Journal of Parallel and Distributed Computing 138 (2020) 55–64

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

CHAMELEON: Reactive Load Balancing for Hybrid MPI+OpenMP Task-Parallel Applications

Jannis Klinkenberg[a,*], Philipp Samfass[b], Michael Bader[b], Christian Terboven[a], Matthias S. Müller[a]

[a] Chair for High Performance Computing, IT Center, RWTH Aachen University, Aachen, Germany
[b] Department of Informatics, Technical University of Munich, Garching, Germany

ARTICLE INFO

Article history:
Received 19 July 2019
Accepted 5 December 2019
Available online 16 December 2019

Keywords:
Reactivity
Task migration
Hybrid MPI+openMP
Load balancing
Tasking

ABSTRACT

Many applications in high performance computing are designed based on underlying performance and execution models. While these models could successfully be employed in the past for balancing load within and between compute nodes, modern software and hardware increasingly make performance predictability difficult if not impossible. Consequently, balancing computational load becomes much more difficult. Aiming to tackle these challenges in search for a general solution, we present a novel library for fine-granular task-based reactive load balancing in distributed memory based on MPI and OpenMP. With our approach, individual migratable tasks can be executed on any MPI rank. The actual executing rank is determined at run time based on online performance data. We evaluate our approach under an enforced power cap and under enforced clock frequency changes for a synthetic benchmark and show its robustness for work-induced imbalances for a realistic application. Our experiments demonstrate speedups of up to 1.31X.

1. Introduction

Over the past decades, most scientific applications have been developed under the assumption of a homogeneous execution environment where every compute node – and even every single core – in a larger cloud or High Performance Computing (HPC) system has a constant equal speed. Therefore, executing the same work on every node should require the same computation time. In the past, this execution model was shown to be highly accurate and efficient for balancing computational load. However, as both hardware and software become increasingly complex, this model might no longer be sufficient on current and future systems.

Today's architectures already exhibit run time variations, e.g., with dynamic voltage and frequency scaling (DVFS), sophisticated memory hierarchies comprising caches, DRAM, NVRAM and HBM or features like Intel's Turbo Boost [1,4]. Further, CPU power efficiency variations arising from the manufacturing process can lead to performance variations in presence of an enforced power cap [12]. Another source of *dynamic variability* stems from modern numerics in simulation applications such as particle simulations or iterative codes employing adaptive mesh refinement (AMR) where the workload distributed across processing units changes over time causing load imbalances both in shared and distributed memory. In the ADER-DG numerical scheme with a-posteriori limiting [22], additional computation work arises dynamically whenever the numerical solution is not considered to be physically admissible.

Consequently, the assumption that the execution time can be accurately predicted does no longer apply. In order to prevent load imbalances and performance declines resulting from *performance variability* both in hardware and software, we believe that it is necessary that applications are able to dynamically react on the changing execution conditions.

Traditional approaches like global re-partitioning of work were an effective technique to ensure proper balance in the past. However, they are based on a cost model to predict future execution time. Such a cost model is doomed to fail in increasingly complex hardware and software environments. Existing solutions usually apply the re-partitioning at fixed synchronization points, where load balancing is performed exclusively, i.e., nothing else is happening except moving data or work.

To mitigate the shortcomings of predictive load balancing approaches, we present *CHAMELEON*, a library for fine-grained reactive load balancing of task-parallel MPI+X applications that allows reactive load balancing within and across process boundaries. Further, as our goal is not to create a completely new programming language or paradigm, our library rather builds on

* Corresponding author.
E-mail addresses: j.klinkenberg@itc.rwth-aachen.de (J. Klinkenberg), samfass@in.tum.de (P. Samfass), bader@in.tum.de (M. Bader), terboven@itc.rwth-aachen.de (C. Terboven), mueller@itc.rwth-aachen.de (M.S. Müller).

https://doi.org/10.1016/j.jpdc.2019.12.005
0743-7315/© 2019 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

top of the established standards MPI and OpenMP and provides an incremental solution to support the large amount of existing codes developed in C, C++ and Fortran. In our previous work [20], we carried out a feasibility study of our reactive approach in the PDE framework sam(oa)^2. In this work, we extend and improve our concept as well as generalize and modularize it to make it available to other MPI-parallel applications. Consequently, this paper makes the following contributions:

1. We present the first conceptual generalization of reactive load balancing to arbitrary MPI-parallel task-based applications, detailing both requirements and limitations associated with it.
2. We present our library implementation based on MPI+OpenMP that allows an incremental integration into existing task-based applications with minimal programming efforts. Further, we have a deeper look at implementation extensions and decisions that have changed compared to our feasibility study.
3. To demonstrate the effectiveness and scalability of our approach we conduct a systematic evaluation using a comprehensible synthetic benchmark comparing different implementation decisions as well as the sam(oa)^2 framework [15].

This paper is an extended version of our previous publication [14] at the WLPP (Workshop on Language-Based Parallel Programming) held in conjunction with PPAM (International Conference on Parallel Processing and Applied Mathematics). For the journal version, we significantly extended our evaluation section: We now present an additional performance experiment that investigates the load balancing behavior of our library with a highly imbalanced benchmark. In this setting, we discuss the factors that affect performance and load balancing. Besides, we conducted new tests with the sam(oa)^2 framework using the ADER-DG numerical time-stepping scheme to demonstrate the effectiveness of reactive load balancing for unpredictable imbalances caused by the numerical scheme. Further, we isolated and expanded related work.

The remainder of this paper is structured as follows. Section 2 introduces the fundamental concept and requirements for a reactive hybrid task-based load balancing solution. In Section 3, we describe our implementation and different design choices. An experimental evaluation is carried out in Section 4. Section 5 describes related work and discusses the differences compared to our approach before we conclude and present future work in Section 6.

2. Reactive load balancing

This section details our concept of fine-granular task-based load balancing in both shared and distributed memory. We review fundamental assumptions and objectives with respect to the generalizability of our approach. Further, we identify three essential components: a *task-based execution environment*, *self introspection* and an *analysis* component and discuss their requirements and implications for a general solution.

2.1. Assumptions & objectives

Our guiding underlying observation is that any imbalances (both predictable and unpredictable imbalances) manifest in increased waiting times at global MPI synchronization points. In case an imminent imbalance is detected, our approach attempts to quickly/immediately migrate tasks to underloaded processes, thus replacing the aforementioned waiting times with useful computation. A key assumption is that tasks represent *basic units of work* without any side effects (e.g., accessing global variables inside the task) that can be executed on the local or on a remote process. As these tasks are candidates for being executed remotely, we call them *migratable*.

In our previous work [20], we identified the following key objectives of a distributed work stealing implementation:

1. **Reactivity:** Since load imbalances can result from dynamically changing execution conditions or computational load on a very short time scale, it is necessary to detect these changes as quickly as possible.
2. **Smart decision-making:** Relying on permanently collected introspection data an implementation has to identify an emerging imbalance and decide whether to migrate tasks or not. Further, it has to select adequate victims to migrate tasks to. However, inaccurate or incorrect decisions can result in a performance decline.
3. **Hiding overhead:** Compared to work stealing in shared memory, migrating tasks in distributed memory induces additional overhead as task-related information and data needs to be transferred over the network. Consequently, it is desired to migrate tasks as soon as possible to sufficiently overlap communication with computation and hide any migration overhead.
4. **Ease of integration:** Augmenting existing applications with task migration should not require extensive programming efforts or code modifications.
5. **Generalization and modularity:** Although the objective is to create a generally applicable solution that can be integrated into arbitrary applications, it might be desired and profitable to customize introspection/load specification or migration strategy in order to incorporate domain and application knowledge. Nevertheless, an implementation should provide a default behavior.

2.2. Execution environment for migratable tasks

An execution environment for migratable tasks needs to satisfy the following requirements. First, it has to provide means to create migratable tasks by specifying an *action* to perform as well as *data items* accessed by the task. To be able to migrate tasks via inter-process communication the specification for a *data item* must contain a reference to the corresponding data, its size, as this information might not be available automatically (e.g., when using native pointers in C/C++), and a type $t \in \{input, output, input \; \& \; output\}$ that indicates whether the corresponding item is only used within the task ($input$) or whether it is updated and needs to be available for subsequent operations ($output$).

To trigger the execution of queued tasks synchronization is required, similar to `taskwait` or `barrier` in OpenMP. However, this synchronization is not allowed to terminate until all tasks (of all processes) and outstanding communication are finished. An implementation can then decide at run time to either execute a task locally or migrate the task to another process. Contrary to other approaches that perform a redistribution in separate defined phases, it is desired to detect impending imbalances and take appropriate counter measures as soon as possible to overlap data transfers and communication with calculation, i.e., the execution of other tasks.

After a migrated task has been executed on a remote process, *data items* specified as *output* will be sent back to the original process that created the task. This process allows an incremental integration into existing applications that use the resulting data for subsequent calculations or communication such as a halo exchange, effectively preventing a complex change of communication partners.

It is recommended to perform both a thread-parallel task creation and task execution to apply load balancing within a process and exploit a large degree of shared memory concurrency.

2.3. Introspection & analysis

A reactive solution requires to quickly detect changing run conditions, hardware behavior or unequal workload distribution. Hence, each process continuously monitors its execution condition and characteristics. As suitable characteristics can range from coarse-grained information to fine-grained metrics (e.g., time measurements or hardware performance counters) and might depend on the application, one question is:

Question 1. *What is an appropriate general load metric that can be used for arbitrary applications?*

Complemented with an analysis component that consolidates collected per-process introspection data to a coherent global view this procedure lays the foundations for identifying dynamically changing execution conditions and predicting imbalances between processes. Based on the result of the analysis the implementation can decide to trigger task migrations in order to mitigate upcoming imbalances. Yet, an implementation also needs to address the following questions (implementation details are discussed in Section 3):

Question 2. *Based on provided introspection/load data, what is a good default strategy to decide whether to migrate tasks and when to stop migrating?*

Question 3. *How to select proper victims for task migration?*

3. Implementation

We implemented our reactive load balancing concept in an MPI+OpenMP-parallel library called *CHAMELEON* that is available as a free open source software under a BSD 3 license[1] allowing existing codes to use our proposed solution with only minimal code modifications.

3.1. A migratable task paradigm

In contrast to our previous application-level prototype [20] where we used a pull-oriented work stealing approach, we now follow a push-oriented mechanism, where *migratable* tasks are offloaded from overloaded to underloaded processes. While this is logically only an inversion of responsibility, it saves some communication overhead: in the pull-oriented variant, a handshake between a stealing rank and the selected stealing victim was required. Further, offloading allows us to leverage OpenMP's target offloading infrastructure making a first step towards an extension of OpenMP's programming model. Conceptually, instead of offloading tasks to an accelerator, tasks are offloaded to MPI processes in our approach. However, while the decision where to execute the offloaded task is already made at task creation for classic OpenMP offloading, we need to defer that decision to runtime as we strive to reactively balance load.

We implemented a custom libomptarget plugin in the LLVM OpenMP runtime that calls our library at task creation. Combined with the clang compiler, this enables us to fully specify a *migratable* task using the `#pragma omp target` directive and its data environment using the associated `map` clause. The compiler takes care of creating a task entry function and generates appropriate

[1] https://github.com/chameleon-hpc/chameleon

Listing 1: Example of a synthetic dense matrix multiplication code creating migratable tasks in parallel with the OpenMP target construct or API

```
// function that performs MxM
void compute_matrix_matrix(double *A, double *B, double *C, int
      mat_size);

int main()
{
  ... // MPI initialization , matrix allocation , ...
  void* lit_size = *(void**)(&size); // pointer literal
        representing value of size
  #pragma omp parallel
  {
    #pragma omp for nowait
    for(int i=0; i<num_tasks; i++) {
      double *A = matrices_a[i];
      double *B = matrices_b[i];
      double *C = matrices_c[i];

#if USE_OPENMP_TARGET_CONSTRUCT
      #pragma omp target map(tofrom: C[0:size*size]) map(to: A
            [0:size*size], B[0:size*size])
      compute_matrix_matrix(A, B, C, size);
#else // API approach
      map_data_entry_t* args = new map_data_entry_t[4];
      args[0] = chameleon_map_data_entry_create(A, size*size*
            sizeof(double), MAPTYPE_INPUT);
      args[1] = chameleon_map_data_entry_create(B, size*size*
            sizeof(double), MAPTYPE_INPUT);
      args[2] = chameleon_map_data_entry_create(C, size*size*
            sizeof(double), MAPTYPE_OUTPUT);
      args[3] = chameleon_map_data_entry_create(lit_size,
            sizeof(void*), MAPTYPE_INPUT | MAPTYPE_LITERAL);

      cham_migratable_task_t *cur_task = chameleon_create_task
            ((void *)&compute_matrix_matrix, 4, args);
      chameleon_add_task(cur_task);
#endif
    }
    // trigger execution (In background: introspection + task
          migration)
    chameleon_distributed_taskwait();
  }
  ... // MPI finalization , clean up
}
```

calls to the custom plugin, where both a reference to the task entry function (*action*) and the task's data environment (*data items*) are then forwarded to our library. As usually the same hybrid binary is executed by all ranks, an offset from the start of the loaded binary to the corresponding task entry function can be used to determine the correct entry point on a remote rank in case a task is offloaded.

While creating *migratable* tasks using OpenMP's target offloading construct is our preferred choice, we found that there is a lack of compiler support for this variant, specifically for Fortran compilers. Therefore, we additionally implemented an API (C and Fortran bindings available) that allows to create *migratable* tasks by manually specifying a reference to the task entry function and the task's data environment. An example code snippet for both approaches is shown in Listing 1.

3.2. Communication infrastructure

We implemented a communication infrastructure to handle task migration as well as introspection and continuous global exchange of online load information. All communication is fully non-blocking using a dedicated communication thread on a separate core per rank with the desire to overlap communication and computation. We found that using a dedicated core is essential to guarantee sufficient progression of MPI messages and achieving our objective of fine-granular reactivity. Similar findings have been reported in [7,11]. In contrast to predictive load balancing, there is no mutual a-priori agreement on the task migration pattern. In fact, the reactive nature of our approach demands that an overloaded rank can very quickly migrate tasks which requires responsiveness on the sender and victim rank. Even employing hyper-threading where a physical core is shared by the communication thread and another application thread is not a viable option here: there is in-determinism in thread scheduling by the OS and the hyper-thread would compete for resources with a computation thread, thus creating additional imbalances and degrading reactivity.

3.3. Task execution and termination detection

As mentioned in Section 2.2 an implementation requires a synchronization point that ensures that all tasks of all ranks (i.e., local and migrated tasks executed on a remote rank) and all outstanding communication (i.e., transferring results back to the original rank) have been completed. We implemented a `chameleon_distributed_taskwait` function (see Listing 1 line 31) where each thread of each rank participates in completing the created tasks and communication before terminating. Although there are more efficient solutions for global termination detection like proposed by Dinan et al. [8], we already exchange load information continuously. Hence, we append the number of outstanding operations per rank to the corresponding messages, achieving a termination detection almost for free. The `chameleon_distributed_taskwait` routine triggers the execution of queued tasks and activates the communication thread that handles task migration and load exchange. As it is desired to overlap the communication as much as possible, our implementation prioritizes the execution of incoming migrated tasks before working on local tasks.

3.4. Making effective load balancing decisions

To build a generally applicable responsive load balancing solution three relevant questions have been pointed out in Section 2.3 that we address with our implementation.

What is an appropriate general load metric that can be used for arbitrary applications?

A suitable introspection metric that precisely reflects the load or run condition of a rank is the key for a good outcome. This metric might highly depend on the hardware, application and domain knowledge confronting us with two conflicting goals. On one hand it might be desired to incorporate such domain or application knowledge. On the other hand we are seeking for an appropriate default metric that can be used for arbitrary applications. Since most tasking codes apply over-decomposition we selected the *number of tasks per rank* as a general load metric. While it is easy to determine and induces low overhead compared to more sophisticated calculations, it might not work well for tasks with varying complexity or size. However, our library also provides a *tools interface* that enables the user to customize introspection, load specification and migration strategy including victim selection.

What is a good default strategy to decide whether to migrate tasks and when to stop migrating?

The migration strategy is a sensitive component of this approach. Although our solution targets to compensate small imbalances, migrating a tasks comes with an additional communication cost that has to be considered. An imbalance between sender and victim rank has to be large enough in order to amortize the task migration. Thus, our default strategy only migrates tasks if the imbalance is larger than a configurable absolute or relative *threshold*. Further, migration should not be performed too late, as this would prevent full overlap. As a result, threads would only wait for completing outstanding communication. It is critical to determine when to stop migrating tasks. Our strategy only migrates if the *number of outstanding local tasks per rank* is greater than a configurable value, e.g. *number of threads*. Thus, we ensure to keep all threads busy while preventing idle times and overhead caused by late migrations. All thresholds can be set via environment variables.

How to select proper victims for task migration?

Our previous prototype selected the rank with the highest load as victim for task stealing. However, as now task migration decisions are made on each rank separately[2] in a short time frame only pushing tasks from the rank with maximum load might not be sufficient and pushing tasks to the rank with the lowest load might lead to contention or could result in load imbalances again. Our library applies a sort-based approach to identify proper victims aiming to achieve a good load balance while avoiding contention as illustrated in Fig. 1. Ranks are sorted by load. After that, in case the imbalance between the current rank and the corresponding counterpart exceeds the configured *thresholds* this rank is selected as victim.

4. Experimental evaluation

In this section, we evaluate our generalized approach and implementation decisions against hardware variability (Section 4.1) and work-induced imbalances (Section 4.2).

All tests are conducted on the HPC production system of RWTH Aachen University CLAIX that is equipped with an Intel Omni-Path interconnect and dual-socket Intel Xeon E5-2650v4 (codename "Broadwell") processor nodes with a TDP of 105 W and 24 cores in total running at 2.2 GHz.[3] Our library as well as benchmarks are compiled with Intel C/C++ or Intel Fortran Compiler 19.0.1 and Intel MPI 2018.4. Hybrid MPI+OpenMP application runs are performed using a single rank per node where OpenMP threads are pinned to cores using `OMP_PLACES` and `OMP_PROC_BIND`. In order to exploit the shared-memory parallelism of the nodes using OpenMP we usually differentiate between the following two situations, if not specified otherwise:

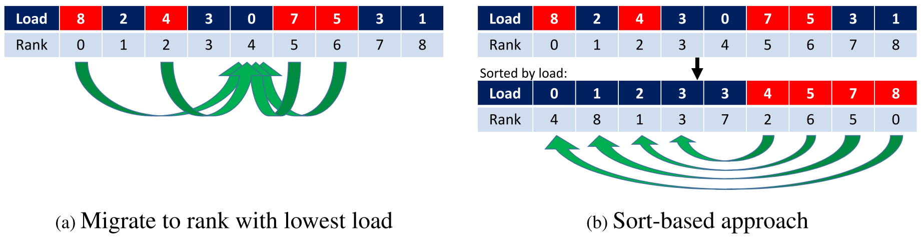
1. **Runs without task migration:** Applications are executed with n_{cores} OpenMP threads acting as a baseline.
2. **Runs with task migration:** As a separate communication thread is running on the last core, applications are executed with $n_{cores} - 1$ threads.

4.1. Robustness against hardware variations

In order to evaluate the robustness against dynamic variability caused by hardware we use a synthetic hybrid matrix multiplication benchmark, where each rank has to perform a configurable number of dense matrix multiplications $C = A * B$ with the same computational complexity (see Listing 1). To ensure an adequate execution time and sufficiently large tasks, every rank has to solve 2400 multiplications with a matrix size of $S = 600 \times 600$. In an ideal scenario where all nodes and CPUs have the same speed and efficiency, all ranks are expected to finish the calculation in the same time. However, to demonstrate the effectiveness of reactive load balancing when working with dynamic imbalances and to show effects of varying hardware efficiencies we run experiments under an enforced power cap or in the presence of variable CPU core clock frequencies.

[2] Migration decisions are made on each rank separately based on per rank load information that have been exchanged before. Consequently, this step does not require any additional two-sided or collective communication.

[3] Although we planned to conduct the tests on our new Intel Xeon Skylake processors, this partition was still in the process of getting into production at the time of creating the paper.



Load	8	2	4	3	0	7	5	3	1
Rank	0	1	2	3	4	5	6	7	8

Load	0	1	2	3	3	4	5	7	8
Rank	4	8	1	3	7	2	6	5	0

Fig. 1. Potential choices for identifying proper task migration victims assuming that only ranks with load higher than average (red) migrate tasks. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Experiment 1: Power Capping

In this experiment, we run the aforementioned benchmark on 4 nodes with a version that solely employs regular OpenMP tasks to balance load in shared memory. We compare this to an implementation variant that features our task migration approach, i.e., that is also capable of balancing load in distributed memory. Every run is conducted 10 times under enforced power caps ranging from 40 W to 105 W (no powercap). Resulting mean values and standard deviations are depicted in Fig. 2. However, it should be noted that the results highly depend on the energy efficiency of the selected compute nodes. For our tests we included a compute node known to have a lower energy efficiency.

Empirical tests have shown that the average power draw for such an application run is around 90 W. If the selected power cap is close to or larger than 90 W task migration will not result in much improvement but is also not suffering much from overhead. With lower thresholds effects from varying hardware efficiencies become visible and task migration can help mitigate arising imbalances. This leads to improvements of 4 % to 20 % depending on the selected power cap. As an example, an execution of the task migration version with a power cap of 60 W took on average 17.42 s. Investigating a single execution showed that during this time frame the dedicated communication threads that are also responsible for continuous self introspection (e.g. load of the corresponding rank) performed 1, 267, 978 load exchanges. Based on that information it was able to dynamically detect imbalances between ranks at run time leading to a migration of 148 out of 4800 tasks between the participating ranks.

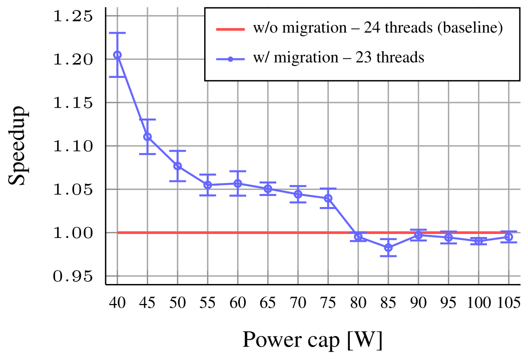


Fig. 2. Speedup under enforced power cap.

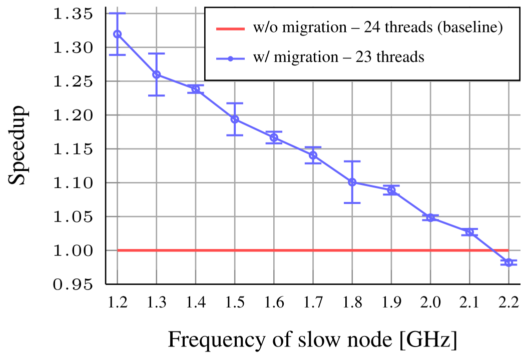


Fig. 3. Speedup with a single slow node.

Experiment 2: Varying core clock frequencies

To provoke imbalances we run the same setups as in experiment 1. However, we are not setting any power cap but use *likwid-setFrequencies* [21] to reduce the core clock frequency of a single node whereas the other nodes run with the default frequency of 2.2 GHz. We conduct tests varying the frequency of the single slow node from 1.2 GHz to 2.2 GHz. Results are shown in Fig. 3. As expected, selecting 2.2 GHz (no frequency reduction) leads to a slight performance decline due to losing one core for communication purposes. Selecting a frequency close to the base frequency of the other nodes results in only a marginal speedup. With larger frequency differences, e.g. with 1.2 GHz, task migration achieves a speedup up to 1.31X.

4.2. Robustness against work-induced imbalances

While our load balancing approach is targeted at treating unpredictable imbalances, we also evaluated whether it can be used to improve performance of work-imbalanced codes or scenarios. In this context, we further examine the fundamental behavior of reactive load balancing and the factors that influence or limit its performance.

Experiment 3: Behavior with a highly imbalanced scenario

In this experiment, we investigate how well reactive load balancing works when dealing with highly imbalanced scenarios. For that purpose, we conduct tests with the same dense matrix multiplication application as introduced in Section 4.1 where computing a task corresponds to computing a dense matrix–matrix multiplication. To construct an extreme case we use two ranks where the first rank initially spawns 2400 migratable tasks with the same computational complexity and the second rank is idle, i.e., it spawns 0 tasks. Additionally, we either vary the size and complexity of tasks by choosing matrix sizes from $S = 50 \times 50$ up to 600×600 or the number of threads per rank from 2 up to 22. Ideally, a load balancing solution requiring no overhead would achieve a speedup of 2X by executing 1200 tasks on the second rank. Thus, our results compare a regular hybrid MPI+OpenMP execution without any inter-process load balancing to our reactive task migration approach. Each test is executed 10 times and mean values as well as standard deviations are reported. As described earlier, migrating tasks in distributed memory induces additional overhead. Efficiently balancing the load and hiding the communication overhead requires enough time for the migration

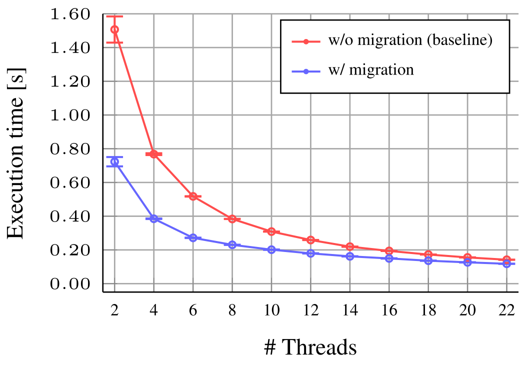


Fig. 4. Execution time for experiments with a matrix size of $S = 150 \times 150$ at different numbers of threads.

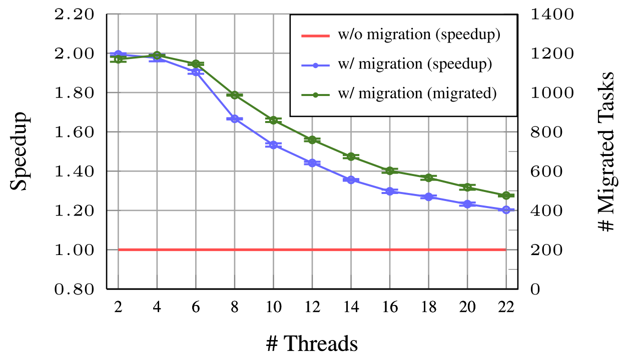


Fig. 5. Number of migrated tasks and speedup achieved by task migration approach for experiments with a matrix size of $S = 150 \times 150$ at different numbers of threads.

and a sufficient overlap with useful computation, i.e., execution of other tasks.

Figs. 4 and 5 show the number of migrated tasks, the execution time and the corresponding speedup that we obtained for an experiment with a fixed matrix size of $S = 150 \times 150$ per task at different numbers of threads. By increasing the number of threads per rank the overall execution time of the program and the work phase decreases as the 2400 tasks are divided across the participating threads. Consequently, the available time to migrate a sufficient number of tasks to balance the load evenly also decreases. We found this to be limiting the overall load balance and therefore the total speedup that can be achieved. A run with 4 threads per rank obtains a speedup of 1.97X as about 1190 tasks could be migrated. For a run with 22 threads, however, the speedup drops to 1.2X as only a lower number of tasks (about 480) could be migrated by the communication threads in the corresponding time frame.

Figs. 6 and 7 illustrate the same performance metrics for a different experiment where we now fix the number of threads but test different matrix sizes. We found that short work phases in combination with small tasks (small in terms of the task's computation work) are problematic, in this case resulting in a performance loss of 0.79X although approximately 200 tasks were correctly migrated to the idle rank. The overhead for the migration of a very small task is much higher than the time required to execute it. Migrating such a small task with a certain delay in time in the work phase can lead to additional latencies and possible performance declines. For instance, there may be additional latencies for receiving the task and its resulting output data. Nevertheless, reactive load balancing can also work well with small tasks if the work phase is sufficiently long and it is guaranteed that the migration overhead can be completely overlapped with the execution of other tasks, e.g., if the underlying imbalance is lower. Increasing the computational complexity and size of the tasks, however, results in a better ratio between execution time and the time required to migrate a task. A run with a matrix size of $S = 450 \times 450$ therefore obtains a speedup of 1.84X by migrating 1122 tasks.

Both effects identified in this experiment confirm that the ratio between the rate at which tasks can be migrated (which eventually will also be limited by the hardware or the network) and the rate at which tasks are processed is crucial, particularly when high load imbalances are present.

Experiment 4: Evaluation with a realistic AMR application

Next, we perform tests to demonstrate that reactive task migration can improve performance of realistic applications employing AMR. We use the sam(oa)^2 framework for parallel adaptive mesh refinement [15] to simulate the tsunami arising from

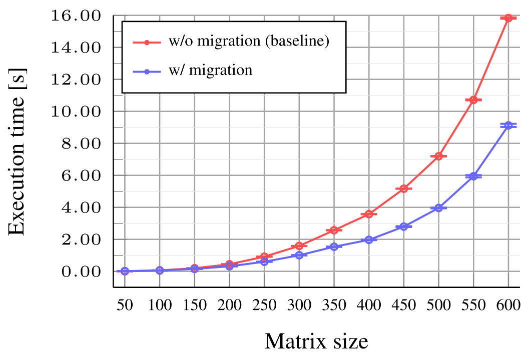


Fig. 6. Execution time for experiments with 16 threads and different matrix sizes.

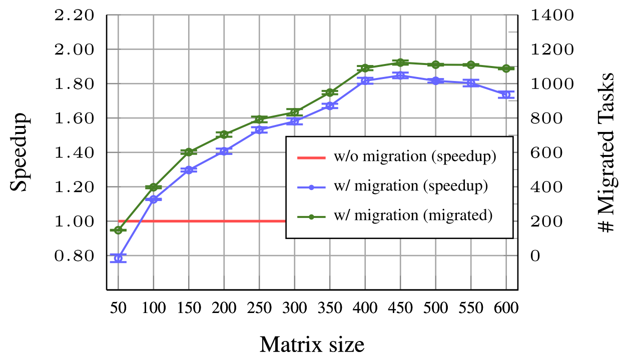


Fig. 7. Number of migrated tasks and speedup achieved by the task migration approach for experiments with 16 threads and different matrix sizes.

the Tōhoku earthquake in 2011[4] with a static initial displacement [6]. To provoke work imbalances, we disabled the application-level load balancing and benchmark against our reactive load balancing library instead. Fig. 8 presents the degree of work imbalance that changes during the simulation time for a run with and without reactive load balancing on 32 nodes. As illustrated, task migration can help to reduce the emerging imbalance but is not capable of completely eliminating it. Fig. 9 shows the strong scaling results on up to 32 nodes where we tested the victim selection strategies depicted in Fig. 1. We find that using reactive load balancing improves scalability despite using one core less for computation. The sort-based victim selection

[4] Data from: http://www.gebco.net/data_and_products/gridded_bathymetry_data/gebco_30_second_grid.

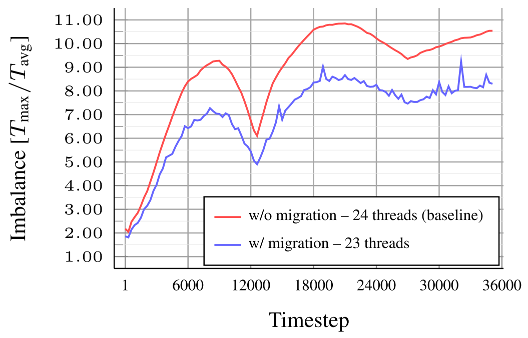


Fig. 8. Degree of work imbalance over time due to AMR for the Tohoku scenario using 32 nodes.

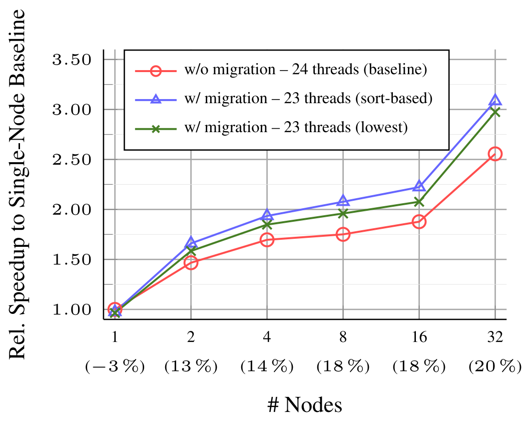


Fig. 9. Strong scaling results for the Tohoku scenario with and without reactive load balancing. Speedup compared to the base line is depicted below corresponding number of nodes.

outperforms the strategy where the rank with the lowest load is selected as a victim. Speedups of up to 1.20X are obtained with our approach relative to the baseline.

Experiment 5: Reactivity for unpredictable imbalances in the numerical scheme

In this experiment, we demonstrate that reactive load balancing can help to balance unpredictable load imbalances in a numerical scheme that cannot be treated well with traditional domain decomposition-based predictive load balancing.

Our benchmark application is the sam(oa)^2 framework where we now employ the ADER-DG numerical time stepping scheme [9] with a limiter [10,22]. We benchmark the implementation presented in [18] for our tests which we extended to support our reactive load balancing library. Load balancing for ADER-DG with limiting is particularly challenging as the cost of processing a grid cell cannot be accurately predicted. In fact, depending on the scenario and its evolution during the simulation, a grid cell may be computed with either the finite volume numerical scheme, the ADER-DG scheme or both if the solution would otherwise be unstable. Due to this dynamic behavior, a cost model to steer a new load re-partitioning of the grid cannot be derived.

We benchmark performance for the oscillating lake scenario [2] on a uniform grid at polynomial order 4 with 169,869,312 cells as shown in Fig. 10. Depending on the cell location in space and on the progress of the simulation in time, a different numerical method needs to be applied. Our baseline without task migration (*w/o migration — cell-based decomposition*) employs domain decomposition based on a uniform cost model

Table 1
Comparison of typical reactive load balancing speedups with the slow-downs obtained with a timing-based problem decomposition for different rank/thread combinations for the oscillating lake scenario. Both speedups are relative to the cell-based decomposition baseline.

Ranks	Threads	Speedup reactive	Slow-down timing-based
16	1	1.06	0.72
32	1	1.05	0.95
64	1	1.09	0.74
128	1	1.07	0.73
16	8	1.05	0.42
32	8	1.02	0.39
64	8	1.05	0.43
128	8	1.00	0.36

per cell as an application-level load balancing scheme: cells are evenly distributed across ranks. We add reactive load balancing on top (*w/ migration — cell-based decomposition*).

We used our reactive load balancing library in order to measure the load imbalance in the ADER-DG scheme. Fig. 11 shows the average load imbalance during 1000 time steps of the oscillating lake scenario on 32 nodes where we vary the number of OpenMP threads available to each of the 32 processes.

During these tests, we found average imbalances up to a factor of 1.3 caused by the numerical scheme in our baseline. With an increasing number of threads, load imbalance decreases. A more fine-granular grid decomposition on the thread level (resulting in more fine-granular tasks) and work stealing between threads result only in small imbalances at the process level. Yet, reactive load balancing continuously helps to mitigate the remaining imbalances between ranks.

In Fig. 12, we plot the measured load imbalance per time step during a simulation run with and without reactive load balancing. During the first 300 time steps, the load imbalance in the baseline decreases as a result of the evolution of our benchmark scenario. Reactive task migration achieves a stable improved load balance in this scenario even in the presence of such dynamic effects.

Finally, we evaluated whether a traditional *timing-based domain decomposition* may help to mitigate the imbalances. In this load balancing approach, sam(oa)^2 instruments the grid processing times for each grid traversal. Based on these measurements, a predictive load balancing approach re-partitions the grid if the imbalance surpasses a user-defined threshold. Load re-partitioning can be conducted in every time step.

Table 1 compares our reactive load balancing approach with the application-level predictive timing-based domain decomposition approach. We found severely degraded performance for the latter: the load predictions do not accurately reflect the future cost of ADER-DG. In fact, expensive grid re-partition steps are frequently conducted that result in less balanced partitionings in the following computation steps. In contrast, we obtain slight speedups of up to 1.09X with reactive load balancing in line with the small observed imbalances in the baseline. Exactly at this point where load predictions fail, temporary reactive task migration mitigates small imbalances on a fine-granular level effectively without the expensive overhead of an application-level re-partitioning.

5. Related work

The literature describes several approaches to mitigate effects of load imbalance. Shared memory runtime systems such as Cilk [3], TBB [19] and several OpenMP [16] implementations apply work stealing to dynamically balance load between threads in shared-memory only.

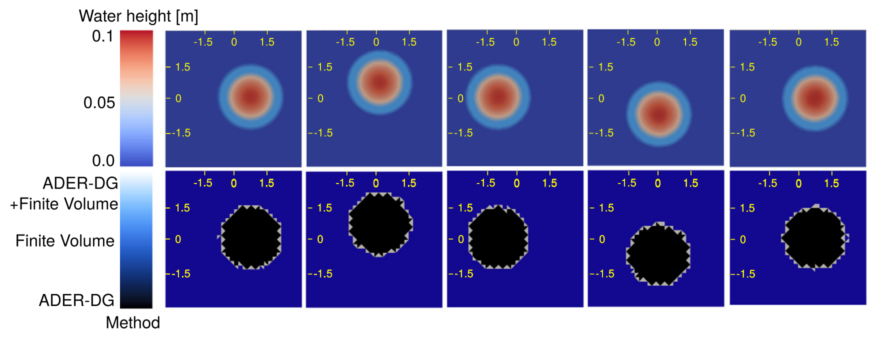


Fig. 10. Simulation of the oscillating lake scenario with ADER-DG for various time steps. *Source:* Adapted from [18].

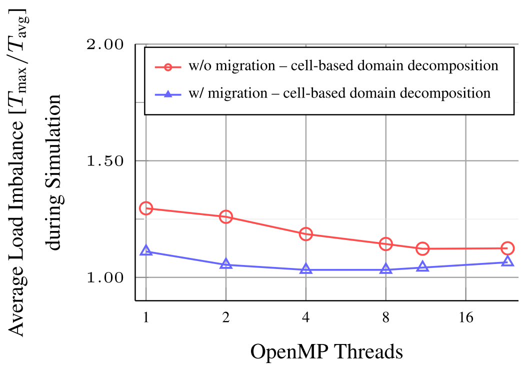


Fig. 11. Average load imbalance at different numbers of threads per rank for the oscillating lake scenario on 32 nodes.

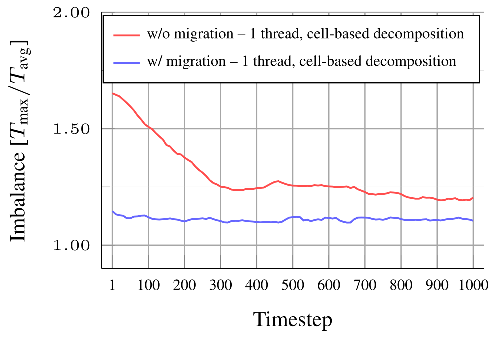


Fig. 12. Load imbalance per time step due to ADER-DG for the oscillating lake scenario on 32 nodes.

In distributed memory, producer–consumer patterns or global repartitioning of work or data (e.g., [17]) are common application-level load balancing approaches. However, they typically induce higher overhead for message and data transfer between processes. Additionally, to the best of our knowledge they usually act on a rather coarse-grained level and require defined synchronization points where load migration is triggered. As an example, distributed runtime systems such as Charm++ [13] enable work re-distribution in distributed memory. Charm++ employs a message-driven runtime model using migratable objects called chares that are assigned to a processing unit. Messages can be sent between chares resulting in an asynchronous execution of functions that is comparable with task executions. Although Charm++ provides means to either manually trigger load balancing or perform it automatically after a configurable time interval, this phase is running exclusively. Another difference to *CHAMELEON* is that Charm++ performs permanent migration, i.e., chares that are migrated to a different processing unit remain there and will be executed by that unit at least until a potential new re-balancing step is executed. This is an efficient solution when dealing with coarse-grained and rather persistent load imbalances. In contrast to that, *CHAMELEON* targets fine-grained imbalances by *temporarily* migrating tasks to underloaded ranks and aims at overlapping migration communication with execution of other tasks to hide the overhead and reduce waiting times. Experiments indicate that Charm++ is well-suited for iterative applications in case of a similar workload distribution for the next iteration but Charm++ does not seem to be able to efficiently compensate unpredictable imbalances [5].

Dinan et al. [8] introduce a scalable work stealing implementation based on the PGAS model provided by the Aggregate Remote Memory Copy Interface (ARMCI) where idle processes randomly steal work from victim processes. They present promising results for a particular class of recursive task-parallel applications with high efficiencies at scale. However, work stealing and migration of tasks do not start until processes run idle. Consequently, communication required to migrate tasks cannot be overlapped with computation. That is one of the reasons why we decided to switch from a pull oriented work stealing prototype [20] to a push oriented reactive task migration mechanism in *CHAMELEON*.

6. Conclusion & future work

In this paper, we presented *CHAMELEON*, a library for reactive load balancing across process boundaries for hybrid task-parallel applications. We demonstrated how continuous introspection, a (possibly user-defined) migration strategy and a task-based execution environment interplay in order to effectively balance the load in distributed memory at run time. Our results show performance improvements up to 1.31X for hardware-induced imbalances and 1.20X for work-induced imbalances using a realistic application with AMR. Our approach is minimally invasive in that it builds upon the established programming models MPI and OpenMP and requires little code modifications, facilitating the integration into existing MPI+OpenMP applications. Our library is designed to tackle fine-granular and unpredictable imbalances by temporarily migrating tasks to other processes while any communication overhead is hidden. We discussed factors that affect or limit performance and load balancing capabilities, and we demonstrated that task migration is even capable of reducing larger work imbalances (see Figs. 5 and 8) as well as unpredictable work imbalances coming from the numerical scheme (see Figs. 11 and 12). However, for applications where a cost model can be derived for predictive application-level load balancing, we recommend combining both predictive and reactive approaches: for instance, the computational domain could be re-partitioned every x iteration step to account for large load imbalances while our reactive approach allows targeting emerging fine-granular and potentially unpredictable imbalances in between.

There are multiple natural directions for future work. Our present model assumes that tasks are independent from each other. A more general approach would allow for dependencies between tasks, rendering the decision-making of when and where to offload tasks more complicated. It is an ongoing research question how fine-granular tasking in distributed memory with dependencies can be implemented effectively.

Moreover, we are currently exploring another reactive load balancing mechanism that makes use of *task replication*. Keeping tasks replicated on multiple MPI ranks and deciding at run time which rank computes a replicated task would allow us to further boost reactivity and help mitigate potential issues arising when too many tasks have been migrated to a victim rank.

Finally, we strive to broaden the class of applications benefiting from our approach.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.12.005.

Acknowledgments

We like to thank Leonhard Rannabauer for his very helpful application support on ADER-DG in sam(*oa*)^2. Further, we like to thank Marcus Wagner and Sascha Bücken for the excellent technical support and administrative help during our evaluation phase.

Some of the experiments were performed with computing resources granted by JARA-HPC from RWTH Aachen University, Germany under projects jara0001 and nova0027. Parts of this work were funded by the German Federal Ministry of Education and Research (BMBF) under Grant Nos. 01IH16004B and 01IH16004C (Project Chameleon).

References

[1] B. Acun, P. Miller, L.V. Kale, Variation among processors under turbo boost in HPC systems, in: Proceedings of the 2016 International Conference on Supercomputing, ICS '16, ACM, New York, NY, USA, 2016, pp. 6:1–6:12, http://dx.doi.org/10.1145/2925426.2926289.

[2] E. Audusse, F. Bouchut, M.-O. Bristeau, R. Klein, B. Perthame, A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows, SIAM J. Sci. Comput. 25 (2004) http://dx.doi.org/10.1137/S1064827503431090.

[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95, ACM, New York, NY, USA, 1995, pp. 207–216, http://dx.doi.org/10.1145/209936.209958.

[4] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, A. Fedorova, Evaluation of the Intel® Core™ i7 turbo boost feature, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC, IISWC '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 188–197, http://dx.doi.org/10.1109/IISWC.2009.5306782.

[5] S. Convent, Investigating Scheduling and Load Balancing Characteristics of Task-based Programming Models for Hybrid HPC Applications, RWTH Aachen University, Aachen, 2019, http://dx.doi.org/10.18154/RWTH-2019-09023.

[6] L.A. Dalguer, P. Galvez, J.-P. Ampuero, S.N. Somala, T. Nissen-Meyer, Dynamic earthquake rupture modelled with an unstructured 3-D spectral element method applied to the 2011 M9 Tohoku earthquake, Geophys. J. Int. 198 (2) (2014) 1222–1240, http://dx.doi.org/10.1093/gji/ggu203, arXiv:http://oup.prod.sis.lan/gji/article-pdf/198/2/1222/1651569/ggu203.pdf.

[7] A. Denis, J. Jaeger, H. Taboada, Progress thread placement for overlapping MPI non-blocking collectives using simultaneous multi-threading, in: G. Mencagli, D.B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R.R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J.D. Garcia Sanchez, S.L. Scott (Eds.), Euro-Par 2018: Parallel Processing Workshops, Euro-Par 2018, Springer International Publishing, 2019, pp. 123–133, http://dx.doi.org/10.1007/978-3-030-10549-5_10.

[8] J. Dinan, D.B. Larkins, P. Sadayappan, S. Krishnamoorthy, J. Nieplocha, Scalable work stealing, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, 2009, pp. 1–11, http://dx.doi.org/10.1145/1654059.1654113.

[9] M. Dumbser, D.S. Balsara, E.F. Toro, C.-D. Munz, A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes, J. Comput. Phys. 227 (18) (2008) 8209–8253, http://dx.doi.org/10.1016/j.jcp.2008.05.025, URL http://www.sciencedirect.com/science/article/pii/S0021999108002829.

[10] M. Dumbser, R. Loubère, A simple robust and accurate a posteriori subcell finite volume limiter for the discontinuous Galerkin method on unstructured meshes, J. Comput. Phys. 319 (2016) 163–199, http://dx.doi.org/10.1016/j.jcp.2016.05.002, URL http://www.sciencedirect.com/science/article/pii/S0021999116301292.

[11] T. Hoefler, A. Lumsdaine, Message progression in parallel computing – To thread or not to thread? in: Proceedings - IEEE International Conference on Cluster Computing, ICCC, Vol. Proceeding, 2008, pp. 213–222, http://dx.doi.org/10.1109/CLUSTR.2008.4663774.

[12] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, I. Miyoshi, Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 78:1–78:12, http://dx.doi.org/10.1145/2807591.2807638.

[13] L.V. Kale, S. Krishnan, CHARM++: A portable concurrent object oriented system based on C++, SIGPLAN Not. 28 (10) (1993) 91–108, http://dx.doi.org/10.1145/167962.165874.

[14] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, M.S. Müller, Reactive task migration for hybrid MPI+OpenMP applications, in: Parallel Processing and Applied Mathematics, PPAM 2019, Springer International Publishing, 2020, (in press).

[15] O. Meister, K. Rahnema, M. Bader, Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells, ACM Trans. Math. Software 43 (3) (2016) 1–27, http://dx.doi.org/10.1145/2947668.

[16] OpenMP Architecture Review Board, OpenMP application program interface, Version 5.0, 2018, http://www.openmp.org/. (November 2018).

[17] A. Pinar, C. Aykanat, Fast optimal load balancing algorithms for 1D partitioning, J. Parallel Distrib. Comput. 64 (8) (2004) 974–996, http://dx.doi.org/10.1016/j.jpdc.2004.05.003.

[18] L. Rannabauer, M. Dumbser, M. Bader, ADER-DG with a-posteriori finite-volume limiting to simulate tsunamis in a parallel adaptive mesh refinement framework, Comput. & Fluids 173 (2018) 299–306, http://dx.doi.org/10.1016/j.compfluid.2018.01.031.

[19] J. Reinders, Intel Threading Building Blocks, first ed., O'Reilly & Associates, Inc., 2007.

[20] P. Samfass, J. Klinkenberg, M. Bader, Hybrid MPI+OpenMP reactive work stealing in distributed memory in the PDE framework sam(oa)^2, in: 2018 IEEE International Conference on Cluster Computing, CLUSTER, CLUSTER '18, IEEE, 2018, pp. 337–347, http://dx.doi.org/10.1109/CLUSTER.2018.00051.

[21] J. Treibig, G. Hager, G. Wellein, LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments, in: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, 2010.

[22] O. Zanotti, F. Fambri, M. Dumbser, A. Hidalgo, Space–time adaptive ADER discontinuous Galerkin finite element schemes with a posteriori sub-cell finite volume limiting, Comput. & Fluids 118 (2015) 204–224, http://dx.doi.org/10.1016/j.compfluid.2015.06.020, URL http://www.sciencedirect.com/science/article/pii/S0045793015002030.

Jannis Klinkenberg is a Ph.D. candidate at the Chair for High Performance Computing at RWTH Aachen University. He received a B.Sc. degree in Scientific Computing from FH/RWTH Aachen in 2010 and a M.Sc. degree in Artificial Intelligence from Maastricht University in 2012. His research interests include runtime improvements for dynamic and heterogeneous systems, parallel and task-based programming models and the optimization of scientific workloads for modern HPC platforms. Since 2016, he is also actively participating in the OpenMP Language Committee.

Philipp Samfass is a Ph.D. candidate at the Chair of Scientific Computing in Computer Science at Technical University of Munich (TUM). He holds a B.Sc. degree in Computer Science from TUM and a M.Sc. degree in Computer Science from the University of Illinois at Urbana–Champaign. His research focuses on developing new algorithmic approaches for upcoming challenges in high performance computing such as load balancing, fault tolerance and hybrid parallel programming. He is funded in part by the BMBF (Chameleon project) and the EU (ExaHyPE project).

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

Christian Terboven received a Doctor of Natural Sciences degree from RWTH Aachen University, Germany. He leads the HPC Group at RWTH Aachen University, and his research interests include parallel programming models, related software engineering aspects, and the optimization of simulation codes for modern HPC architectures. Since 2006, he serves on the OpenMP Language Committee and is Chair of the Affinity Subcommittee.

Matthias S. Mueller is full professor for High Performance Computing at RWTH Aachen University and head of the Computation and Communication Center at RWTH. His research interests include programming methodologies, software development tools and computational science on high performance computers. He received his Ph.D. in Physics from Stuttgart University in 2001. From 1999 to 2005 he worked at the High Performance Computing Center (HLRS) in Stuttgart, Germany, which he left as a deputy director. From 2005 until 2012 he was deputy director and CTO at the Center for Information Services and High Performance Computing (ZIH) at Technical University of Dresden.