Scalable High-Quality 1D Partitioning

Matthias Lieber, Wolfgang E. Nagel
Technische Universität Dresden, 01062 Dresden, Germany
{matthias.lieber, wolfgang.nagel}@tu-dresden.de

***Abstract*—The decomposition of one-dimensional workload arrays into consecutive partitions is a core problem of many load balancing methods, especially those based on space-filling curves. While previous work has shown that heuristics can be parallelized, only sequential algorithms exist for the optimal solution. However, centralized partitioning will become infeasible in the exascale era due to the vast amount of tasks to be mapped to millions of processors. In this work, we first introduce optimizations to a published exact algorithm. Further, we investigate a hierarchical approach which combines a parallel heuristic and an exact algorithm to form a scalable and high-quality 1D partitioning algorithm. We compare load balance, execution time, and task migration of the algorithms for up to 262 144 processes using real-life workload data. The results show a 300 times speed-up compared to an existing fast exact algorithm, while achieving nearly the optimal load balance.**

Keywords*—*High performance computing, Dynamic load balancing, One-dimensional partitioning, Hierarchical partitioning, Scalability

I. Introduction

Load balance is one of the major challenges for efficient use of current and future HPC systems [1], [2], especially when the workload is changing dynamically. Many scientific simulations exhibit workload variations due to adaptive spatial grids [3] or adaptive time stepping techniques [4], [5]. An additional source of workload variations are descriptions of physical or chemical phenomena, whose runtime depend locally on variables of the model, like detailed atmospheric simulations [6], [7] and particle simulations [8], [9].

In this work we focus on computational scenarios with a fixed number of work items (tasks) of spatially and temporally varying workload that need to be distributed over a large number of processes. The assignment of tasks to processes needs to be adapted periodically over runtime to ensure high balance. The calculation of a new assignment is called *repartitioning*, which has to achieve the following objectives [10]: (a) balanced workload in the new partitioning, (b) low communication costs between distributed tasks due to data dependencies within the application, (c) low migration costs, and (d) fast execution of the repartitioning. Typically, these objectives are contradictory. For example, the optimal solution for (a) and (b) is known to be NP-complete, which makes objective (d) hard to reach. The balance between the four objectives therefore depends on the application, e. g. highly dynamic applications should focus on low migration and repartitioning costs and accept a non-optimal load balance. Many heuristics have been developed for (re)partitioning; Teresco et al. [10] provide a good overview.

A widely used method is space-filling curve (SFC) partitioning [10], [11]. It is applied for scalable adaptive mesh refinement [3] and dynamic load balancing of a fixed number of tasks with varying workload [4], [8]. In general, SFCs provide a fast mapping from n-dimensional to one-dimensional space that preserves spatial locality. This property is used to reduce the partitioning problem to one dimension. In comparison to rectangular partitioning methods, SFC partitioning has the advantages of better load balance due to finer granularity (no restriction to rectangular partitions) and highly local, low-degree task migration, which has a one-dimensional logical structure according to the curve [11]. With respect to the four objectives of dynamic load balancing, SFCs are a good heuristic to implicitly optimize for low communication costs. The remaining three objectives have to be handled by 1D partitioning algorithms. Published 1D partitioning heuristics execute very quickly and can be implemented in parallel [11], but they do not achieve the optimal load balance. On the other hand, exact algorithms [12] are, to the best of our knowledge, sequential only. However, sequential algorithms will fail in the exascale era due to large communication, memory, and calculation costs. This work closes the gap between exact algorithms and parallel heuristics. The main contributions are:

- Investigation of runtime optimizations for a published exact 1D partitioning algorithm.
- A new hierarchical and highly scalable 1D partitioning algorithm that provides nearly optimal balance.
- Experimental evaluation of existing and proposed algorithms comparing load balance, migration costs, and calculation time using real-life workload data.

The rest of the paper is organized as follows: In the next section we define the 1D partition problem. Then, in Sec. III, we give an overview of related work about 1D partitioning. Sec. IV introduces the benchmark and systems we use for experimental evaluation of the algorithms, which are described and evaluated in Sections V and VI.

II. The 1D Partitioning Problem

In the 1D partitioning problem, a vector w_i, $i = 1, 2, \dots, N$, of positive task weights, representing N computational loads, is to be decomposed into P consecutive partitions while minimizing the maximum load among the partitions. This problem is also referred to as the chains-on-chains partitioning problem [12]. The result is a partition vector s_p that denotes the first task in w_i assigned to partition p, with $p = 0, 1, \dots P-1$. Note, that each partition contains a contiguous subset of tasks. The load of partition p is determined with $L_p = \sum_{i=s_p}^{s_{p+1}-1} w_i$. Alternatively, the load can be computed as $L_p = W_{s_{p+1}-1} - W_{s_p-1}$ using the prefix sum of task weights $W_j = \sum_{i=1}^{j} w_i$, $j = 1, 2, \dots, N$ and $W_0 = 0$. The maximum load among all partitions

TABLE I. SUMMARY OF SYMBOLS.

B	bottleneck of a partitioning, i. e. maximum load among all partitions, $B = max(L_p)$
B^*	ideal bottleneck, $B^* = \Sigma w_i/P$
B^{opt}	bottleneck of the optimal partitioning
G	number of coarse partitions of the hierarchical algorithm, $2 \leq G \leq P/2$
L_p	load of partition p, i. e. sum of its task weights
Λ	balance of a partitioning, $\Lambda = B^*/B$
Λ^{opt}	balance of the optimal partitioning, $\Lambda^{opt} = B^*/B^{opt}$
N	number of tasks to assign to the partitions
P	number of partitions (i. e. parallel processes)
q	quality factor of a partitioning, $q = \Lambda/\Lambda^{opt}$
s_p	partition vector, i. e. index of the first task assigned to partition p for $p = 0, 1, \ldots, P-1$; $s_0 = 1$
w_i	computational weight of task i for $i = 1, 2, \ldots, N$
W_j	prefix sum of task weights, $W_j = \sum_{i=1}^{j} w_i$; $W_0 = 0$

$B = max(L_p)$ is called the bottleneck of a partitioning. The objective of 1D partitioning is to find a partition vector s_p with the minimal bottleneck B^{opt}, which is not known a priori. The lower bound for any B is the ideal bottleneck $B^* = \Sigma w_i/P = W_N/P$, which assumes equal load among all partitions. We define the ratio of ideal bottleneck B^* to the bottleneck B of a partitioning as the load balance Λ of this partitioning, i. e. $\Lambda = B^*/B$ with $1/P \leq \Lambda \leq 1$. The optimal load balance Λ^{opt} of a given 1D partitioning problem is $\Lambda^{opt} = B^*/B^{opt}$ and the quality factor q of a partitioning is $q = \Lambda/\Lambda^{opt} = B^{opt}/B$, which follows the definition by Miguet and Pierson [13]. One important property of the task weights w_i is their maximum $max(w_i)$, since perfect balance cannot be achieved if $max(w_i) > B^* = \Sigma w_i/P$. In this case applies $B^{opt} \geq max(w_i)$ and further parallelism will not decrease the bottleneck. Thus, well-balanced partitionings are only achievable if $P \leq \Sigma w_i/max(w_i)$. Table I summarizes the introduced symbols.

III. RELATED WORK

One of the first 1D partitioning heuristics for SFC-based load balancing is described by Oden et al. [14]. They use the recursive bisection approach where the weight vector is recursively cut in two parts with as equal as possible load. Pilkington and Baden [11] introduce a parallel heuristic. The processes search their new partition boundaries within the local part of the weight vector prefix sum W_j and within the part of direct neighbors along the curve. Of course, this only works so long as the partition borders do not shift across the neighbor processes. Miguet and Pierson [13] describe two heuristics and their parallelization and provide a detailed discussion about the costs and quality bounds of the algorithms. Their first heuristic *H1* computes s_p to be the smallest index such that $W_{s_p} > pB^*$. The second heuristic *H2* refines the partition boundaries found by *H1* by incrementing s_p if $(W_{s_p} - pB^*) < (pB^* - W_{s_p-1})$, i. e. if the cumulated task weight W_{s_p} is closer to the border's ideal cumulated task weight pB^* than W_{s_p-1}. They also prove that for their heuristics the bottleneck is bounded by $B < B^* + max(w_i)$, which means that these algorithms are very close to the optimal solution if $max(w_i) \ll B^*$. However, this yields the tightened requirement for well-balanced partitionings $P \ll \Sigma w_i/max(w_i)$ compared to $P \leq \Sigma w_i/max(w_i)$ introduced for the general case in Sec. II.

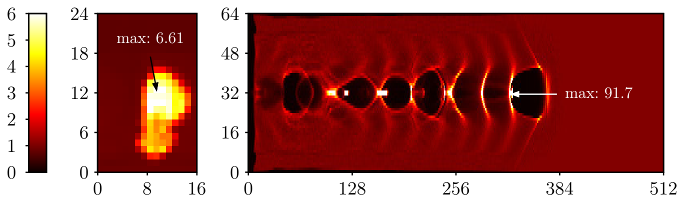
Much work has been published on exact algorithms for the 1D partition problem; a very extensive overview is given by Pınar and Aykanat [12]. They provide detailed descriptions of existing heuristics and exact algorithms, improvements and new algorithms, as well as a thorough experimental comparison. However, they only consider sequential algorithms. The fastest exact algorithm proposed by Pınar and Aykanat is the exact bisection algorithm *ExactBS*. It is based on the binary search for the optimal bottleneck B^{opt}. The initial search interval is $I = [B^*, B^{RB}]$, where B^{RB} is the bottleneck achieved by the recursive bisection heuristic. To guide the binary search, it is required to probe whether a partitioning can be constructed for a given B or not. The *Probe* function successively assigns each partition $p = 0, 1, \ldots P-2$ the maximum number of tasks such that the partition's load is not larger than B. *Probe* is successful if the load of the remaining partition $P-1$ is not larger then B. A simple probing method using binary search on W_j for each s_p has $O(P \, log \, N)$ complexity. Han et al. [15] propose an improved *Probe* with $O(P \, log(N/P))$ complexity which partitions W_j in P equal-sized segments. For each s_p to be found, first the segment containing s_p is determined using linear search and then binary search is used within the segment. Pınar and Aykanat [12] also use binary search for probing, but they further restrict the search space by keeping record of the lowest and highest values found for each s_p in earlier steps of the search for B^{opt}. Pınar and Aykanat show that the complexity of their restricted probe function is $O(P \, log(P) + P \, log(max(w_i)/avg(w_i)))$, which is very attractive for large N. For a detailed description of *ExactBS* we refer to the original publication [12].

To the best of our knowledge, no parallel exact algorithms for the 1D partition problem have been published. Parallel heuristics can be used in large-scale applications requiring frequent load balancing. However, as Miguet and Pierson [13] have shown, the load balance is only close to optimal as long as $max(w_i) \ll B^*$. Current trends suggest that this condition will be fulfilled less often in future [1], [2]: Firstly, simulations incorporate more and more complex phenomena and adaptivity, giving rise to workload variations and thus increasing the maximum task weight $max(w_i)$ stronger than the average load $B^* = \Sigma w_i/P$. Secondly, the parallelism in HPC systems is growing greatly, which leads to strong scaling replacing increasingly weak scaling and thus to a reduction of B^*. Consequently, scalable and high-quality partitioning algorithms are required for many future large-scale simulations.

One solution for the scalability challenge is the application of hierarchical methods for load balancing. Zheng et al. [16] investigate such methods in the runtime system Charm++. They organize processes in a tree hierarchy and use centralized partitioning methods within each level and group independently. The main advantages of their approach are considerable memory savings due to data reduction strategies and faster execution of the partitioning at large scale.

IV. EVALUATION BENCHMARK

We have developed an MPI-based benchmark to compare existing 1D partitioning algorithms with our methods. Like in typical applications, the task weights are only known to the process owning the task. This distributed task weight vector is input to the algorithms. The output is the partition vector s_p, which should be replicated on each process. Following existing algorithms are compared to our new methods:



(a) CLOUD Dataset (b) LWFA Dataset

Fig. 1. Visualization of workload on a slice through the center of the 3D computational domain. The workload is shown relative to the average. The most imbalanced time step of each dataset is shown.

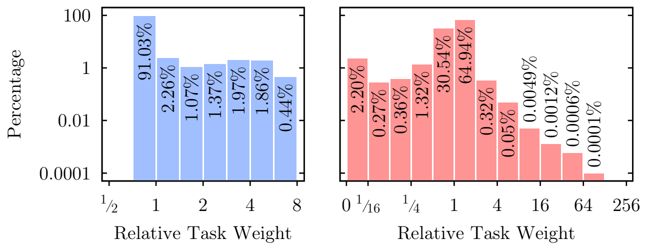
- Exact algorithm *ExactBS* by Pınar and Aykanat [12]:
  1) Parallel prefix sum of weights w_i using MPI_Exscan, determination of $max(w_i)$ on rank 0 using MPI_Reduce
  2) Collection of prefix sum W_j on rank 0 via MPI_Gatherv
  3) Serial execution of *ExactBS* on rank 0
  4) Distribution of partition vector s_p with MPI_Bcast
- Serial heuristic *H2* of Miguet and Pierson [13]:
  1) Parallel prefix sum of weights w_i using MPI_Exscan
  2) Collection of prefix sum W_j on rank 0 via MPI_Gatherv
  3) Serial execution of *H2* on rank 0 using the *Probe* algorithm by Han et al. [15]
  4) Distribution of partition vector s_p with MPI_Bcast
- Parallel version of *H2*:
  1) Parallel prefix sum of weights w_i using MPI_Exscan
  2) Point-to-point communication of first local value in W_j to $rank - 1$ (to ensure consistency when using floating point weights) and communication of total weight from last rank to all via MPI_Bcast
  3) Execution of *H2* on local part of W_j
  4) Each found border s_p is sent to rank p, final distribution of partition vector to all processes with MPI_Allgather

The benchmark determines the runtime of each phase of the partitioning algorithm and the achieved load balance. To observe the amount of migration, the benchmark iterates over a set of task weight vectors. The task weights are derived from two different HPC applications as described in the following.

A. Real-life Datasets CLOUD and LWFA

The CLOUD dataset is extracted from COSMO-SPECS+FD4 [6], [17], which simulates the evolution of clouds and precipitation in the atmosphere in a high level of detail. In our scenario, a growing cumulus cloud leads to locally increasing workload of the cloud microphysics model. We measured the execution times of $16 \times 16 \times 24 = 6144$ grid blocks for 100 successive time steps. The weight imbalance $max(w_i)/avg(w_i)$ varies between 4.32 and 6.61. Fig. 1 (a) visualizes the weights of the most imbalanced step. To construct larger weight vectors, we replicated the original block weights in the first two (horizontal) dimensions, e. g. a replication of 13×7 results in $208 \times 112 \times 24 = 559\,104$ weights. After this, we used a Hilbert SFC to create task weight vectors.

The second dataset originates from a laser wakefield acceleration (LWFA) simulation with the open source particle-in-cell code PIConGPU [9], [18]. In LWFA, electrons are accelerated by high electric fields caused by an ultrashort laser pulse in a gas jet [19]. The dense accumulation of electrons following the laser pulse leads to severe load imbalances, see Fig. 1 (b). The computational grid consists of $32 \times 512 \times 64 = 1\,048\,576$ supercells whose workload is determined by the number of particles per supercell. We created task weight vectors for 2000 consecutive time steps (out of 10 000) using a Hilbert SFC. The weight imbalance varies between 91.7 and 32.5.



(a) CLOUD Dataset (b) LWFA Dataset

Fig. 2. Histograms of the most imbalanced task weight vectors of both datasets. The weight is specified relative to the average. Note, that the leftmost column in the LWFA chart includes zero weight tasks (i. e. no particles).

TABLE II. DESCRIPTION OF THE BENCHMARK SYSTEMS.

Name	JUQUEEN	SuperMUC (thin nodes)
System	IBM BlueGene/Q	IBM iDataPlex
Processor	IBM PowerPC A2 1.6 GHz	Intel Xeon E5-2680 2.7 GHz
Cores / RAM per node	16 cores / 16 GiB RAM	16 cores / 32 GiB RAM
Total nodes	28 672	18 × 512 = 9216
Total cores	458 752	18 × 8192 = 147 456
Network & Topology	IBM proprietary 5D torus	Infiniband FDR10 tree
Peak PFlop/s	5.872 PFlop/s	3.185 PFlop/s

Fig. 2 shows histograms of the most imbalanced task weight vectors in both datasets. Most of the weights are near the average, except for a few strong peaks. Due to the so-called bubble, a region without electrons behind the laser pulse, the LWFA dataset also contains tasks with zero weight. The standard deviation for the shown relative task weight vectors are 0.747 for CLOUD and 0.305 for LWFA.

B. Benchmark Systems

We performed measurements on two Top10 systems of the November 2013 Top500 list: The IBM BlueGene/Q system JUQUEEN and the IBM iDataPlex system SuperMUC. Their hardware characteristics are compared in Table II. Since both systems support simultaneous multithreading, we used 32 MPI processes per node for our measurements.

V. Improving the Exact Bisection Algorithm

This section describes and evaluates our optimizations for the exact bisection algorithm *ExactBS* [12].

A. Probe Algorithm

In *ExactBS*, the *Probe* function checks whether a partition exists for a given bottleneck B. As introduced in Sec. III, Pınar and Aykanat restrict the search space for each individual s_p by narrowing the search interval in W_j dynamically. We developed a *Probe* algorithm which is faster without search space restriction, if (1) the size of partitions adjacent in the weight vector varies only little, or (2) the number of tasks N is not orders of magnitude higher than the number of partitions P. Our *Probe* algorithm, shown in Fig. 3, starts the search for

```
PROBE ($P$, $N$, $B$, $W$)
    $end := 0$;  $sum := B$;  $guess := N/P$
    for $p := 0$ to $P-2$ do
        if $W_{guess} > sum$ then
            $i := guess - 1$
            while $W_i > sum$ do $i := i - 1$
        else
            $i := guess$
            while $i+1 \leq N$ and $W_{i+1} \leq sum$ do $i := i + 1$
        $guess := min(2i - end, N)$
        if $i = N$ then exit
        $sum := W_i + B$
        $end := i$
    if $W_N \leq sum$ then return $true$
    else return $false$
```

Fig. 3. Our proposed *Probe* algorithm.

the next s_p at $2s_{p-1} - s_{p-2}$, which results in a match if the partition $p-1$ has the same size as partition $p-2$. If this is not the case, we start a linear search ascending or descending in W_j. For relatively small partition sizes, the number of linear search steps will likely be very small and outperform binary search. Consequently, we expect our algorithm to be faster than Pınar and Aykanat's *Probe* at relatively low N/P only.

B. Quality-Assuring Bisection Algorithm QBS

While the first optimization was targeted on the cost of the *Probe* algorithm, this optimization reduces the number of search steps for the optimal bottleneck, i. e. the number of *Probe* calls. Fig. 4 depicts the basic idea: It shows the evolution of the target load balance associated with the bottleneck values *ExactBS* is probing for. In this particular example, 16 binary search steps are required to find the optimal load balance $\Lambda^{opt} = 0.97661$. But already after 6 steps, a balance of $\Lambda = 0.97619$ is achieved, which corresponds to a quality factor of $q = 0.9996$. This quality factor should absolutely suffice for many applications, since task weights are typically estimations and such small deviations from the optimal load balance will hardly affect the runtime of the application. Our optimization utilizes this observation and terminates the binary search as soon as the distance between lower and upper bound for B becomes smaller than ϵ, like in the algorithm ϵBS [12]. We can approximate ϵ from the requested q as follows: From the termination condition we know: $\epsilon \geq B - B^{opt}$. Since $q = B^{opt}/B$, we can formulate $\epsilon \geq (1-q)B$. With $\Lambda = B^*/B$ and $\Lambda \leq q$ we finally obtain:

$$\epsilon \geq \frac{1-q}{q} B^* = \frac{1-q}{q} \frac{W_N}{P}$$

This equation estimates the termination condition of the binary search for a given quality factor q. We call the algorithm incorporating the optimizations of this section and the previous one *QBS* (quality-assuring bisection). Running *QBS* with $q = 1$ results in an exact algorithm, while $q < 1$ results in an approximate algorithm with guaranteed quality. For the initial search interval, we build on the findings of Miguet and Pierson [13] and use $I = [max(B^*, max(w_i)), B^* + max(w_i)]$.

C. Parallel Bisection Algorithm QBS*

We now describe a simple parallelization of *QBS* that can be applied to *ExactBS* as well. Given that the complete prefix

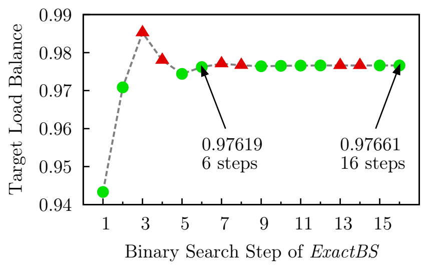


Fig. 4. Exemplary development of the target load balance over the binary search steps of *ExactBS* for one step of the CLOUD dataset (559 104 tasks) and 16 384 processes. Circles denote a successful attempt to construct a partitioning with the indicated balance during *Probe*, triangles a failure.

sum of the weight vector is replicated on K processes, the search interval I for the bottleneck can be split in disjunct, equal-sized parts I_k with $I = \cup I_k$. The processes now search individually in their local parts. To ensure correctness of the termination condition, each process first calls *Probe* for its lower bound of I_k. If this is successful, no further search steps are required in this interval. Otherwise, the binary search is started. Finally, the smallest feasible bottleneck B is determined with MPI_Allreduce and the processes can easily construct s_p from B. The maximum number of search steps among the processes can be reduced at most by a factor of $\lfloor log_2(K) \rfloor$. For *QBS*, it may even happen that binary search is not necessary at all: The algorithm terminates after the first *Probe* for the lower bound of I_k as soon as the length of the interval sections I_k is smaller than ϵ, i. e.:

$$\frac{B^* + max(w_i) - max(B^*, max(w_i))}{K} \leq \epsilon$$

In our benchmark, we implemented parallel *QBS* with fixed $K = P$. This has the downside that W_j has to be distributed to all processes with an MPI_Allgatherv operation, but after the determination of the best B, all processes can construct s_p individually without further communication.

D. Experimental Evaluation of QBS and QBS*

To evaluate our optimizations of *ExactBS*, we ran the benchmark described in Sec. IV with the CLOUD dataset and a replication factor of 13×7 on 16 384 processes. This results in 559 104 tasks and 34.12 tasks per process on average. Fig. 5 shows the results as averages over 100 iterations of the benchmark. The runtime is the measured wall clock time of the 1D partitioning calculation only, i. e. without prefix sum, collection of weights, and broadcast of the partition vector. For the parallel *QBS** algorithm, the runtime is averaged over all processes. The comparison of the results for the existing algorithms *H2seq* and *ExactBS* shows, that the heuristic is clearly faster, but it fails to achieve a sufficient load balance. However, the percentage of migrated tasks per iteration is much higher with the exact algorithm. The reason is that *ExactBS* places the partition borders depending on all individual values in W_j. In contrast, the placement in *H2seq* mainly depends on B^*, which varies much less between the iterations. The results for *QBS* with $q = 1$ show that the improved *Probe* algorithm leads to a 2.7 times speed-up over *ExactBS*. The method *QBS* with $q < 1$ allows to reduce the load balance target in a controlled fashion with further reduction of the runtime. However, the task migration increases with decreasing

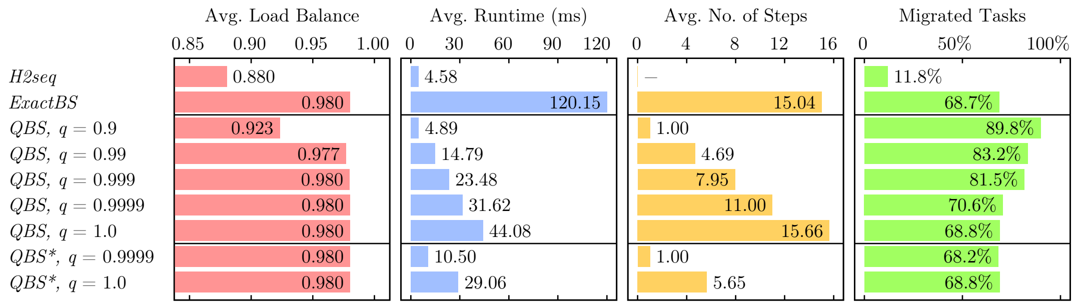


Fig. 5. 1D partitioning benchmark results for the sequential algorithms *H2seq*, *ExactBS*, and *QBS* with the CLOUD dataset (559 104 tasks) for 16 384 processes on JUQUEEN. *QBS** is a version with parallelized search for the optimal bottleneck. The runtime includes the 1D partitioning calculation only.

q, because the achieved bottleneck varies over the interval $[B^{opt}, B^{opt}/q]$ between successive iterations, which leads to larger shifts in the partition border placement. Finally, *QBS** accomplishes a further runtime reduction. With $q = 0.9999$, binary search is not required in any iteration so that probing is required only once per interval. However, the collection of task weights on all ranks, as required by *QBS**, is much more expensive than gathering of task weights in rank 0 only. If we compare the complete runtime of *QBS* and *QBS**, including prefix sum, collection of weights, and broadcast of partition vector, *QBS* is more than 4 times faster than *QBS** in this specific case.

VI. Improving Scalability with a Hierarchical Algorithm

At large scale, the collection of all task weights at one or multiple processes is infeasible due to memory limitations and high communication costs. Only heuristics, like those of Miguet and Pierson [13], can by parallelized completely such that the task weights are evaluated locally only. In this section, we propose a two-level hierarchical method [20] that combines the distributed computation of the heuristic *H2* with the high quality of the *QBS* algorithm.

A. Design of the Hierarchical Algorithms HIER and HIER*

The basic idea picks up the condition $max(w_i) \ll B^*$ for almost optimal partitionings computed by *H2*. If we partition the tasks not in P but $G < P$ parts, B^* would be increased and the condition could be met easier. In our hierarchical approach, we use this property to first create a coarse-grained partitioning in G parts with a fully parallel heuristic. Each of the G coarse partitions is assigned a group of processes. Second, we decompose each coarse part in P/G partitions using the exact methods *QBS* or *QBS** with $q = 1$. In the second phase, G instances of the exact method are running independently to each other and task weights need only to be collected within the groups, i. e. no vector of all task weights needs to be assembled. The number of groups G highly impacts quality and performance of the hierarchical method; it is actually like a slide control which allows to tune the influence of the heuristic versus the exact method. In the following, we provide a more detailed description of the methods *HIER* and *HIER**:

1) Prefix sum of weights and broadcast of total load: The prefix sum of task weights is computed in parallel using MPI_Exscan with the sum of local weights as input. Then all ranks $p > 0$ send $W_{s_p - 1}$ to $rank - 1$ to ensure consistency at the partition borders when using floating point weights. Finally, the total load W_N, which is available in the last process, is communicated to all ranks via MPI_Bcast.

2) Construction of the coarse partitioning: All processes search in their local part of W_j for coarse partition borders using the method *H2* with $B^* = W_N/G$. If a process finds a border, it sends the position to the group masters (first ranks) of both groups adjacent to that border. The group masters broadcast them to all processes within the group.

3) Collection of task weights within the groups: All processes owning tasks that are not part of their coarse partition send the respective W_j to the nearest process of the group that owns these tasks in the coarse partitioning. Then, the (prefix-summed) task weights are exchanged within each group independently using MPI_Gather in *HIER*, such that the master receives W_j for its group, or MPI_Allgather in *HIER**, such that all ranks in the group receive W_j.

4) Exact partitioning within the groups: Based on the local prefix sum of the weight vector for the group, the final partitioning is computed with *QBS* or *QBS** in *HIER* or *HIER**, respectively, using $q = 1$. Now, G instances of the exact method are running independently to each other.

5) Distribution of the partition vector: The final partition vector is communicated to all ranks in a two-stage process: First, the group masters assemble the global vector by exchanging the partition vector of their group among each other using MPI_Allgather. Second, the masters distribute the global partition vector to their group members via MPI_Bcast.

B. Quality Bounds of the Hierarchical Algorithm

The load balance achieved with our hierarchical algorithm is limited by the initial coarse partitioning. Even if the initial partitioning was perfect (i. e. each group has exactly the same load) non-optimal results can be achieved if the optimal bottlenecks B^{opt} of the individual group partitionings vary. Of course, the quality of *HIER* is never worse than the quality of *H2*, since the coarse partition borders are also borders in *H2*, but *HIER* runs an optimal method for the rest of the borders. Miguet and Pierson [13] have shown that the quality factor of *H2* is $q \geq 1/2$. We can construct an artificial case where this lower bound is reached for *HIER*: Let $N \gg P$, $w_i = w_H$ for $i = 1, 2, \ldots, \frac{P}{2} + 1$ and $w_i = 1$ for $i = \frac{P}{2} + 2, \frac{P}{2} + 3, \ldots, N$ with w_H as the maximum weight such that $s_{P/2} = \frac{P}{2} + 2$ in *H2*, i. e. the first half of the partitions contains $\frac{P}{2} + 1$ 'heavy' tasks with weight w_H. The bottleneck of *HIER* is $B = 2w_H$,

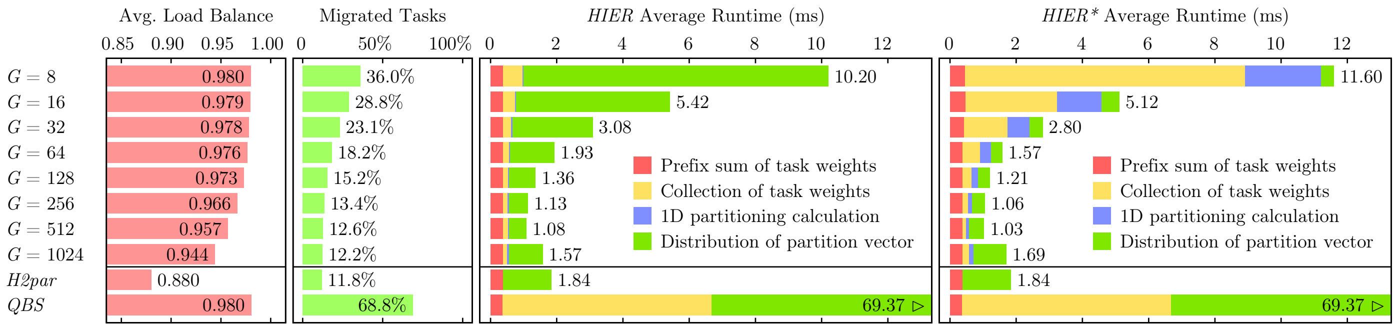


Fig. 6. 1D partitioning benchmark results of the hierarchical methods *HIER* and *HIER** with the CLOUD dataset (559 104 tasks) for 16 384 processes on JUQUEEN. For comparison, the results of the parallel heuristic *H2par* and the sequential exact algorithm *QBS* ($q = 1$) are shown.

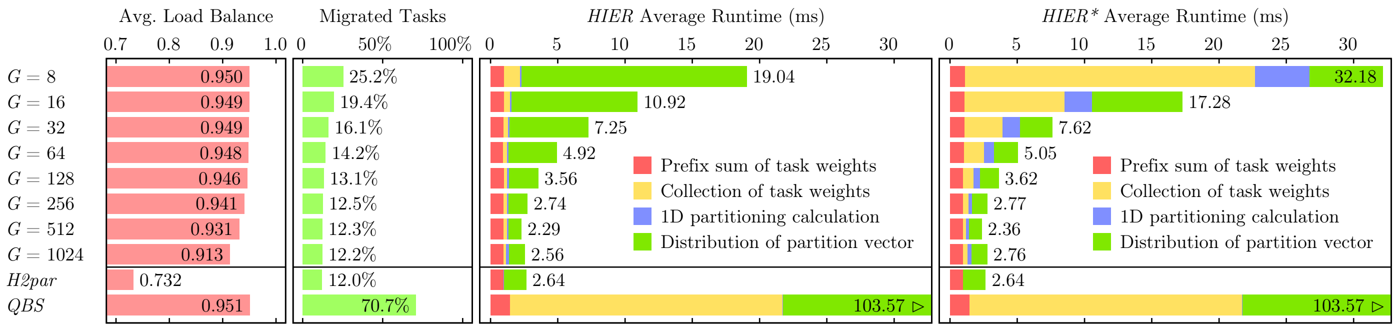


Fig. 7. 1D partitioning benchmark results of the hierarchical methods *HIER* and *HIER** with the LWFA dataset (1 048 576 tasks) for 16 384 processes on JUQUEEN. For comparison, the results of the parallel heuristic *H2par* and the sequential exact algorithm *QBS* ($q = 1$) are shown.

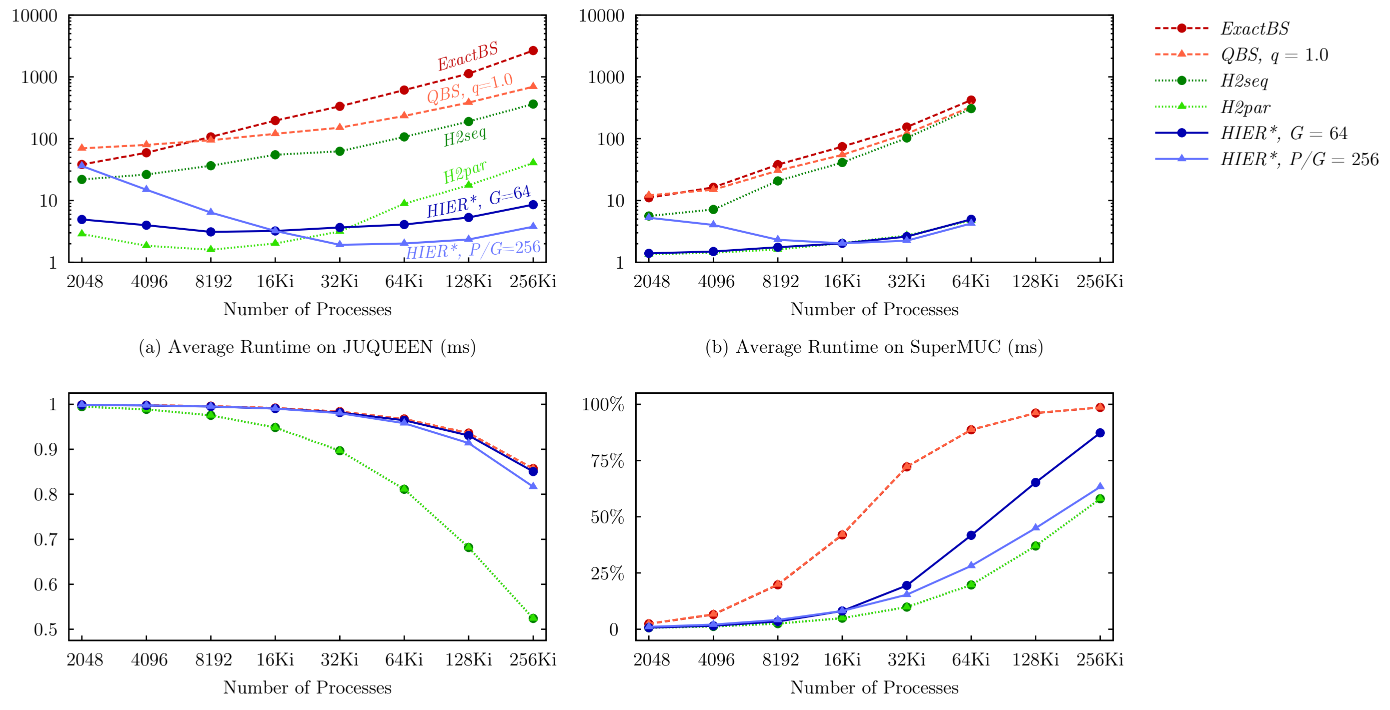
since one partition in the first half needs to take two tasks. With $w_H = \Sigma w_i/(P+1+\epsilon)$ the load balance can be computed as $\Lambda = (P+1+\epsilon)/2P$, which is $1/2$ for $P \to \infty$. In the optimal partitioning, however, $s_{P/2}$ would be $\frac{P}{2}+1$ and the bottleneck B^{opt} would be found in the second half of the partitions. Assuming the weights are such that all partitions of the second half have equal load, B^{opt} would be $(\Sigma w_i - w_H \frac{P}{2})/\frac{P}{2}$. For the optimal load balance for this case we can thus formulate: $\Lambda^{opt} = \frac{P+1+\epsilon}{P+2+2\epsilon}$, which is 1 for $P \to \infty$. This theoretical example shows that *HIER* reaches a quality of $q = 1/2$ in the worst case. However, the following results show that nearly optimal balance is reached for two representative applications.

C. Experimental Evaluation of the Group Count's Impact

To investigate the impact of the group count G on the characteristics of the hierarchical algorithm we ran the benchmark described in Sec. IV with 16 384 processes on JUQUEEN. Fig. 6 shows the results averaged over the 100 iterations of the CLOUD dataset with a replication factor of 13×7. The runtimes are shown as averages over all MPI processes classified into the phases of the partitioning methods. Comparing *H2par* and *QBS*, we see that the serial exact method consumes a large amount of runtime collecting the task weights and even more distributing the partition vector to all ranks. The latter results from the waiting time of 16 383 processes while rank 0 computes the partitioning, which takes 44 ms on average. The two hierarchical methods show exactly the same behavior for load balance and migration and a very similar total runtime. In *HIER*, most time is consumed waiting for the group master to compute the partitioning before the partition vector can be distributed to all processes. The predicted influence of the group count is clearly visible; up to $G = 512$ the runtime is decreasing, even below the runtime of the heuristic. However, with 1024 groups the runtime is increasing because the MPI_Bcast operation to distribute the partition vector to all group members consumes substantially more time. Even with a small number of 8 groups, migration and runtime are clearly reduced in comparison to *QBS*, while nearly reaching optimal load balance. The comparison of runtimes for *HIER* and *HIER** shows no clear winner, though *HIER** is faster for 6 out of 8 group counts.

Fig. 7 shows the results for the LWFA dataset, averaged over the 2000 time steps. This dataset achieves a lower optimal load balance than the CLOUD dataset, due to the very large maximum relative task weights. As a result of the higher number of tasks, all partitioning algorithms have a larger runtime compared to the CLOUD dataset. However, the group count G shows a very similar influence on performance and quality. With LWFA, *HIER* runs faster than *HIER** for all group counts. This can be explained by the higher number of tasks and a noticeable contiguous region within w_i with very low values, which both increases the maximum number of tasks within a single group and, thus, increases the costs of communicating the task weights to all group members. Additionally, the high imbalance of tasks per group leads to waiting time when distributing the partition vector to all ranks.

In summary, these results show that changing the number of groups enables to adjust the hierarchical methods to the needs of the application: For highly dynamic applications requiring frequent load balancing one will prefer a larger group count, such that the costs of partitioning and migration are minimal. On the contrary, a smaller group count is beneficial for less dynamic applications, as the higher costs for partitioning and migration will be compensated by the improved load balance.



(a) Average Runtime on JUQUEEN (ms)

(b) Average Runtime on SuperMUC (ms)

(c) Average Load Balance

(d) Migrated Tasks

Fig. 8. 1D partitioning benchmark results showing the scalability of the sequential exact algorithms *ExactBS* and *QBS*, the heuristic *H2* (sequential and parallelized), as well as the proposed hierarchical method *HIER** with fixed number of groups G=64 and fixed group size P/G=256. The CLOUD dataset with 1 357 824 tasks has been used for this benchmark.

D. Experimental Evaluation of Scalability

We used the CLOUD dataset with a replication factor of 13×17 (1 357 824 tasks) to compare the scalability of the partitioning methods on JUQUEEN and SuperMUC. Based on the findings from Fig. 6, we selected *HIER** as hierarchical method with two different options for the group count: A fixed group count of $G = 64$, which should result in very high load balance, and a fixed group size of $P/G = 256$, which should be more scalable at the cost of balance at high process counts. Note, that both versions are identical at 16 384 ranks. Fig. 8 compares the relevant metrics for both configurations of *HIER** with the sequential exact algorithms *ExactBS* and *QBS* as well as the heuristic *H2* (parallel and sequential).

Scalability on JUQUEEN: The runtime on JUQUEEN, presented in Fig. 8 (a), shows a considerable gap between parallel and sequential methods at large scale. The exact algorithm *ExactBS* requires 2.67 s wall clock time on average to decompose 1.36 million tasks into 262 144 partitions. *QBS* scales little better and achieves 0.69 s. The high-quality hierarchical method *HIER** with $G = 64$ achieves almost the same balance with an average quality factor $q = 0.992$ but requires 8.55 ms only, which is more than 300 times faster compared to *ExactBS*. *HIER** with fixed group size $P/G = 256$ is even faster with 3.77 ms, but achieves lower quality $q = 0.953$. The scalability clearly shows the advantage of the hierarchical methods. *HIER** with $P/G = 256$ is even able to provide superlinear speed-up from 2048 to 32 768 processes. However, the execution time does not decrease further, which is mainly due to the final distribution of the partition vector. Similarly, the performance of *H2* suffers from the poor scalability of the global MPI_Allgather to distribute the final partition.

Scalability on SuperMUC: The scalability behavior, shown in Fig. 8 (b), is similar to JUQUEEN, except that the hierarchical methods show less scalability. *HIER** with fixed group count achieves approximately the same runtime as *H2par* and runs 85 times faster than *ExactBS* on 65 536 processes. The runtime difference between the three serial methods is much less on SuperMUC compared to JUQUEEN. This is caused by the faster computation speed of the processor, which reduces the influence of the serial method on the overall speed. We also observed that global MPI collectives scale better on JUQUEEN, except for MPI_Allgather in *H2par*.

Load balance: Fig. 8 (c) compares the load balance of the methods. The exact algorithms *ExactBS* and *QBS* always achieve the optimal balance, while *HIER** with fixed group count $G = 64$ is close behind. As expected, we observe that a fixed group size for *HIER** leads to less balance at large scale, which is yet clearly higher than the balance achieved by the heuristics.

Task migration: Fig. 8 (d) shows the percentage of migrated tasks. We can see a large difference between heuristics and exact methods. At 262 144 processes, on average 98.6 % of the tasks are migrated every iteration using exact methods, while heuristics migrate 58.0 % only. The hierarchical methods lie in between the heuristics and the exact methods.

VII. Conclusions

Large-scale simulations with strong workload variations in both space and time require scalable and accurate dynamic load balancing techniques. Such applications will benefit from our improved 1D partitioning algorithms presented in this paper. After determining that no parallel exact 1D partitioning algorithms exist, we first investigated runtime optimizations for a published exact algorithm. Second, we introduced a new parallel method that makes high-quality dynamic load balancing feasible at large scale. Our method applies a scalable heuristic to parallelize an exact algorithm and avoid the high communication costs of a centralized method. The hierarchical approach enables to adjust the partitioning algorithm to the dynamical behavior of the application. Our experimental evaluation on 262 144 processes shows that the hierarchical algorithm runs more than 300 times faster compared to the fastest published exact algorithm, while the load balance is almost optimal. Comparing the benchmark results presented in this paper with results based on different artificial datasets of the same total size, we discovered that the task weights have only minor influence on the execution time of the algorithms, due to a dominant amount of communication. On the other hand, load balance and migration costs strongly depend on the characteristics of the task weights. Regarding these two load balancing metrics, the hierarchical algorithm turned out to offer a very good compromise between heuristics and exact methods. Our methods are implemented in the dynamic load balancing and model coupling framework FD4, which is available as open source [21]. The framework has been used to enable scalable load balancing up to 262 144 processes in the coupled cloud simulation model COSMO-SPECS+FD4 [22].

In this practical study we introduced our new hierarchical 1D partitioning methods and compared them to existing algorithms using two datasets. A thorough comparison using more datasets from various applications and bounds on the runtime performance would be beneficial to understand the applicability of the algorithms for certain applications. However, this was out of the scope of this paper and remains future work. While our hierarchical approach considerably reduces the total migration volume, it is not yet clear how 1D partitioning algorithms could explicitly reduce these costs for applications where migration is expensive. Furthermore, our hierarchical method could be extended by automatic runtime tuning for the optimal group count. It should be checked regularly whether the execution time of the application benefits from modifying the group count. To improve the scalability of dynamic load balancing, it will be necessary to avoid the replication of the full partition vector on all processes. As we have seen in our measurements, this is the largest scalability bottleneck of the presented methods. Typically, applications based on domain decomposition require knowledge of the neighbor partitions only. In these cases, it is sufficient to update the partition vector of a process only for the tasks adjacent to the local partition in the multidimensional domain.

Acknowledgment

We thank the Jülich Supercomputing Centre, Germany, for access to JUQUEEN, the Leibniz Supercomputing Centre, Germany, for access to SuperMUC, and the German Meteorological Service for providing the COSMO model. Furthermore, we want to thank Verena Grützun, Ralf Wolke, and Oswald Knoth for their support regarding the COSMO-SPECS model, René Widera for providing the LWFA dataset, and Daniel Hackenberg for his suggestions. This work was supported by the German Research Foundation grant No. NA 711/2-1 and by the 'Center for Advancing Electronics Dresden' (cfaed).

References

[1] J. Dongarra *et al.*, "The International Exascale Software Project Roadmap," *Int. J. High Perform. C.*, vol. 25, no. 1, pp. 3–60, 2011.

[2] A. Geist and R. Lucas, "Major Computer Science Challenges At Exascale," *Int. J. High Perform. C.*, vol. 23, no. 4, pp. 427–436, 2009.

[3] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, "Extreme-Scale AMR," in *Proc. SC '10*, 2010.

[4] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, "Dynamic Load Balancing for Unstructured Meshes on Space-Filling Curves," in *Proc. IPDPSW 2012*, 2012, pp. 1661–1669.

[5] R. Wolke, O. Knoth, O. Hellmuth, W. Schröder, and E. Renner, "The Parallel Model System LM-MUSCAT for Chemistry-Transport Simulations: Coupling Scheme, Parallelization and Applications," in *Proc. ParCo 2003*, ser. Adv. Par. Com., vol. 13, 2004, pp. 363–369.

[6] M. Lieber, V. Grützun, R. Wolke, M. S. Müller, and W. E. Nagel, "Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4," in *Proc. PARA 2010*, ser. LNCS, vol. 7133, 2012, pp. 131–141.

[7] M. Xue, K. K. Droegemeier, and D. Weber, "Numerical Prediction of High-Impact Local Weather: A Driver for Petascale Computing," in *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC, 2008, pp. 103–124.

[8] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, and P. Gibbon, "A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations," *Comput. Phys. Commun.*, vol. 183, no. 4, pp. 880–889, 2012.

[9] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, and R. Widera, "Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability," in *Proc. SC '13*, 2013.

[10] J. D. Teresco, K. D. Devine, and J. E. Flaherty, "Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, ser. LNCSE. Springer, 2006, vol. 51, pp. 55–88.

[11] J. R. Pilkington and S. B. Baden, "Dynamic partitioning of non-uniform structured workloads with spacefilling curves," *IEEE T. Parall. Distr.*, vol. 7, no. 3, pp. 288–300, 1996.

[12] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distr. Com.*, vol. 64, no. 8, pp. 974–996, 2004.

[13] S. Miguet and J.-M. Pierson, "Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing," in *Proc. High-Performance Computing and Networking*, ser. LNCS, vol. 1225, 1997, pp. 550–564.

[14] J. T. Oden, A. Patra, and Y. G. Feng, "Domain Decomposition for Adaptive hp Finite Element Methods," in *Contemp. Math.*, vol. 180, 1994.

[15] Y. Han, B. Narahari, and H.-A. Choi, "Mapping a chain task to chained processors," *Inform. Process. Lett.*, vol. 44, no. 3, pp. 141–148, 1992.

[16] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé, "Periodic hierarchical load balancing for large supercomputers," *Int. J. High Perform. C.*, vol. 25, no. 4, pp. 371–385, 2011.

[17] V. Grützun, O. Knoth, and M. Simmel, "Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM–SPECS: Model description and first results," *Atmos. Res.*, vol. 90, no. 2-4, pp. 233–242, 2008.

[18] PIConGPU website. [Online]. Available: http://picongpu.hzdr.de

[19] A. D. Debus *et al.*, "Electron Bunch Length Measurements from Laser-Accelerated Electrons Using Single-Shot THz Time-Domain Interferometry," *Phys. Rev. Lett.*, vol. 104, p. 084802, 2010.

[20] M. Lieber, "Dynamische Lastbalancierung und Modellkopplung zur hochskalierbaren Simulation von Wolkenprozessen," Dissertation, Technische Universität Dresden, 2012, http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-95674.

[21] FD4 website. [Online]. Available: http://wwwpub.zih.tu-dresden.de/~mlieber/fd4

[22] M. Lieber, W. E. Nagel, and H. Mix, "Scalability Tuning of the Load Balancing and Coupling Framework FD4," in *NIC Symposium 2014*, ser. NIC Series, vol. 47, 2014, pp. 363–370.