshes. *Parall Comput*. 2011;37(2):101-113.
39. Weinzierl T, Verleye B, Henri P, Roose D. Two particle-in-Grid realisations on spacetrees. *Parall Comput*. 2016;52:42-64.

How to cite this article: Samfass P, Weinzierl T, Charrier DE, Bader M. Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement. *Concurrency Computat Pract Exper*. 2020;e5916. https://doi.org/10.1002/cpe.5916