Highly scalable SFC-based dynamic load balancing and its application to atmospheric modeling

Matthias Lieber, Wolfgang E. Nagel

Technische Universität Dresden, 01062 Dresden, Germany

This is the peer-reviewed and revised version (post-print) of the following article:

> M. Lieber and W. E. Nagel, Highly scalable SFC-based dynamic load balancing and its application to atmospheric modeling, Future Generation Computer Systems

Once the article is published, it can be found under the DOI 10.1016/j.future.2017.04.042.

Publication History:

- Manuscript Accepted: 29 Apr 2017
- Manuscript Revised: 29 Mar 2017
- Manuscript Received: 29 Feb 2016

Highly scalable SFC-based dynamic load balancing and its application to atmospheric modeling

Matthias Lieber*, Wolfgang E. Nagel

Technische Universität Dresden, 01062 Dresden, Germany

Abstract

Load balance is one of the major challenges for efficient supercomputing, especially for applications that exhibit workload variations. Various dynamic load balancing and workload partitioning methods have been developed to handle this issue by migrating workload between nodes periodically during the runtime. However, on today's top HPC systems – and even more so on future exascale systems – runtime performance and scalability of these methods becomes a concern, due to the costs exceeding the benefits of dynamic load balancing. In this work, we focus on methods based on space-filling curves (SFC), a well-established and comparably fast approach for workload partitioning. SFCs reduce the partitioning problem from n dimensions to one dimension. The remaining task, the so-called 1D partitioning problem or chains-on-chains partitioning problem, is to decompose a 1D workload array into consecutive, balanced partitions. While published parallel heuristics for this problem cannot reliably deliver the required workload balance, especially at large scale, exact algorithms are infeasible due to their sequential nature. We therefore propose a hierarchical method that combines a heuristic and an exact algorithm and allows to trade-off between these two approaches. We compare load balance, execution time, application communication, and task migration of the algorithms using real-life workload data from two different applications on two different HPC systems. The hierarchical method provides a significant speed-up compared to exact algorithms and yet achieves nearly the optimal load balance. On a Blue Gene/Q system, it is able to partition 2.6 million tasks for 524 288 processes with over 99 % of the optimal balance in 23.4 ms only, while a published fast exact algorithm requires 6.4 s. We also provide a comparison to parallel load balancing methods implemented in the Zoltan library and present results from applying our methods to COSMO-SPECS+FD4, a detailed atmospheric simulation model that requires frequent dynamic load balancing to run efficiently at large scale.

Keywords: Massively parallel algorithms, Dynamic load balancing, Space-filling curves, One-dimensional partitioning, Earth and atmospheric sciences

1. 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 a non-uniform and dynamic distribution of simulated physical or chemical phenomena over the spatial domain. Examples are seismic wave propagation [3], two-phase porous media flow for reservoir simulation [4], astrophysical fluid dynamics [5], simulation of turbulent streams and shocks [6], atmospheric modeling of air quality [7] and clouds [8], molecular dynamics [9], as well as particle simulations of plasmas [10, 11] and ion beams [12]. The workload variations are caused by various numerical and modeling techniques that (inherently) lead to a dependency of the local computational workload from the local intensity and space-time scales of simulated processes. In the mentioned examples, these techniques are adaptive mesh refinement [3–5], p-adaptivity [6], adaptive time stepping [6, 7], particle-based methods [9–12], and multiphase modeling [4, 7, 8]. For these types of applications, a static decomposition of the simulation domain leads to waiting time at synchronization points and waste of computational resources and energy. To resolve this issue, dynamic load balancing is applied to redistribute the workload between computing resources periodically during the runtime in order to ensure load balance. The runtime savings by dynamic load balancing are specified in three of the above mentioned publications [6, 8, 10] and lie in the range from 15 to 66 %. Moreover, the importance of a fast and scalable load balancing method is emphasized [3, 4, 6, 8, 9]. With increasing parallelism it becomes more complicated to keep the overhead of dynamic load balancing low such that it does not counteract the achieved runtime savings.

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 [13]: (a) balanced workload in the new partitioning, (b) low communication

*Corresponding author
Email address: matthias.lieber@tu-dresden.de (Matthias Lieber)

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. [13] provide a good overview.

A widely used method is space-filling curve (SFC) partitioning [13, 14]. It is applied for scalable adaptive mesh refinement [3–5] and dynamic load balancing of a fixed number of tasks with varying workload [6, 8, 10]. 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 [14]. 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 [14], but they do not achieve the optimal load balance. On the other hand, exact algorithms [15] 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:

- A new hierarchical and highly scalable 1D partitioning algorithm that provides nearly optimal balance in practice.
- Experimental evaluation of proposed and existing algorithms, including the Zoltan [16] library, comparing load balance, migration costs, partition surface index, and calculation time using real application workload data.
- Application of the hierarchical partitioning algorithm to an atmospheric model, which translates into a noticeable performance improvement.

In a previous conference publication [17] we already introduced the hierarchical algorithm. This work provides a substantial extension by (1) adding the partition surface index as a comparison metric, (2) comparing with geometric algorithms from the Zoltan library, (3) proposing the distributed partition directory as a method to further enhance scalability, and (4) applying the algorithms to a real application.

The rest of the paper is organized as follows. In the next section we approach the background of this work top-down, that is we briefly describe the atmospheric model COSMO-SPECS+FD4, introduce space-filling curve partitioning, and define the 1D partitioning problem. Then, in section 3, we give an overview of related work about 1D partitioning and introduce fundamental concepts and algorithms as foundation of the following sections. In section 4 we present our hierarchical 1D partitioning algorithm and the distributed partition directory. We also provide a theoretical consideration about the load balance quality our algorithm achieves in worst case. In section 5 we evaluate the performance of the hierarchical algorithm by means of measurements and provide a comparison to other state-of-the-art partitioning algorithms. This also includes an evaluation of the impact on the atmospheric model. Finally, in section 6 we summarize our contributions and give an outlook to future work.

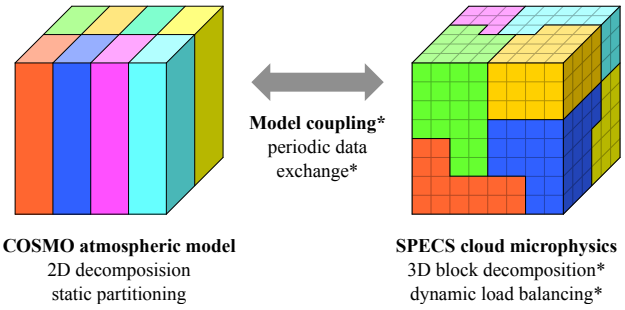


Figure 1: Illustration of the coupling scheme of COSMO-SPECS+FD4 and the partitionings of both model components. Each of the 8 processes owns one partition of COSMO and one of SPECS. FD4 provides services marked by an asterisk (*).

2. From atmospheric modeling to 1D partitioning

In this section we first introduce the atmospheric model COSMO-SPECS+FD4. Due to the high dynamics of its load imbalance, it benefits from applying a fast partitioning method that is scalable and of high quality at the same time, so that load balancing can be executed every few seconds. After that, we introduce space-filling curve partitioning as it is used in COSMO-SPECS+FD4 and define the 1D partitioning problem.

2.1. The atmospheric model COSMO-SPECS+FD4

COSMO-SPECS+FD4 [8, 18] consists of two model components: the non-hydrostatic limited-area weather forecast model COSMO [19] and the cloud microphysics model SPECS [18]. SPECS replaces the so-called bulk parameterization of the COSMO model to describe cloud and precipitation processes with a highly detailed spectral bin cloud microphysics scheme. In this scheme, cloud particles are not modeled with their bulk mass per grid cell only, but with a bin discretization of their size distribution for each grid cell. This allows a more detailed description of the interaction between aerosols, clouds, and precipitation, which is a very important topic in weather, climate, and air quality modeling [20]. SPECS introduces 11 new variables per spatial grid cell to describe water droplets, frozen particles, and insoluble particles, each discretized into 66 size bins, and runs with a smaller time step size than COSMO (in our case 2.5 s and 10 s, respectively). Consequently, the bin cloud microphysics approach is computationally very expensive compared

to widely used bulk schemes so that an efficient parallel implementation is required. Additionally, SPECS causes strong load imbalance since its workload per grid cell depends on the spatially and temporally varying presence of cloud particles. More precisely, the computational effort depends on the number of occupied size bins for each particle class, the evaporation rate, and the presence of ice phase processes at temperatures below freezing point.

For that reason, we and co-authors developed a coupling scheme [8] that separates the data structures of both model components to enable dynamic load balancing in SPECS independently of COSMO, see figure 1. COSMO uses a static domain decomposition of the horizontal grid into regular rectangular partitions. When running SPECS within the decomposition and data management of COSMO, no dynamic load balancing is possible. Therefore, we developed the framework FD4 (Four-Dimensional Distributed Dynamic Data structures [8, 21]) which provides three main services for coupled applications like COSMO-SPECS+FD4, highlighted in figure 1:

- Domain decomposition using several grid blocks per process: data structures to store grid variables are managed by FD4. An iterator concept is used to access blocks and contained variables. All three spatial dimensions are used for decomposition to minimize surface index of partitions and to obtain a finer granularity for load balancing.
- Dynamic load balancing based on the grid blocks as migratable tasks: various repartitioning methods are available, including the hierarchical SFC-based method presented in this paper and an interface to Zoltan [16]. Migration and block allocation/deallocation is performed transparently by FD4. The workload per grid block (i. e. task weight) is specified by the application, e. g. computing time during the previous time step.
- Model coupling: FD4 provides data exchange between data fields managed by FD4 and data fields using a different partitioning of the same grid, e. g. COSMO's partitioning. The coupling is based on serial composition [22], which means that COSMO and SPECS are executed alternately by all processes allocated to the program.

FD4 has been extensively tuned for scalability, which particularly concerns the hierarchical partitioning algorithm and the determination of partition overlaps for coupling [21]. The development of FD4 has been motivated by COSMO-SPECS+FD4. However, it can also be used for other multiphase, multiscale, or multiphysics applications. FD4 is written in Fortran 95 and uses MPI-2 for parallelization. It is available as open source software [23].

In subsection 5.6 we will show that COSMO-SPECS+FD4 is a challenging application for load balancing methods: 1D partitioning heuristics are limited by insufficient achieved load balance, while exact methods cause high overheads for computing the new partitioning. Our hierarchical algorithm reduces the total execution time of COSMO-SPECS+FD4 by more than 10 % compared to these approaches by combining fast partitioning computation with high load balance.

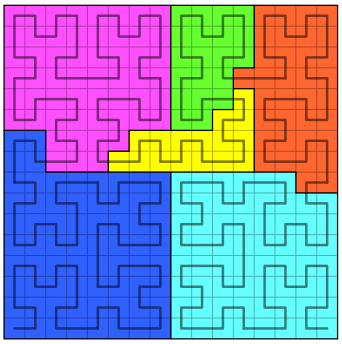


Figure 2: Example of a partitioning created with a 2D Hilbert space-filling curve over a two-dimensional grid.

2.2. *Space-filling curve partitioning*

Space-filling curves provide a fast mapping from n-dimensional to one-dimensional space. One example is shown in figure 2. They can be constructed recursively by refining a specific stencil up to the desired level [13, 14]. One property especially of interest for partitioning is the high locality: discrete neighbor points on the one-dimensional curve are typically also nearby in the n-dimensional space. Relying on this feature, SFC partitioning uses the curve to reduce the partitioning problem to one dimension, which is easier to solve compared to the n-dimensional case. The explicit optimization of the partition shape to reduce inter-partition communication can now be omitted since the SFC's locality typically leads to good partition shapes. Another benefit of SFC partitioning is the simple structure they provide to describe the full partitioning: the partition array s, containing the start indices s_p of all partitions $p = 0, 1, \ldots, P-1$, i. e. one integer per process, is sufficient to store the location of each task, independent from the total number of tasks. Additionally, task migration as reaction to repartitioning is mostly between successive partitions on the curve, which is typically highly local. Different kinds of SFCs have been applied for partitioning. For rectangular grids, the Hilbert SFC shows the best locality properties. Figure 2 shows a 2D grid with a Hilbert SFC. The colors depict six partitions that have been created after applying a 1D partitioning algorithm on the task linearization provided by the SFC.

In this work we assume a regular rectangular grid with tasks at each integer coordinate, i. e. like an n-dimensional matrix. With this assumption and given that we already have a partitioning computed by an SFC, the outline of the SFC-based repartitioning algorithm is as follows: (1) create an array of weights of the local tasks, (2) sort this array by the SFC index of the corresponding tasks, and finally (3) apply a 1D partitioning algorithm, either parallel or serial, on the distributed task weight array. Since the first two steps are straightforward and purely local, we concentrate in this work on the 1D partitioning problem.

Table 1: 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, $L_p = \sum_{i=s_p}^{s_{p+1}-1} w_i = W_{s_{p+1}-1} - W_{s_p-1}$
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)
p	partition index, $p = 0, 1, \ldots, P-1$
q	quality factor of a partitioning, $q = \Lambda/\Lambda^{opt}$
s_p	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, $W_j = \sum_{i=1}^{j} w_i$; $W_0 = 0$

2.3. The 1D partitioning problem

In the 1D partitioning problem, an array w_i, $i = 1, 2, \ldots, N$, 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 [15]. The result is a partition array s_p, $p = 0, 1, \ldots P-1$, that denotes the index in w of the first task assigned to each partition p. Note that each partition contains a contiguous subset of tasks, i. e. partition p contains the tasks $s_p, s_p+1, \ldots, s_{p+1}-1$. 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 $W_j = \sum_{i=1}^{j} w_i$, $j = 1, 2, \ldots, N$ and $W_0 = 0$. The maximum load among all partitions $B = max(L_p)$ is called the bottleneck of a partitioning. The objective of 1D partitioning is to find a partition array s 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 the 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 [24]. One important property of the task weights 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 increasing P will not decrease the bottleneck. Thus, well-balanced partitionings are only achievable if $P \leq \Sigma w_i / max(w_i)$. The introduced symbols are summarized in table 1 and illustrated in figure 3. We denote complete arrays by their symbol without index, i. e. w represents all task weights w_i, $i = 1, 2, \ldots, N$, similarly for W and s.

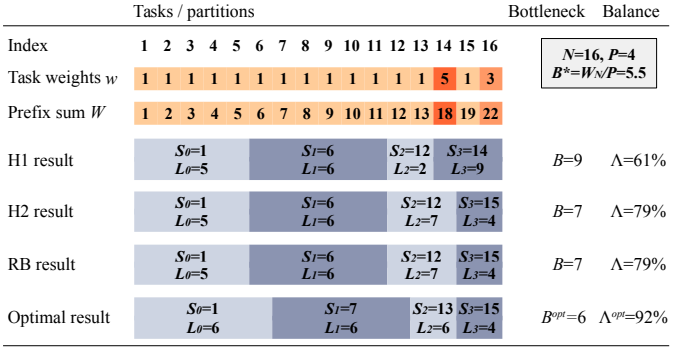


Figure 3: Example to illustrate the load balance deficit of 1D partitioning heuristics for a 4-way partitioning of 16 tasks.

3. Related work

3.1. 1D partitioning heuristics

One of the first 1D partitioning heuristics for SFC-based load balancing is described by Oden et al. [25]. They use the recursive bisection approach where the weight array is recursively cut in two parts with as equal as possible load. Pilkington and Baden [14] introduce a parallel heuristic. The processes search their new partition boundaries within the local part of the weight array prefix sum W 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 [24] 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 subsection 2.3. Harlacher et al. [6] discuss the parallelization of *H2*, especially how to migrate the tasks efficiently when partition borders have been computed locally based on the distributed prefix sum W, i. e. all processes know which tasks to send, but not which tasks to receive.

Example of the heuristics' load balance deficit. Figure 3 shows how the heuristic methods fail to achieve a good load balance by means of an example. Since partition borders are set once and are never refined after scanning the whole task weight array, irregularities in the weight array cannot be compensated. For example, when *H2* determines the start index of the third partition, the algorithm is not aware of that advancing the index by one would help reducing the workload of the third partition and, thus, reduce the bottleneck. Exact algorithms determine the optimal bottleneck and construct a partitioning based on this value.

3.2. Exact 1D partitioning algorithms

Much work has been published on exact algorithms for the 1D partitioning problem; a very extensive overview is given by Pınar and Aykanat [15]. 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 algorithms proposed by Pınar and Aykanat are an improved version of the method by Nicol [26] and their exact bisection algorithm *ExactBS*. Their experiments from 16 to 256 partitions reveal that *ExactBS* offers better performance at larger partition counts. It is based on 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 for B^{opt}, it is required to probe whether a partitioning can be constructed for a given B or not. The probe function successively assigns each partition the maximum number of tasks such that the partition's load is not larger than B, that is it determines the maximum s_p such that $L_{p-1} = W_{s_p-1} - W_{s_{p-1}-1} \leq B$ for $p = 1, 2, \ldots, P-1$ with $s_0 = 1$ and $W_0 = 0$. Probe is successful if the load of the remaining partition $P-1$ is not larger then B, i. e. $L_{P-1} = W_N - W_{s_{P-1}-1} \leq B$. A simple probe algorithm *Probe* using binary search on the prefix sum of task weights W for each s_p has $O(P \log N)$ complexity: binary search finds s_p in the interval $[s_{p-1}+1, N]$ such that $W_{s_p-1} - W_{s_{p-1}-1} \leq B$ and $W_{s_p} - W_{s_{p-1}-1} > B$ with $O(\log N)$ complexity and is carried out for $p = 1, 2, \ldots, P-1$. Han et al. [27] propose an improved probe algorithm with $O(P \log(N/P))$ complexity which partitions W 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. We refer to this improved probe algorithm as *SProbe*. In their method *ExactBS*, Pınar and Aykanat [15] improve probing even further, as will be explained in the next subsection. Based on their previous work, Pınar et al. [28] investigate algorithms for heterogeneous systems with processors of different speed.

3.3. Exact bisection algorithm ExactBS

Since our work builds on *ExactBS*, we describe the complete algorithm briefly in this subsection. For a more detailed description of *ExactBS*, including proofs, we refer to the original publication by Pınar and Aykanat [15]. Two features distinguish this algorithm from an ordinary bisection search algorithm for B^{opt}: (a) the probe function *RProbe* (restricted probe) keeps record of the lowest and highest values found for each s_p in earlier search steps to narrow its search space and (b) the search interval for B^{opt} is always divided such that the bounds are realizable bottleneck values to ensure the search space is reduced by at least one candidate at each iteration.

At first, based on the prefix sum of task weights W, the recursive bisection heuristic is executed to determine an initial upper bound B^{RB} for the bottleneck. Recursive bisection with binary search in W has a complexity $O(P \log(N))$. After that, the partition border's upper bounds SH_p and lower bounds SL_p are initialized, which are required for *RProbe* used during the bisection algorithm. The initialization is performed using *SProbe* by Han et al. [27] and its counterpart *RL-SProbe*, which begins at the last partition and operates from right to left on s and W. The upper bounds SH_p are computed as the minima for each s_p when running *SProbe* on B^{RB} and *RL-SProbe* on B^*. Similarly, the lower bounds SL_p are determined as the maxima for each s_p when running *SProbe* on B^* and *RL-SProbe* on B^{RB}.

Then, the actual bisection algorithm starts searching for the optimal bottleneck value B^{opt} between the lower bound $LB = B^*$ and the upper bound $UB = B^{RB}$. At each iteration, *RProbe* is carried out on $(UB + LB)/2$. If successful, UB is updated; otherwise LB is updated. The iteration terminates when $UB = LB$, with the result $B^{opt} = UB$. In *ExactBS* the bounds are not updated to $(UB + LB)/2$, but to the next smaller realizable bottleneck value in case of UB and to the next larger realizable bottleneck value in case of LB. Thus, the search space is reduced by at least one candidate at each iteration which ensures that the algorithm terminates after a finite number of steps. After a successful *RProbe*, the upper bound is updated to $UB = max(L_p)$, i. e. the maximum load among the partitions determined by *RProbe*. If *RProbe* fails, the lower bound is updated to $LB = min(L_P, min(L_p + w_{s_p+1}))$, i. e. the enlarged load of the first partition that would grow if the bottleneck for probing would be increased gradually.

The restricted probe function *RProbe* improves over the simple *Probe* function outlined in the previous subsection by restricting the search space for each s_p to $[SL_p, SH_p]$. Additionally, these bounds are updated dynamically: if *RProbe* is successful, SH_p is set to the resulting s_p; otherwise SL_p is set to s_p. 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.

3.4. Need for parallel, high-quality 1D partitioning algorithms

To the best of our knowledge, no parallel exact algorithms for the 1D partitioning problem have been published. Only heuristics, like those of Miguet and Pierson [24], can be efficiently parallelized such that the task weight array w stays distributed over all processes according to the current task partitioning and is evaluated locally only. In the probe function of exact algorithms, each iteration over the partitions $p = 0, 1, \ldots, P-2$ depends on the task weights evaluated in the previous iteration. This leads to a serial dependency among the processes and prevents an effective parallelization. Additionally, the collection of all task weights at one single process is infeasible at large scale due to memory limitations and high communication costs. Thus, only parallel heuristics can be used in large-scale applications requiring frequent load balancing. However, as Miguet and Pierson [24] 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, i. e. a decreasing number of

tasks per process N/P, and thus to a reduction of B^*. Consequently, scalable and high-quality partitioning algorithms are required for many future large-scale simulations. This gap has also been observed by Meister and Bader [4], who report that applying an exact 1D partitioning algorithm in their PDE solver 'scales well only up to 1000 cores' so they need to use an approximate method.

3.5. Hierarchical load balancing methods

One solution for the scalability challenge is the application of hierarchical methods for load balancing. Zheng et al. [29] 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. Teresco et al. [30] use hierarchical load balancing in the Zoltan library [16] to adapt to the hierarchy of HPC systems. However, they do not focus on performance at large scale.

4. The proposed hierarchical 1D partitioning algorithm

In this section we propose our two-level hierarchical algorithm for 1D partitioning [17] that aims to overcome the scalability limits of exact algorithms. We combine the parallel computation of a heuristic with the high quality of a serial exact algorithm. Therefore, we first run a parallel heuristic to obtain a coarse partitioning and then we run an exact algorithm within each coarse partition independently. Thus, the exact algorithm is parallelized to a specified degree to achieve runtime and memory savings. We firstly describe the design of the algorithm in subsection 4.1. Then we propose two further enhancements: firstly, in subsection 4.2, we present optimizations for the serial exact bisection algorithm that we apply at the second level. And secondly, we propose a distributed partition directory to increase the scalability of disseminating the partition array in subsection 4.3. Finally, in subsections 4.4 and 4.5 we provide theoretical considerations on the number of required groups and the quality bounds of the hierarchical algorithm, respectively.

4.1. Design of the hierarchical algorithm HIER

The basic idea for our hierarchical method *HIER* 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. 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 a serial exact method. In the second level, 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. the task weight array of size N is not 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. We expect that increasing G should lead to faster execution times and less load balance, whereas decreasing G should result in a slower algorithm but better load balance. In the following, we provide a more detailed description of the five successive phases of our hierarchical method *HIER*. Please refer to table 2 for a brief explanation of the MPI operations we apply.

Table 2: Brief explanation of MPI collective operations [31] used in the algorithm descriptions. Note: all or a group of ranks may participate.

MPI_Bcast	One-to-all: broadcasts a message from a specified rank to all other ranks.
MPI_Gatherv	All-to-one: gathers messages (of different size) from all ranks in a consecutive buffer in rank order at a specified rank.
MPI_Allgather	All-to-all: gathers messages (of same size) from all ranks in a consecutive buffer in rank order and replicates this buffer on all ranks.
MPI_Reduce	Performs a reduction operation (e. g. maximum or sum) of values provided by all ranks and sends the result to a specified rank.
MPI_Exscan	With sum as reduction operation it computes a parallel prefix sum such that ranks $i > 0$ receive the sum of the send buffers of ranks $0, 1, \ldots, i-1$.

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 their local part of W 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. Accordingly, all group masters receive two border positions (except first and last group master, which receive one only) and broadcast them, together with the ID of the rank that found the border, to all processes within the group using MPI_Bcast.

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 (rank-wise) 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_Gatherv such that the master receives all W_j for its group.

4. Exact partitioning within the groups. Based on the local prefix sum of the weight array for the group, the group masters compute the final partitioning with an exact method. We use a

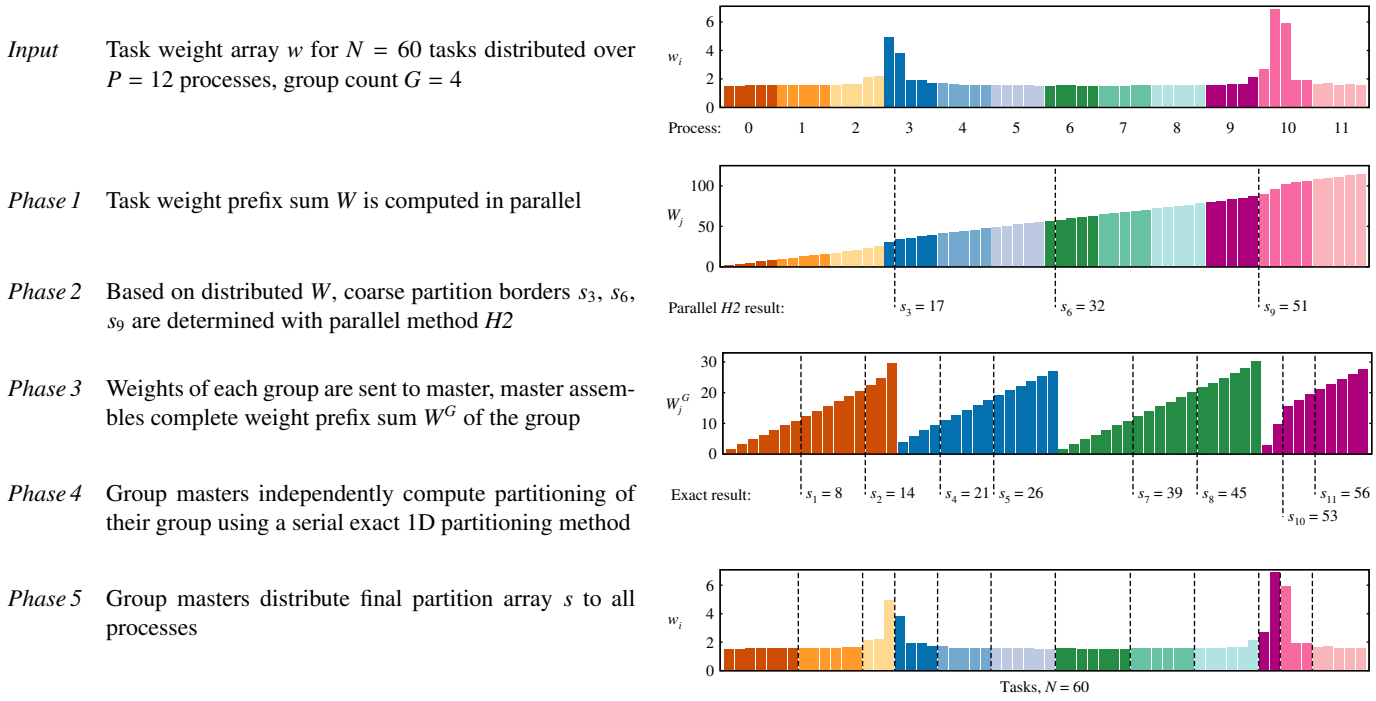


Figure 4: Visual depiction of the 1D partitioning algorithm *HIER* for a small example.

modified version of the exact bisection algorithm, see subsection 4.2. During this phase, G instances of the exact method are running independently to each other.

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

However, since replicating the full partition array on all processes is costly and typically not necessary, we also developed an alternative method that uses a distributed directory of the partition array, see subsection 4.3.

Example. Figure 4 visualizes *HIER* and its 5 phases for a small weight array extracted from the CLOUD2 dataset (see subsection 5.1).

4.2. Modifications to the serial exact bisection algorithm

In the second level of our hierarchical method any exact 1D partitioning algorithm can be used. Because of its execution speed we decided for the exact bisection algorithm *ExactBS* [15]. We applied two slight modifications to this algorithm to achieve further speed improvements that we investigate experimentally in subsection 5.2.

ExactBS+P: New probe algorithm. In *ExactBS*, the *RProbe* function checks whether a partitioning exists for a given bottleneck B. As introduced in subsection 3.3, Pınar and Aykanat restrict the search space for each individual s_p by narrowing the search interval in W dynamically depending on previous *RProbe* calls. Despite the initial runtime overhead of 4 *SProbe* calls to initialize the bounds for each s_p, this results in a fast probe algorithm, especially for large number of tasks N, since the complexity is independent of N.

We developed a probe algorithm which is faster without search space restriction, if (1) the size (number of tasks) of consecutive partitions 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 *EProbe*, shown in figure 5, is based on the estimation that adjacent partitions have the same size, i. e. the same number of tasks. In the first iteration, $s_1 = N/P$ is estimated and, if necessary, linear search ascending or descending in W is performed to find the correct value of s_1. In the remaining iterations, s_p is estimated at s_{p-1} plus the size of the previous partition, i. e. $s_p = s_{p-1} + (s_{p-1} - s_{p-2}) = 2s_{p-1} - s_{p-2}$. This results in an immediate match if the partition $p-1$ has the same size as partition $p-2$ and explains assumption (1). If there is no match, we again start linear search. For relatively small partition sizes, the number of linear search steps will likely be very small and outperform binary search, which explains assumption (2). Consequently, for highly irregular weight arrays we expect our algorithm to be faster than Pınar and Aykanat's *RProbe* at relatively low N/P only.

ExactBS+PI: New initial search interval. In the original *ExactBS* algorithm, Pınar and Aykanat [15] first run the recursive bisection heuristic to use the achieved bottleneck B^{RB} as upper bound in the search. That means their start interval for the binary search for B^{opt} is $I = [B^*, B^{RB}]$. To avoid running the recursive bisection heuristic first, we build on the findings of Miguet and Pierson [24] and use $I = [max(B^*, max(w_i)), B^* + max(w_i)]$ as initial search interval.

```
ExactBS+PI (P, N, B, W, w_max)
  B* := W_N/P
  LB := max(B*, w_max); UB := B* + w_max
  while LB < UB do
    B := (UB + LB)/2
    if EProbe (B) then
      UB := B⁻
    else
      LB := B⁺
  B^opt = UB
  PartitionArray (B^opt)
```

```
EProbe (B)
  start := 0; sum := B; guess := N/P
  B⁻ := 0; B⁺ := W_N
  for p := 1 to P − 1 do
    if W_guess > sum then
      i := guess − 1
      while W_i > sum do i := i − 1
    else
      i := guess
      while i + 1 ≤ N and W_{i+1} ≤ sum do i := i + 1
    guess := min(2i − start, N)
    B⁻ := max(B⁻, W_i − W_start)
    B⁺ := min(B⁺, W_{min(i+1,N)} − W_start)
    if i = N then exit
    sum := W_i + B ; start := i
  if W_N ≤ sum then return true
  else return false
```

```
PartitionArray (B)
  start := 0; sum := B; guess := N/P
  s_0 := 1
  for p := 1 to P − 1 do
    if W_guess > sum then
      i := guess − 1
      while W_i > sum do i := i − 1
    else
      i := guess
      while i + 1 ≤ N and W_{i+1} ≤ sum do i := i + 1
    guess := min(2i − start, N)
    s_p := i + 1
    sum := W_i + B ; start := i
```

Figure 5: Our proposed exact 1D partitioning algorithm *ExactBS+PI* including its probe algorithm *EProbe*. *PartitionArray* uses the same principle as *EProbe* to compute the final partition array s from the optimal bottleneck B^{opt}.

The algorithm *ExactBS+PI* includes both the new probe algorithm *EProbe* and the new initial search interval. The complete algorithm is shown in figure 5, including the function *PartitionArray* that computes the final partition array from the optimal bottleneck determined in *ExactBS+PI*. In contrast to the original *ExactBS*, the *EProbe* algorithm of *ExactBS+PI* does not restrict the search space for the partition borders using (dynamically updated) lower and upper bounds. For dividing the search interval for B^{opt}, *ExactBS+PI* uses the method of *ExactBS* that ensures that interval bounds are set to realizable bottleneck values (see B^- and B^+ in figure 5). In our previous work [17] we introduced another algorithm called *QBS* (quality-assuring bisection) that allows reducing the load balance target in a controlled fashion to reduce the number of binary search steps. *ExactBS+PI* is the same algorithm as *QBS* with quality factor $q = 1$, i. e. without reduction of target load balance.

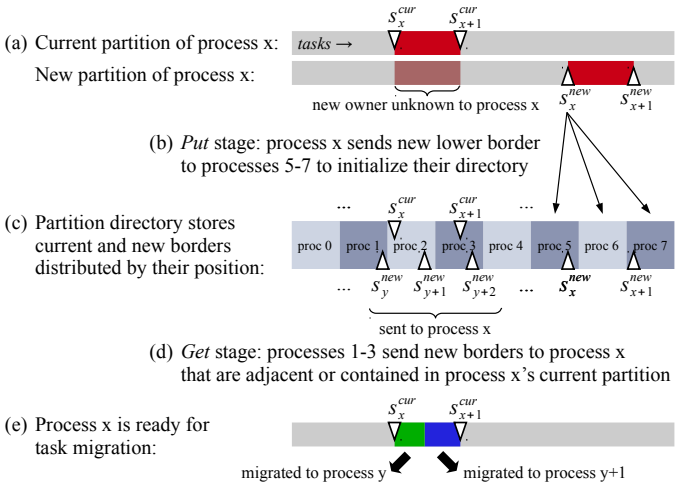


Figure 6: Concept of the distributed partition directory.

4.3. Distributed partition directory

In previous work [17] we already identified the replication of the full partition array on all processes (after the partitioning has been computed) as the major scalability bottleneck of the 1D partitioning heuristic *H2par* and our hierarchical algorithm *HIER*. In most applications, knowledge about the location of all other partitions is not required, since communication takes place typically between neighbor partitions only. Therefore, we developed versions of the parallel methods that do not include the distribution of the full partition array, but rather a distribution of only the parts of the partition array required to perform migration. For migration, each rank needs to know at least the new location of its current tasks, see figure 6 (a). This can be described usually with a very small fraction of the partition array, especially at large scale.

Our method is based on a distributed partition directory, where the partition array itself is partitioned in a fixed, easily to compute way by means of the task index and is distributed over all processes. That means we cut the N SFC-ordered tasks in P consecutive parts of size N/P and assign the start and end indices of these parts consecutively the processes as their fixed part of the partition directory. Each process stores partition borders that are located within its part of the directory, see figure 6 (c). After parallel partitioning computation, we assume that the processes know their own (current and new) partition borders and additionally all still current borders within their part of the partition directory. Our implementation works in two stages. In the *put* stage, the partition directory is updated with the new partition borders: each process p sends its new lower border s_p to ranks that manage partition directory parts overlapping with p's partition, see figure 6 (b). In the following *get* stage, all ranks receive the information they require to perform migration: the location of all new borders that are adjacent to or contained in their current partition, see figure 6 (d). Since the partition directory contains current and new borders, implicit knowledge is present about which information has to be sent to which rank.

With the described procedure the processes do not know from which other processes they will receive their new tasks. This knowledge is not required, if task migration is using point-to-point messages with so-called wildcard receives (receive from

any rank) or one-sided communication. We implemented a small addition, such that the approach also works for point-to-point messages without wildcard receives: during the *get* stage, all processes receive additionally the location of all current borders that are adjacent to or contained in their new partition.

We did not implement the update of the location for tasks adjacent to a partition in the multidimensional application space, since the exact requirements are application-specific. However, in general it could be implemented in a scalable way like this: after partitioning calculation and before migration, all neighbor pairs in the still current partitioning exchange the new owner of their tasks that are located at the corresponding partition boundary. These neighbor IDs are stored within the task and, if a task is migrated, they are also communicated to the new owner.

In the last phase of the 1D partitioning algorithm *HIER*, the distributed partition directory replaces an MPI_Allgather and an MPI_Bcast, both on sub-communicators, by a sparse point-to-point communication pattern with small messages.

4.4. Determining the number of groups

As already mentioned, the number of groups G has an enormous influence on the runtime and quality characteristics of the hierarchical method. Since increasing G reduces both execution time and load balance, it would be practical if we could estimate an upper bound $G(q^{min})$ that guarantees a lower bound q^{min} for the quality factor of the coarse partitioning. Miguet and Pierson [24] have shown that the bottleneck achieved with their heuristics is $B^{H2} < B^* + max(w_i)$. If we use this to replace B in the definition of the quality factor $q = B^{opt}/B$ and insert $B^{opt} \geq B^*$, we obtain: $q^{H2} > B^*/(B^* + max(w_i)) = q^{min}$ which leads to $1/B^* = (1/q^{min} - 1)/max(w_i)$. The average load of the coarse partitions is $B^* = \Sigma w_i/G$. Thus, we obtain the following equation:

$$G(q^{min}) = (1/q^{min} - 1)\,\frac{\Sigma w_i}{max(w_i)}$$

This equation provides an estimation of the maximum group count $G(q^{min})$ that guarantees a minimum quality factor q^{min} for the coarse partitioning

4.5. 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 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 [24] 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*. The idea is to cause a situation where one of the groups is assigned $P/G+1$ tasks of equal weight. Assuming $G = 2$ for now, this can be achieved by setting the weights such that *H2* places the bisecting coarse partition border after the first $\frac{P}{2} + 1$ tasks, i. e. $s_{P/2} = \frac{P}{2} + 2$. In detail that means: let $N \geq P$, $w_i = w^{left}$ for $i = 1, 2, \ldots, \frac{P}{2} + 1$, and $w_i = w^{right}$ for $i = \frac{P}{2}+2, \frac{P}{2}+3, \ldots, N$. To enforce the unfortunate placement of $s_{P/2}$, we set $W_{P/2+1} = \Sigma w_i/2$ and consequently $w^{left} = W_{P/2+1}/(\frac{P}{2} + 1) = \Sigma w_i/(P + 2)$. The weight of the remaining $N - \frac{P}{2} - 1$ tasks in the second coarse partition is $w^{right} = \Sigma w_i/(2N - P - 2)$. The bottleneck of *HIER* is $B^{HIER} = 2w^{left}$, since at least one partition in the first half needs to take two tasks. Consequently, the resulting load balance is:

$$\Lambda^{HIER} = \frac{B^*}{B^{HIER}} = \frac{\Sigma w_i/P}{2w^{left}} = \frac{P+2}{2P} = \frac{1}{2} + \frac{1}{P},$$

which is $1/2$ for $P \to \infty$.

In the optimal partitioning, however, $s_{P/2}$ would be $\frac{P}{2} + 1$, i. e. the first $\frac{P}{2} + 1$ partitions each take one of the $\frac{P}{2} + 1$ 'left' tasks. Assuming the number of 'right' tasks is a multiple of the number of remaining partitions, i. e. $N - \frac{P}{2} - 1 = k(\frac{P}{2} - 1)$ with integer k, the optimal bottleneck would be $B^{opt} = kw^{right}$. Thus, the optimal load balance can be determined with:

$$\Lambda^{opt} = \frac{B^*}{B^{opt}} = \frac{\Sigma w_i/P}{kw^{right}} = \frac{(\frac{P}{2} - 1)(2N - P - 2)}{P(N - \frac{P}{2} - 1)} = 1 - \frac{2}{P},$$

which is 1 for $P \to \infty$. This case can be applied to other even groups counts G, because $s_{P/2}$ would also be determined by the heuristic. For odd G, similar cases can be constructed. This theoretical example shows that *HIER* reaches a quality of $q = \Lambda^{HIER}/\Lambda^{opt} = 1/2$ in the worst case. However, the following results show that nearly optimal balance is reached for two representative applications.

5. Experimental performance evaluation

In this section we present results from measuring the performance of the hierarchical 1D partitioning method and comparing to various other partitioning methods. Firstly, we describe the benchmark program, workload data sets, and HPC systems used. Then, in subsection 5.2, we evaluate the serial exact bisection algorithm to be used at the second level of the hierarchical method. In subsection 5.3 we analyze the group count's impact followed by a comparison to methods from the Zoltan library in subsection 5.4. In subsection 5.5 we show a strong scalability measurement up to 524 288 processes, and, finally, in subsection 5.6 we evaluate the performance impact of the hierarchical partitioning method on an atmospheric model. By combining fast partitioning computation with high load balance, we achieve more than 10 % reduction of application runtime compared to exact methods and parallel heuristics.

5.1. Evaluation benchmark

We have developed an MPI-based benchmark to compare existing 1D partitioning algorithms and partitioning algorithms from the Zoltan library with our methods. Like in typical applications, the task weights are only known to the process owning the task. This distributed task weight array is input to the algorithms. For the 1D partitioning algorithms, the output is the partition array s, which should be replicated on each process. The output of the Zoltan methods are for each process local

lists of tasks that are migrated: the tasks to receive including their current owners and the tasks to send including their new owners.

5.1.1. Existing algorithms implemented in the benchmark

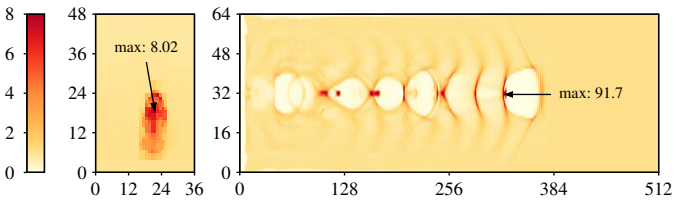
The benchmark program contains several previously published serial and parallel 1D partitioning algorithms as well as an interface to Zoltan to provide a comparison of our proposed algorithms. Please refer to section 3 for a description of the actual 1D partition algorithms. For the serial methods, additional communication steps are necessary to collect task weights and distribute the computed partition array. Here, we outline the existing algorithms in the benchmark and their phases for which we individually collected timings.

- *ExactBS* – Serial exact bisection algorithm by Pınar and Aykanat [15]:
  1. Parallel prefix sum of weights w using MPI_Exscan, determination of $max(w_i)$ on rank 0 using MPI_Reduce
  2. Collection of prefix sum W on rank 0 via MPI_Gatherv
  3. Serial execution of *ExactBS* on rank 0
  4. Distribution of partition array s with MPI_Bcast

- *H1seq* and *H2seq* – Serial heuristics *H1* and *H2* of Miguet and Pierson [24]:
  1. Parallel prefix sum of weights w using MPI_Exscan
  2. Collection of prefix sum W on rank 0 via MPI_Gatherv
  3. Serial execution of *H1* or *H2* on rank 0 using the *SProbe* algorithm by Han et al. [27]
  4. Distribution of partition array s with MPI_Bcast

- *RB* – Serial recursive bisection heuristic:
  1. Parallel prefix sum of weights w using MPI_Exscan
  2. Collection of prefix sum W on rank 0 via MPI_Gatherv
  3. Serial execution of recursive bisection of W on rank 0 using binary search to find the level separators
  4. Distribution of partition array s with MPI_Bcast

- *H2par* – Parallel version of *H2*:
  1. Parallel prefix sum of weights w using MPI_Exscan
  2. Point-to-point communication of first local value in W to $rank - 1$ (to ensure consistency when using floating point weights) and communication of sum of all weights W_N from last rank to all via MPI_Bcast
  3. Execution of *H2* on local part of W
  4. Each found border s_p is sent to rank p, final distribution of partition array to all processes with MPI_Allgather or, in case of *H2par+*, using the distributed partition directory (see section 4.3)



(a) CLOUD Dataset (b) LWFA Dataset

Figure 7: 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.

- *Zoltan* – Recursive coordinate bisection and space-filling curve partitioning from the Zoltan [16] library: The algorithms are applied on the 3D domain, not on the linearization after SFC traversal. We treat Zoltan as a black box, i. e. no timings of internal phases are collected. For more details, refer to subsection 5.4.

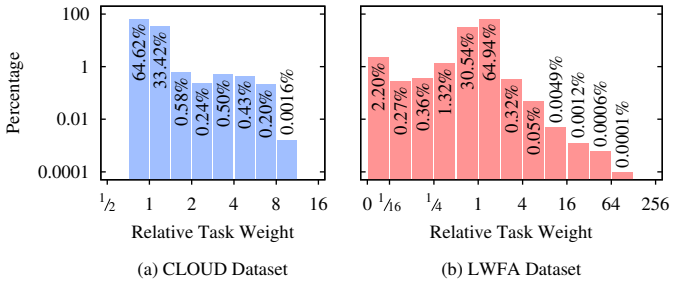
The benchmark determines the runtime of each phase of the partitioning algorithm, the achieved load balance, and the surface index of the partitioning. To observe the amount of migration, the benchmark iterates over a time series of task weight arrays. The task weights are derived from two different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

5.1.2. Real-life datasets CLOUD2 and LWFA

The CLOUD2 dataset is extracted from COSMO-SPECS+FD4 [8, 18], which simulates the evolution of clouds and precipitation in the atmosphere in a high level of detail (refer to subsection 2.1 for a short description). In this scenario, a growing cumulus cloud in the center of the rectangular domain leads to locally increasing workload of the spectral bin cloud microphysics model SPECS. We measured the execution times of $36 \times 36 \times 48 = 62208$ grid cells for 100 successive time steps. The weight imbalance $max(w_i)/avg(w_i)$ varies between 5.17 and 8.02. Figure 7 (a) visualizes the weights of the most imbalanced step. To construct larger weight arrays, we replicated the original block weights in the first two (horizontal) dimensions, e. g. a replication of 3×3 results in $108 \times 108 \times 48 = 559\,872$ weights. After this, we used a Hilbert SFC to create task weight arrays. In subsection 5.6 we show that our new hierarchical algorithm for SFC partitioning improves the runtime of COSMO-SPECS+FD4.

The second dataset originates from a laser wakefield acceleration (LWFA) simulation with the open source particle-in-cell code PIConGPU [11, 32]. In LWFA, electrons are accelerated by high electric fields caused by an ultrashort laser pulse in a gas jet [12]. The dense accumulation of electrons following the laser pulse leads to severe load imbalances, see figure 7 (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 arrays for 200 consecutive time steps (out of 10 000) using a Hilbert SFC. The weight imbalance varies between 44.1 and 91.7.

Figure 8 shows histograms of the most imbalanced task weight arrays in both datasets. Most of the weights are near



(a) CLOUD Dataset (b) LWFA Dataset

Figure 8: Histograms of the most imbalanced task weight arrays 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).

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 arrays are 0.41 for CLOUD2 and 0.31 for LWFA.

5.1.3. Performance metrics and summarization

According to the objectives of repartitioning, we are interested in four important metrics to assess the performance of a repartitioning algorithm:

Load balance of the computed partitioning. As introduced in subsection 2.3, we define load balance as the average load among the partitions divided by the maximum load (i. e. the bottleneck): $\Lambda = B^*/B$. The optimal case is 100 %, which means that all partitions have exactly the same workload.

Surface index of the computed partitioning. The surface index is a measure of the inter-partition communication the partitioning induces if tasks communicate with their direct neighbor tasks (up to 6 neighbors in case of a 3D grid of tasks). It is similar to the edge-cut in graph partitioning problems. The global surface index is defined as the number of faces between neighbor tasks that cross partition borders divided by total number of faces between neighbor tasks [33]. The worst case is 100 %, i. e. no task has a neighbor task in the same partition.

Task migration. The amount of migration is measured as the number of tasks that are migrated after a new partitioning is computed divided be the total number of tasks. Again, the worst case is 100 %, i. e. all tasks are migrated.

Runtime. The benchmark measures the runtime for each phase of the partitioning algorithm (except for Zoltan, which we treat as a black box) and reports averages over all processes. We ensure (as far as possible with MPI) that all processes start the partitioning algorithm timely synchronized.

Summarization over time steps. The benchmark reports the above mentioned metrics per time step of the datasets. To summarize the runtime over the time steps we use median and add percentiles where applicable to show the variation. The other three metrics are averaged over the time steps. In each run, we use a warm-up phase of 10 additional time steps before collecting the metrics.

Table 3: Description of the HPC systems.

Name	JUQUEEN	Taurus (one HPC island)
System	IBM Blue Gene/Q	Bullx DLC 720
Processor	IBM PowerPC A2 1.6 GHz	Intel Xeon E5-2680v3 2.5 GHz, no hyper-threading
Cores per node	16 cores	24 cores
RAM per node	16 GiB RAM	64 GiB RAM
Total nodes	28 672	612
Total cores	458 752	14 688
Network	IBM proprietary	Infiniband FDR
Topology	5D torus	Fat tree
MPI	MPICH2 based, version V1R2M3, xl.legacy.ndebug	Intel MPI 5.1.2.150
Compiler	IBM XL 14.1	Intel 2015.3.187

Table 4: Choice of I_MPI_ADJUST_* parameters for the Intel MPI library.

Parameter name	Value	Meaning
ALLGATHER	2	Bruck's algorithm
ALLGATHERV	3	Ring algorithm
ALLREDUCE	9	Knomial algorithm
BCAST	1	Binomial algorithm
EXSCAN	1	Partial results gathering algorithm
GATHER	3	Shumilin's algorithm
GATHERV	1	Linear algorithm

5.1.4. HPC systems used for the benchmark

We performed measurements on two Petaflop-class HPC systems: the IBM Blue Gene/Q system JUQUEEN installed at the Jülich Supercomputing Centre, Germany, and the Bull HPC Cluster Taurus at Technische Universität Dresden, Germany. Their hardware characteristics and the software used in our measurements are shown in table 3. Since JUQUEEN supports simultaneous multithreading, we used 32 MPI processes per node for our measurements. Taurus is organized in several heterogeneous islands and only one of the HPC islands used for the benchmarks is described in table 3. On Taurus, we observed that the performance of our benchmarks highly depends on the choice of tuning parameters of the Intel MPI library [34]. We experimentally determined optimal parameters for tuning MPI collectives and used the same for all measurements, as listed in table 4.

5.2. Evaluation of the serial 1D partitioning algorithms

In this subsection we investigate the effect of our modifications to *ExactBS* presented in subsection 4.2 to justify the choice of the second level algorithm of the hierarchical 1D partitioning method *HIER*. We compare existing heuristics (*H1seq*, *H2seq*, *RB*) and the exact algorithm *ExactBS* with our variants *ExactBS+P* (new probe function) and *ExactBS+PI* (new probe function and modified start interval). Figure 9 shows the performance results with the LFWA dataset for 8192 partitions on JUQUEEN and Taurus. The average number of

tasks per process is 128. The runtime is the measured wall clock time of the 1D partitioning calculation at rank 0 only, i. e. without prefix sum, collection of weights, and broadcast of the partition array. The comparison between the heuristics and *ExactBS* shows, that the heuristics are clearly faster, but they fail to achieve a sufficient load balance. However, the percentage of migrated tasks per iteration is higher with the exact algorithm. The reason is that *ExactBS* places the partition borders depending on all individual values in W. In contrast, the placement in the heuristics mainly depends on B^*, which varies much less between the iterations than the individual weights. Comparing *ExactBS* and *ExactBS+P*, we can see that both require the same amount of steps and both compute the optimal load balance. However, due to the improved probe algorithm, *ExactBS+P* is clearly faster on both HPC systems (factor 3.2 on Taurus and factor 2 on JUQUEEN). The variations in the task migration are caused by the bounding technique for s_p in *ExactBS*, which leads to different partition borders. Since the LWFA dataset contains many tasks with weight much smaller than the average task weight, partition borders may be set differently in some cases without impacting load balance. For *ExactBS+PI* we observe a small runtime reduction over *ExactBS+P*, but a higher number of search steps. That means that the *RB* heuristic helps to determine a narrower initial search interval than our estimation based on $max(w_i)$ used in *ExactBS+PI*. But it does not translate into a runtime improvement, since the additional cost of the *RB* heuristic does not pay off. Based on these results, we have chosen *ExactBS+PI* as the second level 1D partitioning algorithm for our evaluation of the hierarchical method *HIER* in the following subsections.

5.3. Evaluation of the group count's impact

To investigate the impact of the group count G on the characteristics of the hierarchical algorithm and perform a comparison with the parallel version of the heuristic *H2* and the sequential exact algorithms, we ran the benchmark described in subsection 5.1 with 16 384 processes on JUQUEEN.

Results with the CLOUD2 dataset. Figure 10 shows the results with the CLOUD2 dataset with a replication factor of 3×3. The runtimes are classified into the phases of the partitioning methods. The serial exact methods *ExactBS* and *ExactBS+PI* consume a large amount of runtime collecting the task weights and even more distributing the partition array to all ranks (not shown in the graph). The latter results from the waiting time of 16 383 processes while rank 0 computes the partitioning, which takes 40 ms on average in case of *ExactBS+PI*. The hierarchical method *HIER* is much faster and yet able to compete with the exact methods with respect to load balance. With group count $G = 16$ more than 99 % of the optimal balance is achieved while the runtime is decreased by a factor of 12 compared to *ExactBS+PI* and 28 compared to *ExactBS*. In *HIER*, the most time is consumed waiting for the group master to compute the partitioning before the partition array can be distributed to all processes. The expected influence of the group count is clearly visible; up to $G = 256$ the runtime is decreasing down to only two times the runtime of the parallel heuristic *H2par*. However, with 1024 groups the runtime is increasing because the MPI_Bcast operation to distribute the partition array to all group members consumes substantially more time. The amount of migration is also influenced by the group count. Interestingly, it is much closer to the low amount observed for the heuristic. Even with a small number of 16 groups, migration is clearly reduced in comparison to the exact methods. Regarding surface index, all methods achieve approximately the same results, which is not unexpected since the same Hilbert SFC is used in all cases for mapping of tasks from three dimensions to one dimension.

Results with the LWFA dataset. Figure 11 shows the results for the LWFA dataset. This dataset achieves a lower optimal load balance than the CLOUD2 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 CLOUD2 dataset. However, the group count G shows a very similar influence on performance and quality. For the same G, an even higher quality factor and a lower task migration (relative to the task migration of the exact methods) is achieved for the LWFA dataset compared to CLOUD2. The difference in task migration between the heuristic and the exact methods is extremely high for the LWFA dataset and we observe that *HIER* achieves relatively low migration very close to the heuristic.

Summary. 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. Even with a very small number of groups, the runtime is reduced considerably compared to exact methods with negligible impact on load balance.

5.4. Comparison with geometric methods from Zoltan

The previous comparison has shown that the hierarchical method for partitioning with space-filling curves is able to achieve a very good trade-off between partition quality and runtime performance. Now we want to compare the partition quality with other implementations of partitioning methods included in the Zoltan library [16] that is available as open source software [35]. Zoltan implements various parallel geometric and graph-based partitioning methods under the same interface, which allows testing different methods for the same application. Specifically, we include three geometric methods from Zoltan in the comparison: (1) *Zoltan SFC*: Hilbert space-filling curve partitioning, (2) *Zoltan RCB*: rectangular bisection, and (3) *Zoltan RCB/R*: rectangular bisection with rectilinear partitions. We did not include RIB (recursive inertial bisection) in the comparison, because this method is not suitable for repartitioning. We also excluded graph methods because our test

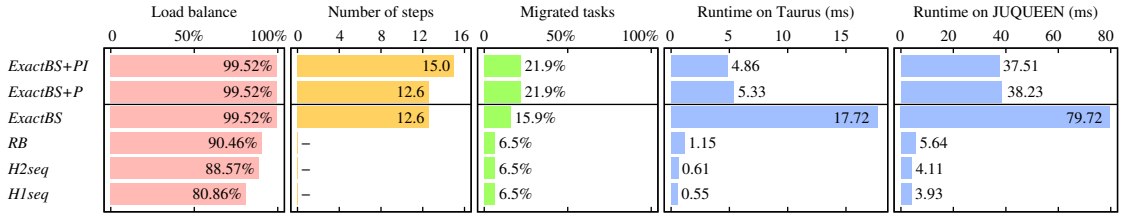


Figure 9: Comparison of the sequential 1D partitioning algorithms with the LWFA dataset (1 048 576 tasks) for 8192 partitions. The reported runtimes include the 1D partitioning calculation only, i. e. no prefix sum and communication.

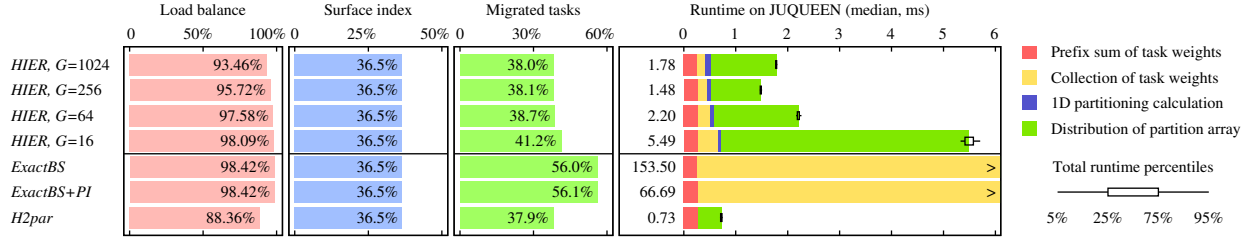


Figure 10: Comparison between the hierarchical method *HIER*, sequential exact algorithms, and the parallel heuristic *H2par* for partitioning the CLOUD2 dataset (559 872 tasks) on 16 384 processes on JUQUEEN. Runtime variation among the 100 iterations is shown as lines (5/95-percentiles) and boxes (25/75-percentiles).

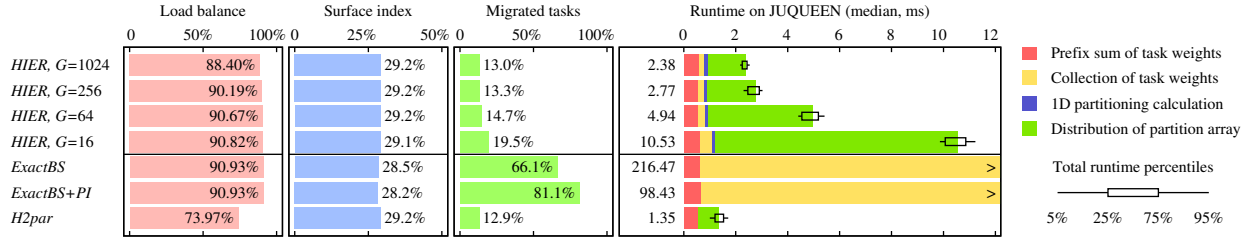


Figure 11: Comparison between the hierarchical method *HIER*, sequential exact algorithms, and the parallel heuristic *H2par* for partitioning the LWFA dataset (1 048 576 tasks) on 16 384 processes on JUQUEEN. Runtime variation among the 200 iterations is shown as lines (5/95-percentiles) and boxes (25/75-percentiles).

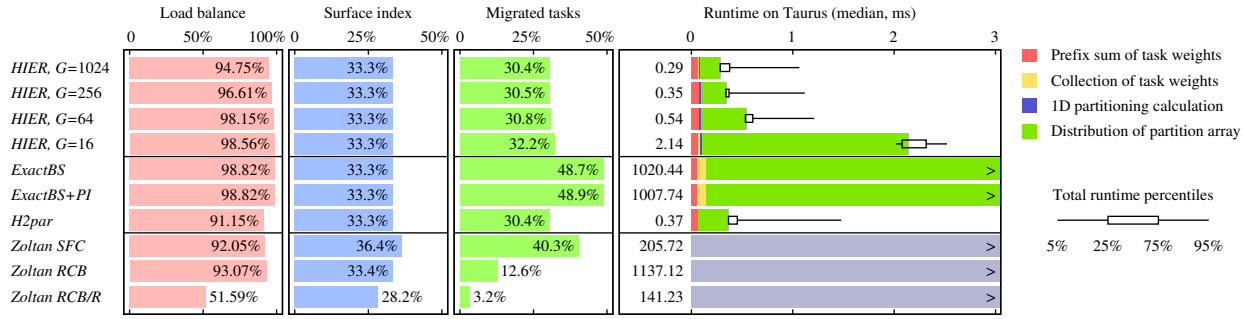


Figure 12: Comparison between the hierarchical method *HIER*, sequential exact algorithms, the parallel heuristic *H2par*, and Zoltan methods for partitioning the CLOUD2 dataset (559 872 tasks) on 12 288 processes on Taurus. Runtime variation among the 100 iterations is shown as lines (5/95-percentiles) and boxes (25/75-percentiles). Note: the runtimes of both *Zoltan RCB* methods vary extremely strong such that the semi-interquartile range is larger than median.

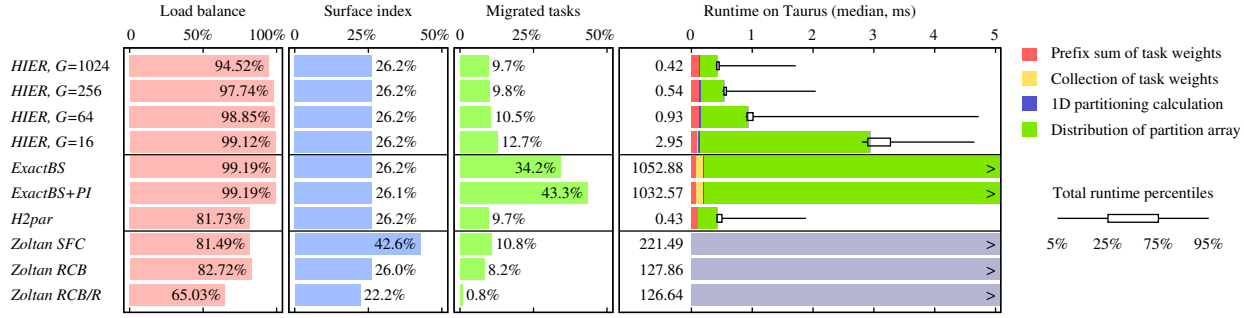


Figure 13: Comparison between the hierarchical method *HIER*, sequential exact algorithms, the parallel heuristic *H2par*, and Zoltan methods for partitioning the LWFA dataset (1 048 576 tasks) on 12 288 processes on Taurus. Runtime variation among the 200 iterations is shown as lines (5/95-percentiles) and boxes (25/75-percentiles). Note: the runtimes of *Zoltan RCB/R* vary extremely strong such that the semi-interquartile range is larger than median.

applications have regular rectangular grids best suited for geometric methods and we already tested a graph partitioner in previous work [36]. One of Zoltan's design goals is a high generality of the implemented methods. They are not tuned for a specific application case. For example, the geometric methods do not know the grid size and need to determine a bounding box around all objects (i. e. tasks) passed to the library before computing a partitioning. Furthermore, objects may exist at arbitrary (floating point) coordinates, whereas the methods we developed assume a fixed-size, cuboid-shaped regular grid with objects at each integer coordinate. Consequently, the primary goal of this comparison is not to evaluate the runtime performance, but the partition quality indicators load balance, surface index, and task migration.

Zoltan setup. For the measurements we used Zoltan version 3.83 with the following non-default parameters: REMAP=1 (maximize overlap between old and new partition), IMBALANCE_TOL=1.0 (strive for best possible balance), RCB_REUSE=1 (reuse previous RCB cuts as initial guesses), and RCB_LOCK_DIRECTIONS=1 (keep order of directions of cuts constant when repeating RCB). *Zoltan RCB/R* is the same as *Zoltan RCB* except that additionally the parameter RCB_RECTILINEAR_BLOCKS=1 is set to force cuboid-shaped partitions. In case of *Zoltan RCB*, objects located on a cutting plane may be moved either to one or the other partition to improve the load balance. However, this also increases the surface index.

Comparison at 12 288 processes. Figures 12 and 13 show results of the comparison with 12 288 processes on Taurus for the CLOUD2 and the LWFA datasets, respectively. Our hierarchical method shows generally the same behavior as on JUQUEEN: a strong improvement of runtime compared the exact methods, a very good trade-off with respect to load balance and migration, and a tunable quality by adapting the group count G. However, the runtime improvement is even more pronounced on Taurus, which is due to the very efficient implementation of global MPI collectives on the IBM Blue Gene/Q that enables the serial methods to run much faster than on Taurus. Regarding the Zoltan methods, we can see that none of the three methods is able to improve the load balance strongly over the parallel heuristic *H2par*. However, both versions of *Zoltan RCB* are able improve the migration clearly over the all SFC-based methods, especially in case of *RCB/R*. This method also achieves the best surface index, but it fails with respect to load balance, which comes not unexpected given the restriction to cuboid-shaped partitions. *Zoltan RCB* achieves approximately the same surface index than our SFC-based methods while *Zoltan SFC* achieves the worst surface index among all methods. The runtime of the Zoltan methods is much higher compared to our parallel methods, but at least in some cases clearly faster than the serial SFC-based methods. However, the runtimes of the RCB-based methods are varying strongly among the time steps of the datasets. For example in case of *Zoltan RCB/R* and the LWFA dataset, the semi-interquartile range, defined as 0.5×(75-percentile−25-percentile), is two times higher than the median among the 200 time steps. The RCB implementation of Zoltan is based on repeatedly splitting the MPI communicator when performing the recursive cuts and relies on heavy MPI collective communication. We identified this as the main source of variations.

Scalability comparison. Figure 14 shows a scalability comparison of the partitioning methods using the LWFA dataset (1 048 576 tasks) on Taurus. Based on the findings from figure 13, we selected two different options for the group count of *HIER*: a fixed group count of $G = 64$, which should result in very high load balance, and a fixed group size of $P/G = 48$, which should be more scalable at the cost of balance at high process counts. Note that both versions are identical at 3072 ranks.

Load balance. The comparison of load balance shows three groups: the best ones are the exact 1D partitioning methods and the hierarchical method, with the $P/G = 48$ version falling slightly behind at 12 288 processes. The second group shows still a very high balance at low process counts but drops notably starting at 3072 processes. It is made up by *Zoltan RCB*, *Zoltan SFC*, and *H2par*. And finally the third group consists of *Zoltan RCB/R* only, which achieves very poor load balance.

Surface index. Regarding surface index, our own SFC-based methods are very close together, whereas *Zoltan SFC* achieves a more than 60 % higher surface index only. This is probably due to the different kind of mapping of tasks to the SFC and the high aspect ratio of the LWFA dataset's grid (32×512×64). The rectangular bisection methods of Zoltan are able to improve the surface index, especially *Zoltan RCB/R*.

Task migration. *Zoltan RCB/R* also achieves the lowest task migration at all processor counts which increases by a factor of 3.3 from 768 to 12 288 ranks. The exact SFC-based methods *ExactBS* and *ExactBS+PI* show a very strong increase of migration over the process count range, more than factor 40. The other SFC-based methods, including *Zoltan SFC*, are very close and grow by factors 15 to 18, approximately the factor 16 by which the number of processes increases, which indicates a reasonable result. *Zoltan RCB* shows the best scalability behavior for task migration growing only by a factor of 2.5, but only at 12 288 ranks it is able to improve over the SFC-based methods.

Runtime. The runtime scalability comparison shows two groups: as expected, our parallel SFC-based methods show the best runtime performance. They require always below 1 ms for the repartitioning calculation. *H2par* is fastest, but *HIER* is only factor 2 to 3 slower, which is remarkable considering the great improvement in load balance over *H2par*. The other methods take one to three orders of magnitude more runtime than the group of the three fastest, with *Zoltan RCB* and *RCB/R* showing at least much better scalability behavior than the exact SFC-based methods.

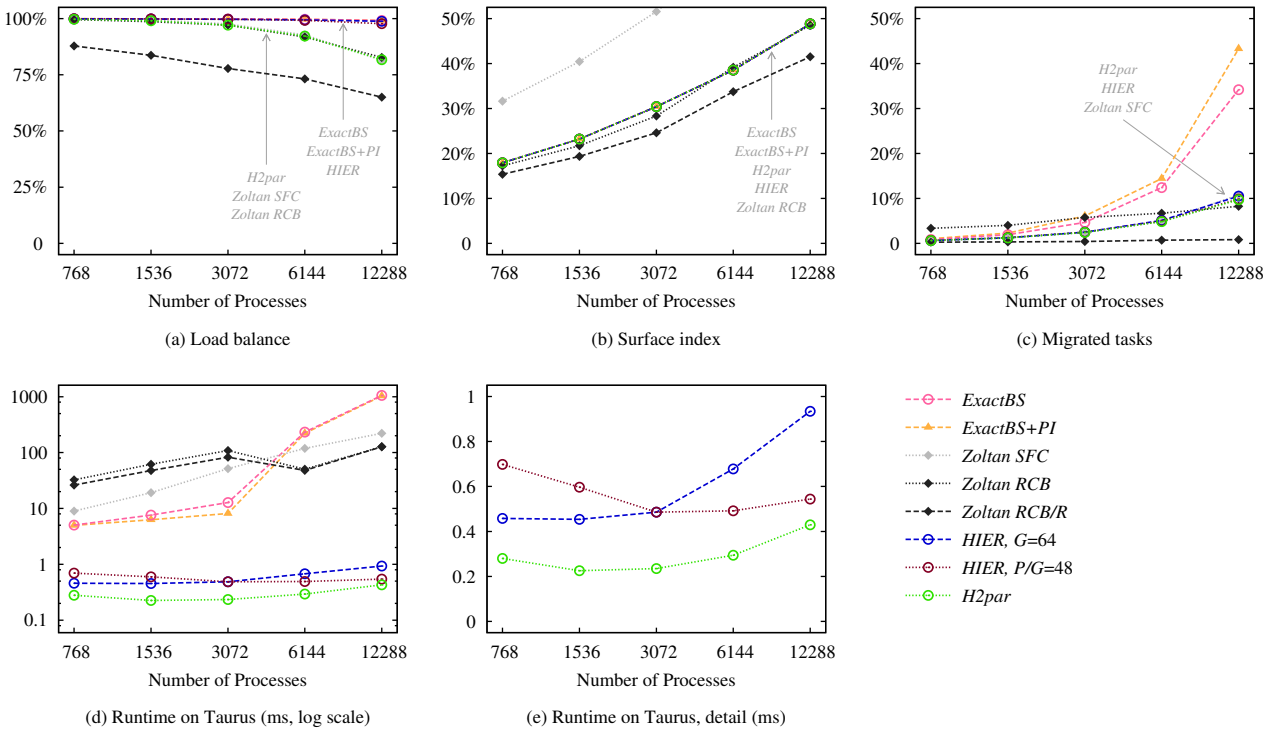


Figure 14: Comparison of scalability between the hierarchical method *HIER*, sequential exact algorithms, the parallel heuristic *H2par*, and Zoltan methods for partitioning the LWFA dataset (1 048 576 tasks) on Taurus. Overlapping lines are additionally labeled within the plots.

Summary. In summary, the comparison with Zoltan has shown that our SFC-based hierarchical methods achieve a better load balance and a comparable surface index for two realistic datasets. In case migration is the major target for optimization, methods based on recursive bisection might be a better choice than SFC-based methods. However, in our implementations the migration is expressed as shifts of borders along the space-filling curve so that tasks are migrated between processes that are nearby (rank-wise), mostly even neighbor ranks. Such communication can often be satisfied within a node via shared memory or via short network paths (i. e. torus neighbors in case of Blue Gene/Q or only a few hops in a fat tree network without going over the top-level switch), which reduces network contention and thus improves bandwidth and latency.

5.5. Evaluation of strong scalability up to 524 288 processes

We used the CLOUD2 dataset with a replication factor of 6×7 (2 612 736 tasks) to compare the scalability to large process counts on JUQUEEN. Again, we used two modes to handle the group count when changing the number of processes: a fixed group count of $G = 64$ and a fixed group size of $P/G = 256$. Both versions are identical at 16 384 ranks. We also included an evaluation of the distributed partition directory for the methods *H2par* and *HIER*, which we label *H2par+* and *HIER+*, respectively. Note that for the metrics load balance, surface index, and task migration, the versions with distributed partition directory behave exactly the same as their counterparts without distributed partition directory. Figure 15 compares the relevant metrics among the various 1D partitioning algorithms.

Load balance. The exact algorithms *ExactBS* and *ExactBS+PI* 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 heuristic.

Surface index. Regarding surface index, all methods behave nearly the same. The very high surface index at large scale is due to the very small number of tasks per process; at 512 Ki[1] processes only an average of 4.98 blocks is assigned to each process, which also explains the relatively low optimal load balance.

Task migration. The percentage of migrated tasks is rising strongly with the number of processes for all studied methods, but the serial exact methods are worse than the others, which are all close together. At 128 Ki processes the exact methods migrate on average more than 88 % of the tasks per time step, whereas the fraction is 77 % for the parallel methods, which is still quiet high.

Runtime. The runtime scalability shows that, firstly, there is a huge gap of four orders of magnitude between the serial exact methods and the heuristic *H2par+* with distributed partition directory. The partition directory enables the heuristic to show a very good scalability behavior, improving its speed up to 16 Ki processes and then only slightly slowing down up to

[1] The binary prefix Ki denotes 1024 as opposed the decimal prefix k.

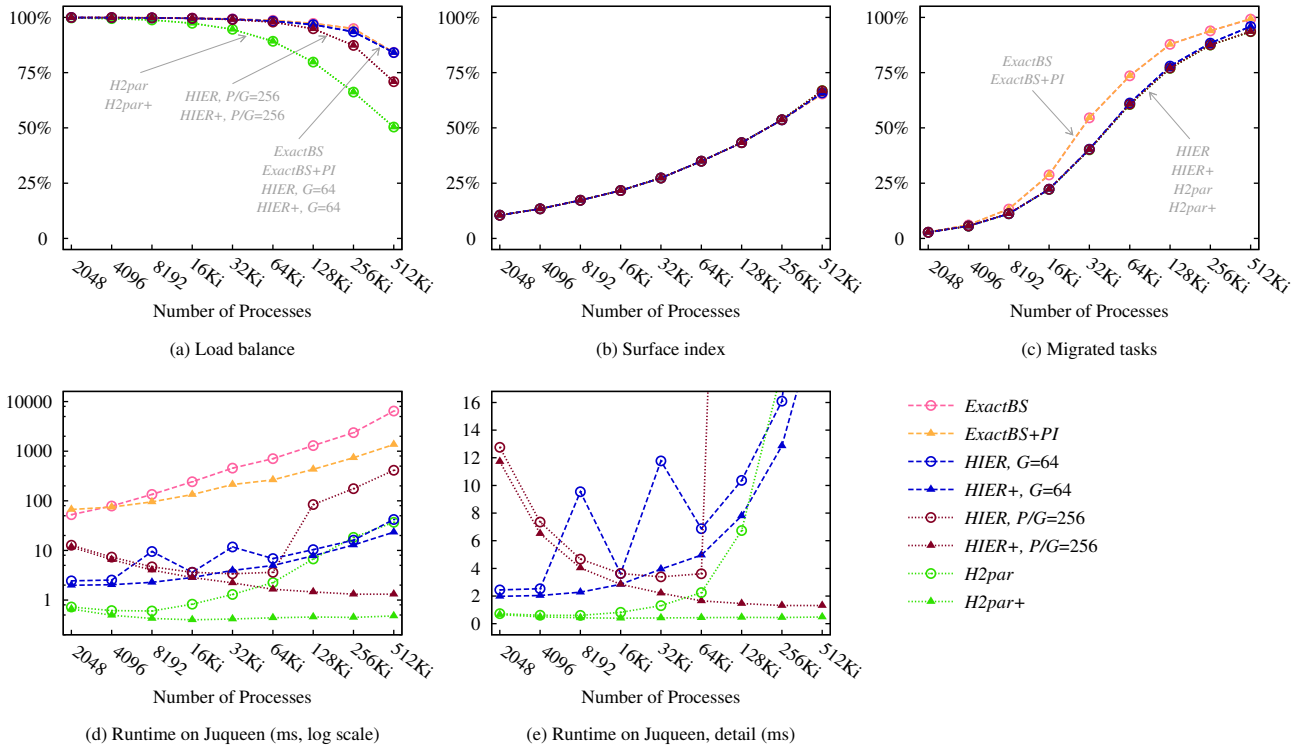


Figure 15: Comparison of scalability between the hierarchical method *HIER*, sequential exact algorithms, and the parallel heuristic *H2par* for partitioning the CLOUD2 dataset with 2 612 736 tasks on JUQUEEN. *HIER+* and *H2par+* are versions using the distributed partition directory instead of the distribution of the full partition array. Overlapping lines are additionally labeled within the plots.

512 Ki processes. The version with distribution of the full partition array to all ranks, *H2par*, is clearly hampered by the global MPI_Allgather operation. Similarly, *HIER* with $P/G = 256$ shows a great improvement through the distributed partition directory and even achieves a speed-up up to 256 Ki processes, but of course at costs of load balance. *HIER* with $G = 64$ achieves a very high load balance with 98.5 % of the optimal balance in the worst case (at 256 Ki processes) and requires only 42.1 ms (23.4 ms for *HIER+*) at 512 Ki processes. This is a substantial speed-up compared to the 6.4 s of *ExactBS* and 1.3 s of *ExactBS+PI*. The outliers at 8192 and 32 Ki processes are reproducible and are caused by longer runtime of the MPI collectives for distributing the full partition array at specific group counts.

Summary. We observed that the hierarchical method is able to close the gap between fast, but inexact heuristics and serial exact algorithms for the 1D partitioning problem. Even with a relatively small group count a huge speed-up compared to the exact methods can be achieved, while maintaining nearly the optimal balance. Tuning the group count allows trading off quality against runtime performance and, thus, adapting to the requirements of the application. If the processes do not need to know the location of all other partitions, a considerable scalability improvement is possible by communicating only the required parts of the partition array.

5.6. Application to atmospheric modeling

The benefit of an optimized partitioning algorithm for highly parallel applications can be determined ultimately only by measuring the end user application runtime. In this subsection we evaluate the impact of our hierarchical 1D partitioning method for load balancing the atmospheric simulation model COSMO-SPECS+FD4 (refer to subsection 2.1 for a brief description). We first evaluate the impact of the partitioning method on the runtime performance of the simulation and then show scalability results.

For PIConGPU we cannot directly measure the runtime benefit of an improved partitioning algorithm, since load balancing is currently not implemented. However, PIConGPU achieves strong scaling such that a single time step of LWFA is computed in less than 100 ms [37]. Together with our findings from sections 5.3 and 5.4 that LWFA requires a high-quality method to achieve high balance and exact methods require approximately one second to compute a partitioning, this indicates a large potential for our algorithm.

Impact of partitioning method. Figure 16 shows the impact of the choice of partitioning method and parameter G on the total runtime of COSMO-SPECS+FD4. In this study the grid of $512 \times 512 \times 48$ was decomposed into 786 432 grid blocks for load balancing SPECS. Note that dynamic load balancing is carried out every time step in this case (180 steps) and that the reported load balance value is the one measured after computations have been performed, i. e. not the load balance based on the task weights (i. e. execution times) from the previous time

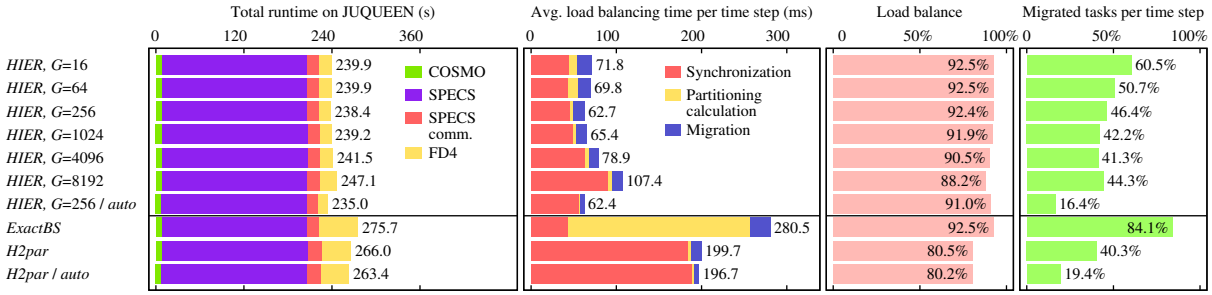


Figure 16: Comparison of the hierarchical 1D partitioning algorithm in COSMO-SPECS+FD4 with *ExactBS* and *H2par* on 65 536 processes on JUQUEEN.

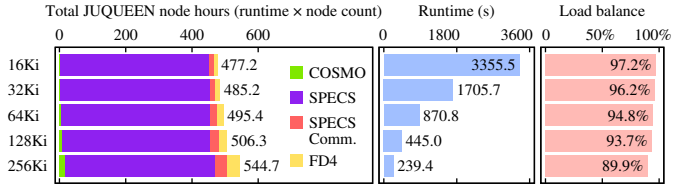


Figure 17: Strong scaling of COSMO-SPECS+FD4 with hierarchical 1D partitioning on JUQUEEN.

step that are used to calculate the partitioning. The results show that the exact method *ExactBS* achieves a high load balance but introduces a noticeable overhead for partitioning calculation. The heuristic *H2par*, on the other side, is fast, but fails to provide a sufficient load balance, which leads to increased synchronization costs. Though, the heuristic achieves a better total runtime. With the hierarchical 1D partitioning method with group count $G = 256$, the runtime of the application is reduced by more than 10 % compared to the heuristic. One can also see that the task migration is relatively high. But even with 84.1 % migrated tasks on average per time step with *ExactBS*, the runtime fraction of task migration is only 1.5 %.

In practice, performing load balancing at every time step may generate noticeable overhead. To reduce the number of load balancing invocations, FD4 is able to decide automatically if load balancing is beneficial. In this *auto mode*, FD4 weighs the time lost due to deferring repartitioning (i. e. the increased synchronization time when not performing repartitioning) against the time required for repartitioning (i. e. partitioning calculation and migration times). Both times are measured at runtime and a history from the last 4 load balancing invocations is kept. Using the *auto mode*, load balancing is carried out only at 30 % of the time steps. Note that synchronization happens at every time step, even if repartitioning is not carried out, to measure the imbalance. As can be seen in figure 16 (line *HIER*, *G*=256 / *auto*), the execution time of COSMO-SPECS+FD4 is reduced further to 235 s and the amount of migration is strongly reduced.

Strong scalability. Figure 17 shows results from a strong scalability measurement with COSMO-SPECS+FD4 using our method *HIER* with *auto mode* for dynamic load balancing. Here, we used a horizontal grid size of 1024×1024 cells and 3 145 728 FD4 blocks. The speed-up from 16 384 to 262 144 processes is 14.0, mainly due to the nearly perfectly scaling SPECS computations that dominate the execution time. The FD4 load balancing achieves a speed-up of 3.7 only, which is mainly caused by synchronization times. Not surprisingly, the load balance is decreasing with the process count, since the number of blocks per process decrease.

In summary, the good scalability of COSMO-SPECS+FD4 allows to run atmospheric simulations with detailed cloud microphysics and a large grid size faster than forecast time. In this case, a 30 min forecast has been computed in 4 min with 262 144 processes.

6. Conclusions

Large-scale simulations with strong workload variations in both space and time require scalable and accurate dynamic load balancing techniques. Partitioning based on space-filling curves is comparably fast and has a high potential for scalable applications. However, attention has to be paid on the 1D partitioning algorithm since heuristics may fail to achieve a good load balance. Therefore, we introduce a new parallel and hierarchical method that makes high-quality dynamic load balancing based on SFCs 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. Additionally, the hierarchical approach allows adjusting the partitioning algorithm's characteristics to the dynamical behavior of the application. Our experimental evaluation on up to 524 288 processes shows that our new algorithm runs more than two orders of magnitude faster compared to the fastest published exact algorithm, while the load balance is almost optimal. To improve the scalability further, we propose a method to avoid the replication of the full partition array on all processes and communicate only the few parts necessary for migration. The comparison with other implementations of geometric partitioning methods shows that the hierarchical SFC-based method runs clearly faster and achieves better load balance and comparable surface index. We show that the improvement in the partitioning algorithm leads to a more than 10 % performance improvement of an atmospheric simulation model with detailed cloud microphysics. Since the partitioning algorithm is implemented as a library, this reduction in execution time comes at almost no effort for users and could benefit various applications. All methods studied in this work are implemented in the dynamic load balancing and model coupling framework FD4, which is available as open source [23], including the benchmark program and

the CLOUD2 dataset to allow reproducibility of our measurements.

The benefits of our hierarchical 1D partitioning algorithm for dynamic applications on highly parallel systems are twofold: (1) the overhead of high-quality repartitioning is reduced strongly allowing more frequent dynamic load balancing and (2) trading off between repartitioning costs and quality is enabled by the group count. With our improvements it is now feasible to perform effective load balancing in cases where application dynamics lead to strong imbalances every few seconds, since repartitioning takes less than 100 ms at half a million processes compared to several seconds with prior published methods. Our measurements indicate that this gap will stay or even be widened at higher concurrency expected for exascale systems.

Our practical comparison with workload datasets from two different relevant applications on two HPC systems showed the applicability and performance of our proposed methods. However, it remains future work to study which type of workloads and applications are less well-suited for our partitioning algorithms and how the algorithms could be improved, especially with regard to the theoretical lower bound on load balance, which we found to be as low as the one for the heuristics. While our hierarchical approach reduces the total migration volume compared to exact algorithms, 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.

Acknowledgments

We thank the Jülich Supercomputing Centre, Germany, for access to JUQUEEN 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 the anonymous reviewers who helped improving the paper. 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. 25 (1) (2011) 3–60. doi:10.1177/1094342010391989.

[2] R. Lucas, et al., Top Ten Exascale Research Challenges, DOE ASCAC subcommittee report (2014).

[3] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, L. Wilcox, Extreme-Scale AMR, in: Proc. SC '10, 2010, pp. 1–12. doi:10.1109/SC.2010.25.

[4] O. Meister, M. Bader, 2D adaptivity for 3D problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids, J. Comput. Sci. 9 (2015) 101–106. doi:10.1016/j.jocs.2015.04.016.

[5] J. J. Carroll-Nellenback, B. Shroyer, A. Frank, C. Ding, Efficient parallelization for AMR MHD multiphysics calculations; implementation in AstroBEAR, J. Comput. Phys. 236 (2013) 461–476. doi:10.1016/j.jcp.2012.10.004.

[6] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, F. Wolf, Dynamic Load Balancing for Unstructured Meshes on Space-Filling Curves, in: Proc. IPDPSW 2012, 2012, pp. 1661–1669. doi:10.1109/IPDPSW.2012.207.

[7] M. Lieber, R. Wolke, Optimizing the coupling in parallel air quality model systems, Environ. Modell. Softw. 23 (2) (2008) 235–243. doi:10.1016/j.envsoft.2007.06.007.

[8] M. Lieber, V. Grützun, R. Wolke, M. S. Müller, W. E. Nagel, Highly Scalable Dynamic Load Balancing in the Atmospheric Modeling System COSMO-SPECS+FD4, in: Proc. PARA 2010, Vol. 7133 of LNCS, 2012, pp. 131–141. doi:10.1007/978-3-642-28151-8_13.

[9] J. Phillips, K. Schulten, A. Bhatele, C. Mei, Y. Sun, E. Bohm, L. Kalé, Scalable Molecular Dynamics with NAMD, in: A. Bhatele, L. Kalé (Eds.), Parallel Science and Engineering Applications: The Charm++ Approach, CRC Press, 2013, Ch. 4, pp. 61–77.

[10] M. Winkel, R. Speck, H. Hübner, L. Arnold, R. Krause, P. Gibbon, A massively parallel, multi-disciplinary Barnes-Hut tree code for extreme-scale N-body simulations, Comput. Phys. Commun. 183 (4) (2012) 880–889. doi:10.1016/j.cpc.2011.12.013.

[11] 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, R. Widera, Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability, in: Proc. SC '13, 2013. doi:10.1145/2503210.2504564.

[12] A. D. Debus, et al., Electron Bunch Length Measurements from Laser-Accelerated Electrons Using Single-Shot THz Time-Domain Interferometry, Phys. Rev. Lett. 104 (2010) 084802. doi:10.1103/PhysRevLett.104.084802.

[13] J. D. Teresco, K. D. Devine, 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, Vol. 51 of LNCSE, Springer, 2006, pp. 55–88. doi:10.1007/3-540-31619-1_2.

[14] J. R. Pilkington, S. B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, IEEE T. Parall. Distr. 7 (3) (1996) 288–300. doi:10.1109/71.491582.

[15] A. Pınar, C. Aykanat, Fast optimal load balancing algorithms for 1D partitioning, J. Parallel Distr. Com. 64 (8) (2004) 974–996. doi:10.1016/j.jpdc.2004.05.003.

[16] E. G. Boman, U. V. Catalyurek, C. Chevalier, K. D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring, Scientific Programming 20 (2) (2012) 129–150. doi:10.3233/SPR-2012-0342.

[17] M. Lieber, W. Nagel, Scalable high-quality 1D partitioning, in: Proc. HPCS 2014, 2014, pp. 112–119. doi:10.1109/HPCSim.2014.6903676.

[18] V. Grützun, O. Knoth, M. Simmel, Simulation of the influence of aerosol particle characteristics on clouds and precipitation with LM–SPECS: Model description and first results, Atmos. Res. 90 (2-4) (2008) 233–242. doi:10.1016/j.atmosres.2008.03.002.

[19] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, T. Reinhardt, Operational Convective-Scale Numerical Weather Prediction with the COSMO Model: Description and Sensitivities, Mon. Weather Rev. 139 (12) (2011) 3887–3905. doi:10.1175/MWR-D-10-05013.1.

[20] IPCC, Climate Change 2013: The Physical Science Basis. Contribution of Working Group I to the Fifth Assessment Report of the Intergovernmental Panel on Climate Change (IPCC), Cambridge University Press, 2013. doi:10.1017/CBO9781107415324.

[21] M. Lieber, W. E. Nagel, H. Mix, Scalability Tuning of the Load Balancing and Coupling Framework FD4, in: NIC Symposium 2014, Vol. 47 of NIC Series, 2014, pp. 363–370, http://hdl.handle.net/2128/5919.

[22] J. W. Larson, Ten organising principles for coupling in multiphysics and multiscale models, ANZIAM J. 48 (2009) C1090–C1111. doi:10.21914/anziamj.v48i0.138.

[23] FD4 website, http://www.tu-dresden.de/zih/clouds (accessed 03/2017).

[24] S. Miguet, 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, Vol. 1225 of LNCS, 1997, pp. 550–564. doi:10.1007/BFb0031628.

[25] J. T. Oden, A. Patra, Y. G. Feng, Domain Decomposition for Adaptive hp Finite Element Methods, in: Contemporary Mathematics, Vol. 180, 1994, pp. 295–301. doi:10.1090/conm/180.

[26] D. M. Nicol, Rectilinear partitioning of irregular data parallel computations, J. Parallel Distr. Com. 23 (1994) 119–134. doi:10.1006/jpdc.1994.1126.

[27] Y. Han, B. Narahari, H.-A. Choi, Mapping a chain task to chained processors, Inform. Process. Lett. 44 (3) (1992) 141–148. doi:10.1016/0020-0190(92)90054-Y.

[28] A. Pınar, E. K. Tabak, C. Aykanat, One-dimensional partitioning for heterogeneous systems: Theory and practice, J. Parallel Distr. Com. 68 (11) (2008) 1473–1486. doi:10.1016/j.jpdc.2008.07.005.

[29] G. Zheng, A. Bhatelé, E. Meneses, L. V. Kalé, Periodic hierarchical load balancing for large supercomputers, Int. J. High Perform. C. 25 (4) (2011) 371–385. doi:10.1177/1094342010394383.

[30] J. D. Teresco, J. Faik, J. E. Flaherty, Hierarchical Partitioning and Dynamic Load Balancing for Scientific Computation, in: Applied Parallel Computing, Vol. 3732 of LNCS, Springer, 2006, pp. 911–920. doi:10.1007/11558958_110.

[31] MPI: A Message-Passing Interface Standard, Version 3.1, 2015, http://www.mpi-forum.org/docs (accessed 03/2017).

[32] PIConGPU website, http://picongpu.hzdr.de (accessed 03/2017).

[33] J. D. Teresco, L. P. Ungar, A comparison of Zoltan dynamic load balancers for adaptive computation, Tech. Rep. CS-03-02, Williams College Department of Computer Science (2003).

[34] Intel Corp., Intel MPI Library for Linux OS Reference Manual, version 5.1.2, https://software.intel.com/en-us/intel-mpi-library (accessed 03/2017).

[35] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, Zoltan home page, http://www.cs.sandia.gov/Zoltan (accessed 03/2017).

[36] M. Lieber, R. Wolke, V. Grützun, M. S. Müller, W. E. Nagel, A framework for detailed multiphase cloud modeling on HPC systems, in: Proc. ParCo 2009, Vol. 19 of Adv. Par. Com., IOS Press, 2010, pp. 281–288. doi:10.3233/978-1-60750-530-3-281.

[37] A. Debus, M. Bussmann, R. Pausch, U. Schramm, R. Widera, Simulating Radiation from Laser-wakefield Accelerators, in: ICAP2012, 2012, http://accelconf.web.cern.ch/accelconf/icap2012/talks/tusbc1_talk.pdf (accessed 03/2017).