

EXTREME-SCALE BLOCK-STRUCTURED ADAPTIVE MESH REFINEMENT

FLORIAN SCHORNBAUM* AND ULRICH RÜDE*†

Abstract. In this article, we present a novel approach for block-structured adaptive mesh refinement (AMR) that is suitable for extreme-scale parallelism. All data structures are designed such that the size of the meta data in each distributed processor memory remains bounded independent of the processor number. In all stages of the AMR process, we use only distributed algorithms. No central resources such as a master process or replicated data are employed, so that an unlimited scalability can be achieved. For the dynamic load balancing in particular, we propose to exploit the hierarchical nature of the block-structured domain partitioning by creating a lightweight, temporary copy of the core data structure. This copy acts as a local and fully distributed proxy data structure. It does not contain simulation data, but only provides topological information about the domain partitioning into blocks. Ultimately, this approach enables an inexpensive, local, diffusion-based dynamic load balancing scheme.

We demonstrate the excellent performance and the full scalability of our new AMR implementation for two architecturally different petascale supercomputers. Benchmarks on an IBM Blue Gene/Q system with a mesh containing 3.7 trillion unknowns distributed to 458,752 processes confirm the applicability for future extreme-scale parallel machines. The algorithms proposed in this article operate on blocks that result from the domain partitioning. This concept and its realization support the storage of arbitrary data. In consequence, the software framework can be used for different simulation methods, including mesh-based and meshless methods. In this article, we demonstrate fluid simulations based on the lattice Boltzmann method.

Key words. adaptive mesh refinement, dynamic load balancing, supercomputing, scalable parallel algorithms, parallel performance, lattice Boltzmann method, AMR, HPC, LBM

AMS subject classifications. 68W10 68W15 68U20 65Y05 65Y20 76P05

1. Introduction. With the availability of modern computers and the continuing increase in their computational performance, the simulation of physical phenomena plays an important role in many areas of research. These simulations in, e.g., fluid dynamics, mechanics, chemistry, or astronomy often require a large amount of computational resources and therefore rely on massively parallel execution on state-of-the-art supercomputers.

1.1. Adaptive Mesh Refinement. If only parts of the simulation domain require high resolution, many advanced models rely on grid refinement in order to focus the computational resources in those regions where a high resolution is necessary. For time dependent, instationary simulations and in particular for many fluid dynamics simulations of complex flows, a priori static grid refinement cannot capture the inherently dynamic behavior. In such situations, adaptive mesh refinement (AMR) must be used to repeatedly adapt the grid resolution to the current state of the simulation. For AMR to work efficiently on distributed parallel systems, the underlying data structures must support dynamic modifications to the grid, migration of data between processes, and dynamic load balancing.

In this article, we present a novel approach for a block-structured domain partitioning that supports fully scalable AMR on massively parallel systems. We build here on the parallelization concepts, data structures, algorithms, and computational models introduced in [60]. The hierarchical approach of [60] consists of a distributed

*Chair for System Simulation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany (florian.schornbaum@fau.de, ulrich.ruede@fau.de).

†Parallel Algorithms Project, CERFACS, Toulouse, France.

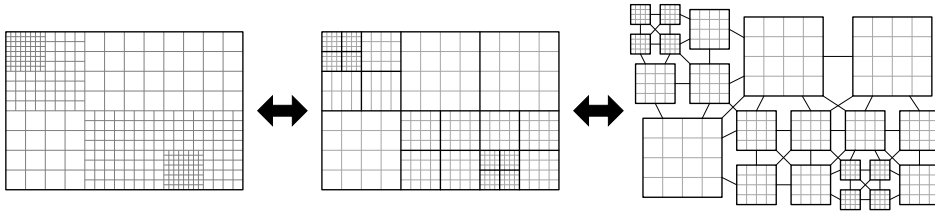


FIG. 1. The figure on the left outlines the entire global mesh of an exemplary simulation that uses local mesh refinement in the upper left and the lower right corner of the domain. The figure in the middle shows the corresponding partitioning of the simulation domain into blocks, where each block contains a uniform Cartesian grid. The underlying data structure does not contain global information about all blocks of the simulation, but for each block, it stores a reference to all neighbor blocks visualized in the figure on the right. In parallel simulations, these blocks are distributed to all available processes as outlined in Figure 2. This kind of a block-structured domain partitioning serves as the basis of the AMR procedure presented in this article. For more details about the domain partitioning concepts and their parallel implementation, we refer to [60]. Although the code fully supports 3D and all simulations in Section 3 are performed in 3D, this illustration as well as all later illustrations are in 2D in order to facilitate the schematic visualization of the methods.

forest of octrees-like domain partitioning into blocks similar to [8, 12], with each block representing a container for arbitrary simulation data. When the blocks are used to store uniform Cartesian grids, this leads to a piecewise uniform, globally nonuniform mesh as outlined in Figure 1. The data structure that manages the block partitioning requires, and thus enforces, 2:1 balance between neighboring blocks. The 2:1 balance constraint requires the levels of neighboring blocks to differ by at most 1. This has, for example, been used in [60] to construct a parallelization scheme for the lattice Boltzmann method (LBM) [2, 18] on nonuniform grids. The corresponding implementation shows excellent and scalable performance for two architecturally different petascale supercomputers. Weak scalability with up to 1.8 million threads and close to one trillion grid cells is demonstrated. In a strong scaling scenario, the implementation reaches one thousand time steps per second for 8.5 million lattice cells.

In the present article, we develop algorithms that enable scalable, parallel AMR functionality by operating on the aforementioned distributed data structures. This requires algorithms that are responsible for dynamically adapting the domain partitioning, performing dynamic load balancing, and migrating data between processes. All concepts and algorithms described in this article are, however, only based on the blocks that result from domain partitioning. Since the blocks act as containers and since the data that is stored inside these containers is kept conceptually distinct, our new implementation of AMR does not only apply to simulations that rely on an underlying mesh, but can, for example, also be employed for particle-based methods. In particular, the implementation can also be used to load balance multibody simulations and granular flow scenarios such as in [53]. We emphasize here that this generality of the distributed data structures and algorithms is key to reaching fully scalable, adaptive multiphysics simulations that require the coupling of different solvers and different simulation methods in a massively parallel environment [3, 32].

1.2. Related Work. Software frameworks for block-structured adaptive mesh refinement (SAMR) have been available for the last three decades. Recently, many SAMR codes have been compared in terms of their design, capabilities, and limitations in [22]. All codes covered in this survey can run on large-scale parallel systems, are written in C/C++ or Fortran, and are publicly available. Moreover, almost all

these software packages can, among other approaches, make use of space filling curves (SFCs) during load balancing. Some of the SAMR codes are focused on specific applications and methods, while others are more generic and provide the building blocks for a larger variety of computational models. The codes also differ in the extent to which their underlying data structures require the redundant replication and synchronization of meta data among all processes. Meta data that increases with the size of the simulation is often an issue on large-scale parallel systems, and eliminating this need for global meta data replication is a challenge that all SAMR codes are facing.

Both BoxLib [9] and Chombo [1], with Chombo being a fork of BoxLib that started in 1998, are general SAMR frameworks that are not tied to a specific application. Both, however, rely on a patch-based AMR implementation that is subject to redundant replication of meta data across all processes. Another generic framework for SAMR is Cactus [14, 31] with mesh refinement support through Carpet [16, 58]. According to [22], FLASH [23, 28] with AMR capabilities provided by the octree-based PARAMESH package [42, 50] was among the first SAMR codes to eliminate redundant meta data replication. Besides FLASH, the authors of [22] conclude that Uintah [51, 63] has gone the farthest in overcoming the limitations of replicating meta data and adapting the software basis to current and future architectures. Uintah employs task-based parallelism with dynamic task scheduling similar to Charm++ [38]. Recently, however, the developers behind Enzo [10, 61], an SAMR code mainly focused on astrophysical and cosmological simulations, have also analyzed that the increasing memory consumption due to data replication is the cause of Enzo’s most significant scaling bottlenecks [8]. A subsequent redesign of the software basis resulted in Enzo-P/Cello [8, 24]. Enzo-P/Cello uses a domain partitioning based on a fully distributed forest of octrees similar to the methods studied in this article. Moreover, Enzo-P/Cello is build on top of Charm++ and therefore program flow also follows a task-based parallelism model similar to Uintah. The idea of first partitioning the simulation domain into blocks using an octree approach and later creating Cartesian meshes inside these blocks was recently also adopted by a new software project: ForestClaw [12]. ForestClaw uses the p4est library [13] for domain partitioning, a library that already demonstrated scalability to massively parallel systems and, being based on a distributed forest of octrees implementation, also shares similarities with our approach. For more details on many of these SAMR codes, we refer to the survey in [22].

Specifically for the LBM, several AMR approaches have been published. [62] employs octrees on a cell level in order to realize cell-based AMR. [65] describes an SAMR implementation based on the PARAMESH package. Other SAMR approaches for the LBM are presented in [47], which makes use of the Peano framework [11] that is based on a generalized spacetree concept, and [26], which relies on an octree-like domain partitioning. [26], however, solely focuses on the AMR methodology and does not provide information about parallelization capabilities of the underlying code. Recently, the AMR scheme of [26] was further extended to support two-phase flow in [25]. First results of a cell-based AMR implementation build on top of the p4est library are presented in [40]. This implementation, however, does not yet support levelwise load balancing as it is necessary for balanced simulations in the context of the AMR schemes for the LBM. A priori static grid refinement in parallel simulations based on the LBM is studied in [29, 35, 59]. To our best knowledge, other popular simulation codes based on the LBM such as Palabos [39, 49], OpenLB [27, 36, 48], LB3D [33, 41], HemeLB [34], HARVEY [55], or LUDWIG [21] are at a state that they

either do not support grid refinement, only support grid refinement for 2D problems, or have not yet demonstrated large-scale simulations on nonuniform grids. The implementation for the LBM on nonuniform grids that we are using [60] consists of a distributed memory parallelization of the algorithm described in [56] combined with the interpolation scheme suggested by [17]. For more information about various grid refinement approaches for the LBM, we refer to our summary of related work in [60].

1.3. Contribution and Outline. The contribution of this article is a pipeline for SAMR that relies on a temporary, lightweight, shallow copy of the core data structure that acts as a proxy and only contains topological information without any additional computational data. The data structure imposes as few restrictions as possible on the dynamic load balancing algorithm, thus enabling a wide range of different balancing strategies, including load balancing implementations that rely on SFCs and graph-based balancing schemes. Particularly, this lightweight proxy data structure allows for inexpensive iterative balancing schemes that make use of fully distributed diffusion-based algorithms. Besides support for distributed algorithms, the data structures themselves are stored distributedly. Each process only stores data and meta data for process-local blocks, including information about direct neighbor blocks in the form of block identifier (ID) and process rank pairs. No data or meta data is stored about the blocks that are located on other processes [60]. Consequently, if a fixed number of grid cells are assigned to each process, the per process memory requirement of a simulation remains unchanged and constant, independent of the total number of active processes. Distributed storage of all data structures is the foundation for scalability to extreme-scale supercomputers. As such, the implementation presented in this article is an example for a state-of-the-art SAMR code. Moreover, to the best knowledge of the authors, the total number of cells that can be handled and the overall performance achieved significantly exceed the performance data that has been published for the LBM on nonuniform grids to date [29, 35, 40, 44, 47, 59, 65].

The remainder of this article is organized as follows. Section 2 contains a detailed description of our SAMR pipeline, from marking blocks for refinement and coarsening to load balancing the new domain partitioning and finally migrating the simulation data between processes. Special focus is put on the design and realization of a lightweight proxy data structure and its implications especially on the dynamic load balancing algorithm. Consequently, the section also provides a detailed discussion about the applicability, the advantages, and the disadvantages of and the differences between a dynamic load balancing scheme based on SFCs and an algorithm that is based on a fully distributed diffusion approach. In Section 3, we present several benchmarks that demonstrate the performance and scalability of our SAMR approach on two petascale supercomputers. We conclude the article in Section 4.

2. Dynamic domain repartitioning. Our AMR algorithms are built on a forest of octrees-like domain partitioning into blocks. The resulting tree structure, however, is not stored explicitly, but it is implicitly defined by a unique identification scheme for all blocks. Additionally, every block is aware of all of its spatially adjacent neighbor blocks, effectively creating a distributed adjacency graph for all blocks (see Figure 1). Consequently, the new software framework supports the implementation of algorithms that operate on the tree-like space decomposition as well as on the distributed graph representation. All concepts, algorithms, and data structures presented in this article are implemented in the open source software framework

WALBERLA [30, 64]. The entire code is written in C++¹ and besides parallelization with only OpenMP for shared memory systems or only with MPI for distributed memory, it also supports hybrid parallel execution where multiple OpenMP threads are executed per MPI process. Moreover, the framework does not impose any constraints on the algorithms concerning program flow. As a result, methods that require more time steps on finer levels as well as methods that perform one synchronous time step on all levels can be implemented on top of the underlying data structures. For more details about our domain partitioning and parallelization concepts and the specifics of the data structures, we refer to section 3 of [60].

In the following five subsections, we outline the different steps of our SAMR process that are necessary to dynamically adapt the domain partitioning and rebalance and redistribute the simulation data. The implementation of these algorithms follows the open/closed software design principle that states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” [43, 45]. Consequently, key components of the algorithms are customizable and extensible via user-provided callback functions. These callbacks are fundamental to our software architecture. They allow to adapt the core algorithms and data structures to the specific requirements of the underlying simulation without any need to modify source code in our AMR framework.

When blocks are exchanged between processes, for example, the framework does not perform the serialization of the block data since this data can be of arbitrary type. Simulations are allowed to define their own C++ classes and add instances of these classes as data to the blocks. As a result, it may be impossible for the framework to know how to serialize the concrete block data. Therefore, when data is added to the blocks, also corresponding serialization functions must be registered. More precisely, any callable object that adheres to a certain signature can be registered. These callable objects can be C-style function pointers, instances of classes that overload `operator()`, or lambda expressions. Internally, they are bound to `std::function` objects². The code in the framework that manages the exchange of blocks uses these callable objects for the serialization of the data to a byte stream. This principle of having callback functions which are registered at simulation setup in order to be later executed by specific parts of the framework is also employed in various essential parts of the AMR pipeline. These callbacks make the framework flexible and extensible. They are, for example, used in order to decide which blocks need to be refined (see Section 2.2) and in order to represent the concrete dynamic load balancing algorithm (see Section 2.4).

2.1. Four-step procedure. The entire AMR pipeline consists of four distinct steps. In the first step, blocks are marked for refinement and coarsening. In the second step, this block-level refinement information is used in order to create a second, lightweight, proxy data structure that only contains this new topological information but no simulation data. The proxy blocks can be assigned weights that correspond to the expected workload generated by the actual simulation data. In the third step of the AMR procedure, the proxy data structure is then load balanced and proxy blocks are redistributed between all available processes. In the fourth and final step, the actual, still unmodified data structure containing the simulation data is adapted

¹including the subset of features of C++11 and C++14 that are supported by all major C++ compilers (GCC, Clang, Intel, Microsoft Visual C++, IBM XL C/C++)

²For compatibility with compilers that do not yet fully support C++11, `boost::function` from the boost library [7] can be used instead of `std::function`.

ALGORITHM 1

Program flow of the AMR scheme. The entire AMR pipeline consists of four distinct steps. First, blocks are marked for refinement/coarsening in [Line 2](#). Then, a proxy data structure representing the new domain partitioning is created in [Line 4](#) and subsequently balanced in [Line 5](#). Finally, the actual simulation data is adapted and redistributed in [Line 6](#). Illustrations corresponding to these four steps are presented in [Figures 3 to 5 and 7](#).

```

1 Function DYNAMICREPARTITIONING
2   call BLOCKLEVELREFINEMENT /* blocks are marked for refinement/coarsening */
   /* (by a user-provided callback and the framework that enforces 2:1 balance) */
3   while blocks marked for refinement/coarsening exist or block weights must be
   reevaluated and blocks must be redistributed do
4     call PROXYINITIALIZATION /* construction of the proxy data structure */
5     call DYNAMICLOADBALANCING /* redistribution of proxy blocks */
6     call DATAMIGRATION /* migration and refinement/coarsening of the ... */
   /* ... actual simulation data according to the state of the proxy structure */
7     if multiple AMR cycles are allowed then
8       | call BLOCKLEVELREFINEMENT /* same as Line 2 */
9     end
10  end
11 end

```

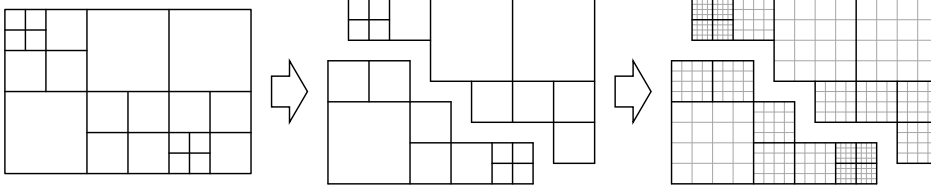


FIG. 2. Example domain partitioning for a simulation with blocks containing 16 cells (4×4) each. These blocks are distributed among two processes as indicated by the separation of the data into two parts in the figures in the middle and on the right. This example partitioning illustrates the initial state of a simulation before AMR is triggered. The AMR pipeline then starts with distributed block-level refinement/coarsening in [Figure 3](#), continues with the creation and load balancing of a lightweight proxy data structure in [Figures 4 and 5](#), and ends with the actual migration and refinement/coarsening of the grid data in [Figure 7](#).

according to the state of the proxy data structure, including refinement/coarsening and redistribution of the simulation data.

This four-step AMR procedure is outlined in [Algorithm 1](#). As indicated by [Algorithm 1](#), the reevaluation of block weights that potentially results in a redistribution of all blocks can be triggered without the need of block-level refinement or coarsening, meaning the entire pipeline can be forced to be executed without any blocks being marked for refinement or coarsening. Moreover, the implementation allows multiple AMR cycles before the simulation resumes. As a result, blocks can be split/merged multiple times during one dynamic repartitioning phase.

The four steps of our AMR scheme are discussed in more detail in the following four subsections. The entire AMR pipeline is also illustrated in [Figures 3 to 5 and 7](#) starting with an example initial domain partitioning outlined in [Figure 2](#).

2.2. Distributed block-level refinement. The objective of the block-level refinement and coarsening phase is to assign a target level to each block such that

$$\ell_{\text{target}} \in \{\ell_{\text{current}} - 1, \ell_{\text{current}}, \ell_{\text{current}} + 1\}.$$

In order to assign target levels to all blocks, the block-level refinement and coarsening phase is divided into two steps. First, an application-dependent callback function is evaluated on every process. The task of this function is to assign a target level to every block handled by the process. As such, initially marking blocks for refinement or coarsening is a perfectly distributed operation that can be executed in parallel on all processes.

These new target levels, as set by an application-dependent callback, might violate the 2:1 balance constraint of the domain partitioning. Consequently, after the application-dependent callback was evaluated, the framework guarantees 2:1 balance by first accepting all blocks marked for refinement and subsequently potentially forcing additional blocks to split in order to maintain 2:1 balance. Afterwards, blocks marked for coarsening are accepted for merging if, and only if, all small blocks that merge to the same large block are marked for coarsening and 2:1 balance is not violated. Consequently, blocks marked for refinement by the application-dependent callback are guaranteed to be split, whereas blocks marked for coarsening are only merged into a larger block if they can be grouped together according to the octree structure.

This process of guaranteeing 2:1 balance can be achieved by exchanging the target levels of all process-local blocks with all neighboring processes. Afterwards, this information can be used to check if process-local blocks must change their target levels, i.e., check if they must be forced to split or are allowed to merge, due to the state of neighboring blocks. This process of exchanging block target levels with neighbors must be repeated multiple times. The number of times every process must communicate with all of its neighbors is, however, limited by the depth of the forest of octrees, i.e., the number of levels available in the block partitioning. Consequently, the runtime of this first stage in the AMR pipeline scales linearly with the number of levels in use, but its complexity is constant with regard to the total number of processes. For the rest of this article, if the complexity of an algorithm is constant with regard to the number of processes, we refer to that algorithm (and the algorithm's runtime) as being independent of the number of processes.

The block-level refinement and coarsening phase is illustrated in [Figure 3](#). The iterative process of evaluating block neighborhoods multiple times means that the medium sized blocks in [Figure 3\(4\)](#) are only accepted for coarsening after their smaller neighbor blocks were accepted for coarsening in [Figure 3\(3\)](#).

Since perfectly distributed algorithms and data structures require that every process possesses only local, but no global knowledge, every process must assume that on distant, non-neighbor processes blocks are marked to be split or to be merged. Consequently, all processes must continue with the next step in the four-step AMR procedure, even if there are no changes to the block partitioning. The actual implementation of the block-level refinement and coarsening phase therefore uses two global reductions of a boolean variable as a means of optimizing the execution time as follows: Immediately after the application-dependent callback is executed, the first reduction can be used to abort the entire AMR procedure early if no blocks have been marked for refinement or coarsening. Even if some blocks are marked for coarsening, they all might violate the requirements that are necessary for merging. Therefore, a second reduction at the end of the block-level refinement and coarsening phase provides another opportunity to terminate the AMR process early. On current petascale supercomputers, the benefit of aborting the AMR algorithm often outweighs the costs of the two global reductions.

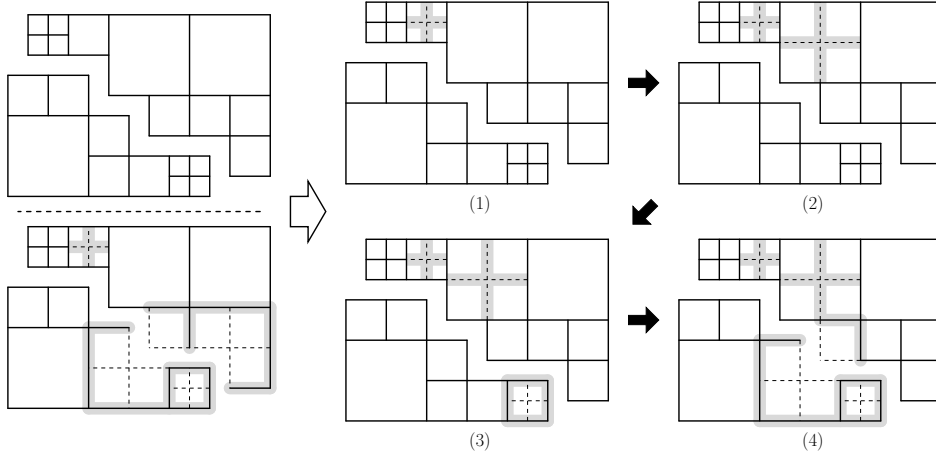


FIG. 3. Starting from the domain partitioning outlined in Figure 2, during the distributed block-level refinement/coarsening phase, an application-dependent callback function determines which blocks must be split and which blocks may be merged (figure on the bottom left). Blocks are marked for coarsening independent of their neighbors. After the evaluation of the callback function, all blocks marked for refinement are accepted (figure (1) on the right) and additional blocks are registered for refinement by an iterative process in order to maintain 2:1 balance (2). Finally, another iterative procedure accepts blocks for coarsening if all small blocks that merge to the same large block are marked for coarsening and if 2:1 balance is not violated (figures (3) and (4)).

2.3. Proxy data structure. The AMR implementation in this article is characterized by using a light-weight proxy data structure to manage load balancing and the dynamic data redistribution in an efficient way. Consequently, the first phase of the AMR procedure outlined in the previous section only assigns target levels to all blocks, but it does not yet apply any changes to the block partitioning. During the second step of the AMR procedure, these target levels are used in order to create a second block partitioning that conforms with the new topology as defined by the target levels. This second data structure acts as a proxy for the actual, still unmodified simulation data structure. For the rest of this article, we will use the term *proxy data structure* as opposed to the primary *actual data structure*. Similarly, we distinguish between *proxy blocks* and *actual blocks*.

The proxy data structure only manages the process association and the connectivity information of all of its blocks, but it does not store any simulation data. Furthermore, during creation of the proxy data structure, links are established between all proxy blocks and their corresponding actual blocks. Consequently, every proxy block always knows the process where its corresponding actual block is located, and vice versa. Particularly during the third step of the AMR procedure when proxy blocks might migrate to different processes, maintaining these bilateral links is vital for the success of the entire AMR scheme. Typically, these links are represented by a *target process* that is stored for each actual block. This target process is the process owning the corresponding proxy block. Additionally, there is a *source process* stored for each proxy block. Analogously, this is the process on which the corresponding actual block is located on. If an actual block corresponds to eight smaller proxy blocks due to being marked for refinement, the actual block stores eight distinct target processes, one for each proxy block. Similarly, if eight actual blocks correspond to one larger proxy block due to all actual blocks being marked for coarsening, the proxy block

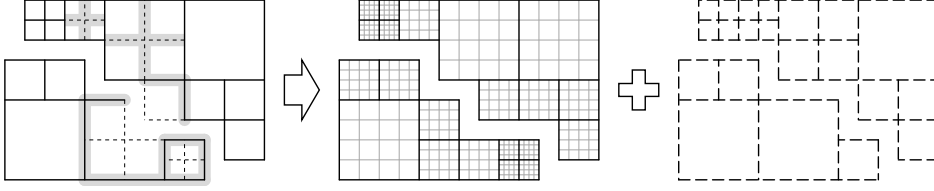


FIG. 4. Using the results from the block-level refinement phase outlined in Figure 3, a lightweight proxy data structure that conforms with the new topology is created. As a result, every process stores the current, unmodified data structure that maintains the simulation data (figure in the middle) as well as the temporarily created proxy data structure that does not store any simulation data (figure on the right). Additionally, every block in each of these two data structures maintains a link/links (which are not visualized in the illustration) to the corresponding block(s) in the other data structure. Initially, the proxy data structure is in an unbalanced state. In this example, most of the blocks, including all of the smallest blocks, are initially assigned to the same process. Therefore, load balancing the proxy data structure is the next step in the AMR pipeline (see Figure 5).

stores eight source processes. For various figures in this article, these eight-to-one relationships in the 3D implementation correspond to four-to-one relationships in the 2D illustrations.

The creation of the proxy data structure is outlined in Figure 4. As illustrated by Figure 4, for multiple blocks to be merged into one larger block, the smaller blocks do not have to be located on the same process. Also, the creation of all proxy blocks, including the initialization of target and source processes for proxy and actual blocks, is a process-local operation. Only when setting up the connectivity information for the proxy blocks, communication with neighboring processes is required. Consequently, the runtime of this second step of the AMR procedure, the creation of the proxy data structure, is, just as the block-level refinement and coarsening phase, also independent of the total number of processes.

2.4. Dynamic load balancing. The third step in the AMR scheme is the dynamic load balancing phase. Here, the goal is to redistribute the proxy blocks among all processes such that the proxy data structure is in balance. Just like the block-level refinement phase (see Section 2.2), the dynamic load balancing procedure is also divided into two parts. First, a simulation-dependent, possibly user-provided callback function determines a target process for every proxy block. This callback function represents the actual, customizable load balancing algorithm. Once this callback is finished and returns, the framework part takes over execution and sends proxy blocks to different processes according to their just assigned target processes. During this migration process, the framework also maintains the bilateral links between proxy blocks and actual blocks.

To be exact, the callback function of the load balancing stage must perform three distinct tasks. It must assign a target process to every proxy block, it must notify every process about the proxy blocks that are expected to arrive from other processes, and it must return whether or not another execution of the dynamic load balancing procedure must be performed immediately after the migration of the proxy blocks. Requesting another execution of the entire dynamic load balancing step enables iterative balancing schemes, as we will use below. Executing the dynamic load balancing procedure multiple times means proxy blocks are also exchanged multiple times. Transferring a proxy block to another process, however, only requires to send a few bytes of data (block ID, the source process of its corresponding actual block,

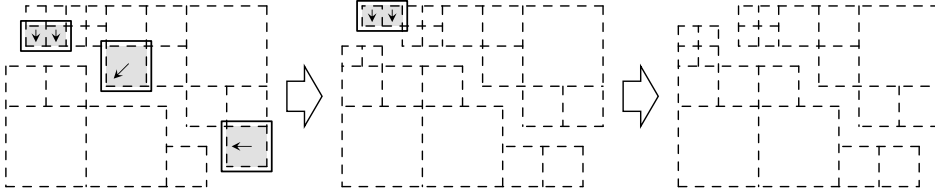


FIG. 5. During dynamic load balancing, an application-dependent callback function determines a target process for every proxy block. The framework then performs the migration of proxy blocks between processes, including an update of all links between these proxy blocks and their counterparts in the simulation data structure. As illustrated in the figure, this process of first determining target processes for the proxy blocks and then migrating them accordingly can be repeated multiple times, enabling iterative, diffusion-based load balancing schemes. In this example, balance is achieved after two steps. Nevertheless, globally load balancing the proxy data structure in one step is also possible.

and the block IDs of its neighbors). As a result, the proxy transfers are inexpensive communication operations.

The implementation is customizable to the requirements of the underlying simulation and the user-defined load balancing algorithm. It is possible to augment each proxy block with additional data that is also transferred when proxy blocks migrate to other processes. The additional proxy data can, for example, be used to encode the workload/weight of a block as it is used in the load balancing algorithm. Generally, the extensibility of the proxy block data is similar to the extensibility of the simulation block data.

The dynamic load balancing procedure is illustrated in Figure 5. The migration of proxy blocks, including the preservation of all links between proxy blocks and actual blocks, only requires point-to-point communication between different process pairs. As a result, the runtime of the framework part of the dynamic load balancing procedure is independent of the total number of processes. Consequently, the runtime and scalability of the entire dynamic load balancing procedure is mainly determined by the runtime and scalability of the callback function. Ultimately, for large-scale simulations, the chosen dynamic load balancing algorithm determines the runtime and scalability of the entire AMR scheme.

The following two subsections present two different dynamic load balancing approaches as they are currently provided by the framework. Section 2.4.1 outlines a balancing scheme based on SFCs that requires global data synchronization among all processes, whereas Section 2.4.2 presents a balancing algorithm built on a fully distributed, local diffusion-based redistribution scheme. The runtime of the latter is independent of the total number of processes. Since the dynamic load balancing algorithms only operate on proxy blocks, the terms “proxy blocks” and “blocks” are used synonymously in the following two subsections.

Both implementations also provide the ability to balance the blocks per level. For LBM-based simulations on nonuniform grids as they are used by all benchmarks and the example application in Section 3, levelwise balancing of the blocks is essential for achieving good performance. Only blocks balanced per level perfectly match the structure of the underlying algorithm [60]. Consequently, providing the ability to balance the blocks per level is a necessary requirement for any balancing algorithm that is used for simulations based on the LBM.

The SFC as well as the diffusion-based approach presented in the next two subsections are both injected into the SAMR framework as callback functions that adhere

to the requirements for load balancing algorithms for the proxy data structure described at the beginning of this section. Since the actual load balancing algorithm is injected into the AMR pipeline as a callback function, other dynamic load balancing strategies can be incorporated without the need to modify framework code. Consequently, future work that builds on the SAMR framework presented in this article will investigate the applicability of dynamic load balancing algorithms provided by other, specialized libraries like ParMETIS [52, 57], Zoltan [6, 66], or PT-Scotch [19, 54].

2.4.1. Space filling curves. Many SAMR codes employ SFCs for load balancing. In general, SFCs map multidimensional data to one dimension while preserving good spatial locality of the data. As such, they can be used to define a global ordering for all the octants of an octree. Consequently, SFCs allow to construct an ordered list of all the blocks stored in our data structure. This list can be used for load balancing by first dividing the list into as many pieces of consecutive blocks as there are processes and then assigning one of these pieces to each process. For this approach to work on homogeneous systems, the sum of the weights of all blocks in each piece must be identical. Consequently, if all blocks have the same weight, i.e, generate the same workload, each piece must consist of the same number of blocks. The generalization to a forest of octrees where each root block corresponds to an octree [60] can be realized by defining a global ordering for all root blocks [13].

The current version of our code features SFC-based load balancing routines that make use of either Morton [46] or Hilbert [37] order. With Hilbert order, consecutive blocks are always connected via faces, whereas with Morton order, several consecutive blocks are only connected via edges or corners, with some consecutive blocks not being connected to each other at all. Hilbert order, therefore, results in better spatial locality for blocks assigned to the same process than Morton order. Both curves can be constructed by depth-first tree traversal. For Morton order, child nodes are always visited in the same order while descending the tree, whereas for Hilbert order, the order in which child nodes are traversed depends on their position within the tree. For the construction of the Hilbert curve, lookup tables exist that specify the exact traversal order [15]. Consequently, the construction of a SFC based on Hilbert instead of Morton order only results in an insignificant overhead. Since in our implementation, the block IDs, which uniquely identify all blocks within the distributed data structure, can be represented as numbers that are related to the Morton order similar to the identification scheme of [13], simply sorting all blocks by ID also results in a Morton ordering of the blocks.

Regardless of whether Morton or Hilbert order is used during the SFC-based load balancing phase, the construction of the curve is built on a global information exchange among all processes as it is also used by other established software [13]. This global data synchronization is usually best realized with an `allgather` operation. If all blocks share the same weight and if all blocks are treated equally regardless of their level, then globally synchronizing the number of blocks stored on each process is enough to determine, locally on every process, where blocks need to migrate in order to achieve a balanced redistribution along the curve. Consequently, this approach results in the synchronization and global replication of one number (typically 1 byte is enough) per process.

If the blocks must be balanced per level, the SFC is used to construct one list of blocks for every level. Load balance is then achieved by distributing each list separately. During dynamic refinement and coarsening, this per-level ordering is mixed up (see Figure 6). As a result, restoring the order for every level and the subsequent

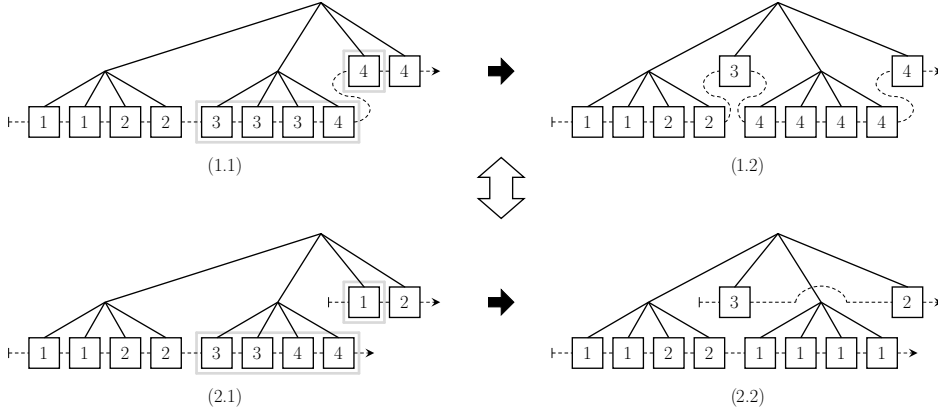


FIG. 6. Example of a SFC-based distribution to 4 processes (as indicated by the small numbers inside the blocks). In 1.1 and 1.2, the blocks are distributed to the 4 processes following the global ordering as given by the curve. If blocks split/merge due to refinement/coarsening (indicated by the gray boxes in 1.1), the new blocks always align themselves correctly (1.2): Blocks along the new curve will always still be in order, but rebalance may be required. Since all blocks are still in order, this rebalancing operation is cheap and only requires knowledge about the number of blocks on every process. If, however, blocks on different levels must be balanced separately (as is illustrated in 2.1 and 2.2), new blocks that are the result of refinement or coarsening may fall out of line (2.2): The lists of blocks for every level are not in order anymore. In contrast to 1.2, rebalancing 2.2 also requires a complete reordering, which is a far more expensive operation in terms of the amount of information that must be synchronized between all processes.

TABLE 1

Typical amount of data that must be globally replicated and synchronized to all processes when using SFC-based dynamic load balancing.

		blocks must be balanced per level	
		no	yes
every block has the same weight	yes	1 byte per process	4-8 bytes per block
	no	1-4 bytes per block	5-12 bytes per block

rebalancing is not cheap anymore and requires an **allgather** of all block IDs (typically 4 to 8 bytes per block). Knowing the ID of every block then allows each process to, locally, reconstruct and redistribute all SFC-based lists of blocks, sorted by level.

If, additionally, blocks are allowed to have individual weights, the amount of data that must be synchronized increases further by 1 to 4 bytes for every block (cf. Table 1). Ultimately, SFC-based dynamic load balancing requires global synchronization regardless of the requirements of the underlying simulation. The communication time as well as the memory consumption per process therefore increases linearly with the number of processes. If all blocks share the same weight and if per-level balancing is not necessary, SFC-based dynamic load balancing may, however, still be feasible with several millions of processes.

2.4.2. Diffusion-based approach. The idea of diffusion-based load balancing schemes is to use a process motivated by physical diffusion in order to determine, for every process x , the neighboring processes to which local blocks of x must be transferred. The process of first executing this diffusion-based rebalancing and then migrating proxy blocks is repeated multiple times, leading to an iterative load bal-

ancing procedure that allows blocks to migrate to distant processes. Diffusion-based load balancing schemes may not always achieve a perfect global balance, but we can expect that they will quickly ($\hat{=}$ few iterations) eliminate processes with high load. Eliminating peak loads is essential to avoid bottlenecks and to achieve good parallel efficiency.

Each step of the diffusion-based algorithm only relies on next-neighbor communication. As a result, if the number of iterations is fixed, the runtime as well as the memory consumption is independent of the total number of processes. Whereas the SFC-based balancing scheme relies on an octree-like domain partitioning, the diffusion algorithm operates on the distributed process graph. This process graph indicates how processes are connected to each other. We refer to two processes i and j as being connected/neighbors if at least one block on process i is connected via a face, edge, or corner with a block on process j . This graph is realized in distributed form since every process maintains only links to its neighbor processes.

The implementation of the diffusion-based balancing algorithm consists of two main parts. In the first part, a diffusion scheme originally proposed by [20] is used to determine the flow of process load f_{ij} that is required between all neighboring processes i and j in order to achieve load balance among all processes. In the second part, the results from the first part are then used to decide which blocks must be moved to which neighbor process by matching block weights to process inflow and outflow information. If load balancing per level is required, all data is calculated for each level based only on the blocks of the corresponding level. However, besides calculating data separately for each level, program flow is otherwise identical to an application that does not require per-level balancing. Consequently, all algorithms in this section can be easily used also to perform load balancing per level.

The algorithm is outlined in [Algorithm 2](#). The diffusion scheme starts by calculating the process load w_i of every process by summing up the weights of all process-local blocks. The flow between all neighboring processes is then determined by an iterative process. First, every process i calculates the current flow f'_{ij} between itself and every neighbor process j with

$$f'_{ij} = \alpha_{ij} \cdot (w_i - w_j) \quad [20],$$

where α_{ij} follows the definition of [5] which allows α_{ij} to be determined with next-neighbor communication only. Hence,

$$\alpha_{ij} = \frac{1}{\max(d_i, d_j) + 1},$$

with d_i and d_j denoting the number of neighbor processes of i and j , respectively. The current flow f'_{ij} is then used to adapt the process loads w_i and w_j of processes i and j . However, no blocks are yet exchanged. This procedure of calculating flows and adapting process loads accordingly is repeated for a fixed number of iterations. We refer to these iterations as “flow iterations” as opposed to the number of “main iterations” of the load balancing stage. Consequently, the diffusion-based load balancing approach represents an iterative load balancing scheme with nested iterations: For every main iteration, first a fixed number of flow iterations is executed and then proxy blocks are exchanged between neighboring processes. The flow f_{ij} , which is used to determine the blocks that must be exchanged between processes i and j , eventually results from the summation of all f'_{ij} . A value for f_{ij} greater than zero indicates outflow, whereas $f_{ij} < 0$ indicates inflow.

ALGORITHM 2

The algorithm outlines the two-part diffusion-based load balancing scheme. First, the algorithm determines the flow of process load f_{ij} that is required between all neighboring processes i and j in order to achieve load balance among all processes. These f_{ij} are then used to decide which blocks must be moved to which neighbor process. This decision is realized by either a pull or a push scheme outlined in Algorithms 3 and 4.

```

1 Function DIFFUSIONLOADBALANCING
2   calculate process weight  $w_i$  //  $\triangleq$  sum of all local block weights  $\triangleq$  process load
3   determine number of neighbor processes  $d_i$ 
4   exchange  $d_i$  with all neighbor processes
5   forall neighbor processes  $j$  do
6      $f_{ij} = 0$  //  $\triangleq$  flow between current process  $i$  and process  $j$ 
7      $\alpha_{ij} = \frac{1}{\max(d_i, d_j) + 1}$ 
8   end
9   repeat                                     /* calculate flow  $f_{ij}$  ... */
10    exchange  $w_i$  with all neighbor processes /* ... that is required ... */
11     $w'_i = w_i$                                /* ... between all ... */
12    forall neighbor processes  $j$  do          /* ... neighbor processes ... */
13       $f'_{ij} = \alpha_{ij} \cdot (w'_i - w_j)$       /* ...  $i$  and  $j$  ... */
14       $f_{ij} += f'_{ij}$                          /* ... in order to ... */
15       $w_i -= f'_{ij}$                            /* ... achieve balance */
16    end
17  until predefined max. number of iterations is reached // “flow” iterations
18  call DIFFUSIONPUSH( $f_{ij}$ ) or DIFFUSIONPULL( $f_{ij}$ ) in order to determine which
19    blocks are sent to which neighbor process
20  inform neighbor processes about whether or not blocks are about to be sent
21 end

```

In order to determine which blocks are sent to which neighbor process, we propose two different approaches that are referred to as “push” and “pull” scheme for the rest of this article. When using the push scheme, overloaded processes decide which blocks to push to which neighbor, whereas when using the pull scheme, underloaded processes choose neighbors from which blocks are requested. A major challenge arises from the fact that although every connection to a neighbor process is assigned a certain outflow or inflow value, these flows are almost always impossible to satisfy because they rarely match with the weights of the available blocks. Only entire blocks can be migrated. It is impossible to only send part of a block to another process. For SAMR codes, computation is usually more efficient when choosing larger blocks. For mesh-based methods like the LBM, larger blocks result in less communication overhead. With larger blocks, the compute kernels may also take better advantage of single instruction/multiple data (SIMD) instructions. Therefore, in practice, large-scale simulations are often configured to use only few blocks per process. However, in these cases where only few blocks are assigned to each process, the weight of any block might exceed every single outflow/inflow value.

Therefore, the central idea of the push scheme is to first calculate the accumulated process outflow $outflow_i$ of every process i by summing up all $f_{ij} > 0$. If the weight w_k of a block k is less than or equal to $outflow_i$, the block k is a viable candidate for being pushed to a neighbor process. Moreover, all $f_{ij} > 0$ are only used as clues as to which neighbor process j is a viable candidate for receiving blocks. Similarly, the pull scheme first calculates the accumulated process inflow $inflow_i$ of every process i by summing up all $f_{ij} < 0$. Neighbor blocks are viable candidates for being requested if their weights do not exceed the accumulated process inflow. All $f_{ij} < 0$ are only used

ALGORITHM 3

Schematic program flow of the “push” approach for determining the blocks that are sent to neighbor processes on the basis of the inflow/outflow values f_{ij} calculated for every neighbor process in the diffusion-based balancing algorithm (see Algorithm 2).

```

1 Function DIFFUSIONPUSH( $f_{ij}$ )
2    $outflow_i = 0$  // accumulated process outflow
3   forall neighbor processes  $j$  do
4     if  $f_{ij} > 0$  then  $outflow_i += f_{ij}$ 
5   end
6   while  $outflow_i > 0$  and a process  $j$  with  $f_{ij} > 0$  exists do
7     pick process  $j$  with largest value for  $f_{ij}$ 
8     construct a list of all the blocks that can be moved to process  $j$  and
9     are not yet marked for migration to another process
10    out of the previous list, pick block  $k$  that is the best fit for
11    migration to process  $j$  and whose weight  $w_k \leq outflow_i$ 
12    if such a block  $k$  exists then
13      mark block  $k$  for migration to process  $j$ 
14       $f_{ij} -= w_k$ 
15       $outflow_i -= w_k$ 
16    else
17       $f_{ij} = 0$ 
18    end
19  end
20 end

```

as clues as to which neighbor process j is a viable candidate for providing blocks.

The entire push scheme is outlined in Algorithm 3. Inside a main loop, the algorithm picks the neighbor process j that is currently associated with the highest outflow value. If the algorithm is able to identify a block that can be sent to process j , then the accumulated process outflow $outflow_i$ as well as the flow f_{ij} towards process j is decreased by the weight of the block. If multiple blocks on process i are viable candidates for being sent to process j , the algorithm picks the block that is the best fit for migration to process j . A block is considered a good fit for migration if its connection to process i is weak and/or its connection to process j is strong. If, for example, a block m is connected to only one other block on process i , but to two blocks on process j , then m is considered a better choice for migration to j than a block n which is connected to two blocks on process i and to no block on process j . Moreover, the type of the connection (face, edge, corner) is also considered while determining the connection strength.

The first part of the pull scheme, outlined in Algorithm 4, is conceptually identical to the push algorithm. Inside a main loop, the algorithm identifies blocks that must be fetched from neighbor processes in order to satisfy the accumulated process inflow. In the second part of the pull procedure, these blocks are then requested from the corresponding neighbor processes. Every process must comply with these requests. A process is only allowed to deny such a request if the same block is requested by multiple neighbors. In that case, only one request can be satisfied. Ultimately, however, all blocks that are requested are passed on to the appropriate neighbor processes.

Eventually, the application can choose the diffusion-based balancing algorithm that uses the push scheme, the pull scheme, or both the push and the pull scheme in an alternating fashion. Consequently, the program flow of the entire load balancing procedure with alternating push/pull schemes and, for example, four main iterations consists of i) Algorithms 2 and 3 followed by the migration of proxy blocks, ii) Algo-

ALGORITHM 4

Schematic program flow of the “pull” approach for determining the blocks that are sent to neighbor processes on the basis of the inflow/outflow values f_{ij} calculated for every neighbor process in the diffusion-based balancing algorithm (see Algorithm 2).

```

1 Function DIFFUSIONPULL( $f_{ij}$ )
2    $inflow_i = 0$  // accumulated process inflow
3   forall neighbor processes  $j$  do
4     if  $f_{ij} < 0$  then  $inflow_i -= f_{ij}$ 
5   end
6   send a list of all local blocks (block IDs only) and their corresponding
   weights to all neighbor processes
7   while  $inflow_i > 0$  and a process  $j$  with  $f_{ij} < 0$  exists do
8     pick process  $j$  with smallest value for  $f_{ij}$  //  $\hat{=}$  largest inflow
9     construct a list of all remote blocks that can be fetched from process  $j$ 
    and are not yet candidates for migration to the current process  $i$ 
10    out of the previous list, pick remote block  $k$  that is the best fit for
    migration from process  $j$  and whose weight  $w_k \leq inflow_i$ 
11    if such a remote block  $k$  exists then
12      locally bookmark remote block  $k$  as candidate for migration from
      process  $j$  to the current process  $i$ 
13       $f_{ij} += w_k$ 
14       $inflow_i -= w_k$ 
15    else
16       $f_{ij} = 0$ 
17    end
18  end
19  send a request to every neighbor process containing a list of all the remote
  blocks that process  $i$  wants to fetch
20  forall local blocks  $k$  do
21    if block  $k$  is requested by one neighbor process  $j$  then
22      mark block  $k$  for migration to process  $j$ 
23    else if block  $k$  is requested by multiple neighbor processes then
24      out of these processes, mark block  $k$  for migration to the neighbor
      process  $j$  with the largest value for  $f_{ij}$ 
25    end
26  end
27 end

```

gorithms 2 and 4 followed by another migration of proxy blocks, iii) Algorithms 2 and 3 followed by a third migration of proxy blocks, and iv) Algorithms 2 and 4 followed by the fourth and final migration of proxy blocks.

The actual implementation of Algorithm 2 also makes use of a global reduction for calculating the total simulation load ($\hat{=}$ sum of all block weights). This information can be used to adapt the process local inflow/outflow values with respect to the exact globally average process load. Moreover, knowledge about the total simulation load enables the algorithm to decide locally if a process is currently overloaded and whether or not balancing is required. Another global reduction then allows to synchronize this information among all processes. As a result, the entire load balancing procedure can be terminated early if all processes are already sufficiently in balance. Ultimately, this leads to a variable number of main diffusion-based balancing iterations. Applications only need to define a maximum number of iterations. But neither of these reductions is mandatory for the algorithm. Just like during the block-level refinement/coarsening phase (see Section 2.2), however, the benefits of aborting the entire load balancing procedure early can easily amortize the cost of these global reductions. In Section 3

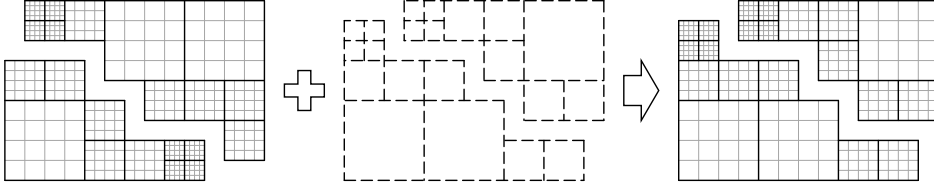


FIG. 7. After successfully balancing the proxy data structure (see Figure 5), links between all proxy blocks and their counterparts in the simulation data structure enable the refinement/coarsening of simulation data, including the migration of simulation data to different processes, in one final step. Afterwards, the temporarily created proxy data structure is destroyed. An important feature also illustrated in this example is that in order to merge multiple fine blocks into one coarse block, the fine blocks do not have to be located on the same source process.

we will demonstrate that this distributed, diffusion-based load balancing pipeline shows applicability and excellent scalability for extreme-scale parallel simulations that involve trillions of unknowns.

2.5. Data migration and refinement. The fourth and final step in the AMR pipeline is the actual refinement, coarsening, and migration of all simulation data. This final step is also outlined in Figure 7. The central idea of this data migration phase is to use the load-balanced state of the proxy data structure in order to adapt the simulation data structure accordingly. Because of the bilateral links between proxy blocks and actual blocks, the refinement, coarsening, and migration of the underlying simulation data is performed in one single step. The bilateral links also mean that for the migration of the block data only point-to-point communication via MPI between processes that exchange data is necessary. Consequently, the runtime of the final stage of the AMR pipeline scales linearly with the amount of block data/the number of blocks that need to be sent, but its complexity is constant with regard to the total number of processes in use by the simulation.

Besides the fast and inexpensive migration of proxy blocks during the dynamic load balancing phase, the refinement, coarsening, and migration of data in one single step proves to be another advantage provided by the proxy data structure. In mesh-based methods like the LBM, octree-based refinement results in eight times the number of grids cells ($\hat{=}$ eight times the amount of memory) in refined regions. As a result, if dynamic refinement of the simulation data occurs before load balancing is performed, every process must have eight times the amount of memory available as is currently allocated by the simulation in case a process is requested to refine all of its data. Consequently, in order to avoid running out of memory, only $\frac{1}{8}$ of the available memory can be used for the actual simulation and $\frac{7}{8}$ of the available memory must always be kept in reserve for the refinement procedure.

The existence of the load-balanced proxy data structure, however, allows an actual block to determine whether or not parts of its data (or even its entire data) end up on a different process after refinement. In case the refined data ends up on another process (possibly even on multiple other processes), the implementation of the data migration algorithm enables the sending source processes to only send the data that is required for the receiving target processes to correctly initialize the refined data. Allocation of the memory required for the refined data, allocation of the corresponding fine blocks, as well as the actual refinement of the data only happen on the target processes. In Figure 7, for example, the four medium-sized blocks that result from the large block in the upper center being split end up on both processes, three remain on the original

process, while one is assigned to the other process. Ultimately, our four-step AMR approach significantly reduces the amount of memory overhead. Almost the entire available memory can be used for simulation data at all times without the risk of running out of memory during the refinement phase.

As outlined in the introduction of [Section 2](#), the framework does not contain any information on how block data is serialized or deserialized directly. However, initially registering block data at the underlying data structure must include the registration of callback functions that perform the serialization/deserialization of the corresponding block data. In total, six such callbacks are required for every block data item: one pair of serialization/deserialization functions used in case a block is split, multiple blocks are merged, or a block is migrated without further modification, respectively. During the data migration phase, the framework then executes the correct callback functions in order to perform the actual migration, refinement, and coarsening of the simulation data. Refinement and coarsening are always performed via first serializing and then deserializing the corresponding blocks, even if no migration to another process is necessary. This software architecture of requiring corresponding serialization functionality to be also registered when block data is registered is essential to ensure the extensibility of our framework to arbitrary simulation data.

3. Benchmarks. For the remainder of this article, we present detailed performance results for a synthetic benchmark application that executes the entire AMR pipeline on two current petascale supercomputers. Finally, we demonstrate the applicability of the AMR procedure by outlining performance metrics for a simulation of highly dynamic, turbulent flow.

3.1. Performance. All benchmarks are run on two petascale supercomputers: JUQUEEN, an IBM Blue Gene/Q system, and SuperMUC, a x86-based system build on Intel Xeon CPUs. JUQUEEN provides 458,752 PowerPC A2 processor cores running at 1.6 GHz, with each core capable of 4-way multithreading. Based on our observations in [\[30\]](#), in order to achieve maximal performance, we make full use of multithreading on JUQUEEN by always placing four threads (which may either belong to four distinct processes, two different processes, or all to the same process) on one core. As a result, full-machine jobs consist of 1,835,008 threads. The SuperMUC system features fewer, but more powerful, cores than JUQUEEN. It is built out of 18,432 Intel Xeon E5-2680 processors running at 2.7 GHz, which sums up to a total of 147,456 cores. During the time the benchmarks were performed, however, the maximal job size was restricted to 65,536 cores. The average amount of memory per core on SuperMUC (2 GiB) is twice the average amount of memory on JUQUEEN (1 GiB).

In all of the following graphs and tables, we report the number of processor cores, i.e., the actual amount of hardware resources in use, not the number of processes. For a fixed number of cores, we can either run the benchmark with α processes (MPI only) or make use of hybrid execution with α/β processes and β OpenMP threads per process. On SuperMUC, we choose α to be equal to the number of cores. Since we make full use of multithreading on JUQUEEN, we choose α to be equal to four times the number of cores when running the benchmark on JUQUEEN. For every measurement, we will state the parallelization strategy in use: MPI only or hybrid execution with typically four ($\beta = 4$) or eight ($\beta = 8$) threads per process. Since hybrid simulations use β times fewer processes, we allocate β times more cells for each block (and process). As a result, for a given benchmark scenario and for a fixed number of processor cores (= fixed amount of hardware resources), the amount of work remains constant, independent of the parallelization strategy in use.

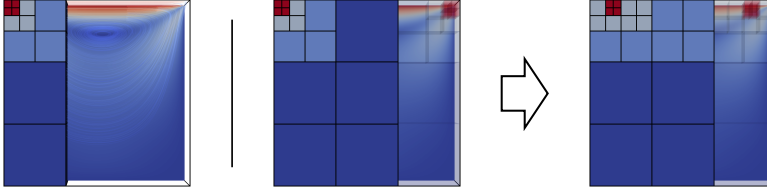


FIG. 8. Block partitioning of the benchmark application before (both figures on the left) and after (right) AMR is triggered. Table 2 summarizes corresponding distribution statistics.

When using MPI/OpenMP hybrid execution, computation that involves block data follows a data parallelism model. First, the block data is uniformly distributed to all available threads. All threads then perform the same operations on their subset of the data. When data must be exchanged between neighboring blocks in-between two iterations/time steps of the simulation, these messages can be sent and received in parallel by different threads. In order to send and receive data, the threads directly call non-blocking, asynchronous send and receive functions of MPI. Packing data into buffers prior to sending and unpacking data after receiving can also be done in parallel by different threads if OpenMP is activated. The OpenMP implementation of the communication module follows a task parallelism model. First, a unique work package is created for every face, every edge, and every corner of every block. These work packages are then processed in parallel by all available threads. As a result, all major parts of the simulation are executed thread-parallel.

Since results on JUQUEEN and SuperMUC are often comparable, we only show a detailed analysis for the performance and scalability of one system: JUQUEEN. We will, however, discuss the key differences to SuperMUC at the end of each section.

3.1.1. Setup. The benchmark application consists of an LBM-based simulation of lid-driven cavity flow in 3D. We employ the D3Q19 model that results in 19 unknowns per cell. Initially, the regions where the moving lid meets the domain boundaries on the side are refined three times. As a result, the benchmark contains four different levels of refinement, with level 0 corresponding to the coarsest and level 3 corresponding to the finest level. At some predefined point in time, AMR is artificially triggered by marking all blocks on the finest level for coarsening and simultaneously generating an equal number of finest cells by marking coarser neighbor blocks for refinement. Some more blocks are automatically marked for refinement in order to preserve 2:1 balance. Consequently, the region of finest resolution moves slightly inwards. Ultimately, 72 % of all cells change their size during this AMR process. The intent is to put an unusually high amount of stress on the dynamic repartitioning procedure. The domain partitioning before and after AMR is triggered is illustrated in Figure 8. During this repartitioning, the average number of blocks assigned to each process increases by 33 %. As a result, the amount of data ($\hat{=}$ number of cells) also increases by the same factor, since every block, independent of its level, stores a grid of the same size.

When increasing the number of processes by a factor of 2, the size of the domain is also extended by a factor of 2, doubling the number of blocks on each level. As a result, the distribution characteristics (see Table 2) remain constant, independent of the number of processes in use. For a fixed number of grid cells stored at each block, increasing the number of processes then corresponds to a weak scaling scenario of the simulation where the global number of blocks and cells increases linearly with the

TABLE 2

Statistics about the distribution of workload and memory among the four available levels in the setup illustrated in Figure 8. Even though the finest cells only cover a small portion of the domain, cells on the finest level account for most of the generated workload and memory consumption (\rightarrow most blocks are located on the finest level). This is true before and after the dynamic repartitioning.

	$L = 0$	$L = 1$	$L = 2$	$L = 3$	
domain coverage ratio	77.78 %	16.67 %	4.17 %	1.39 %	initially
workload share	1.10 %	3.76 %	15.02 %	80.13 %	
memory/block share	6.54 %	11.22 %	22.43 %	59.81 %	
domain coverage ratio	66.67 %	22.23 %	9.72 %	1.39 %	after AMR
workload share	0.78 %	4.13 %	28.94 %	66.15 %	
memory/block share	4.23 %	11.27 %	39.44 %	45.07 %	

TABLE 3

Average and maximal number of blocks assigned to each process for the benchmark application illustrated in Figure 8. These numbers are independent of the total number of processes. If more processes are used, the number of blocks on each level is increased accordingly. As a consequence, the average as well as the maximal number of blocks per process remain identical for any total number of processes.

level	avg. blocks/proc. (max. blocks/proc.)		
	load balancing		
	initially	before	after
0	0.383 (1)	0.328 (1)	0.328 (1)
1	0.656 (1)	0.875 (9)	0.875 (1)
2	1.313 (2)	3.063 (11)	3.063 (4)
3	3.500 (4)	3.500 (16)	3.500 (4)

number of processes. In order to also evaluate the performance of the AMR pipeline for varying amounts of data per process, the benchmarks presented in the following subsections are executed multiple times with different numbers of cells stored at each block. The stated average number of cells per core values always correspond to the state of the simulation after AMR was executed.

The average and maximal number of blocks assigned to each process are listed in Table 3. Before load balancing is executed during the AMR procedure, the distribution of blocks (more precisely: proxy blocks) is highly irregular with some processes containing far more blocks on certain levels than the average number of blocks per process would suggest. Only after load balancing, a perfect distribution is achieved with no single process containing more blocks than expected.

3.1.2. Space filling curves. First, we evaluate the performance of the entire AMR pipeline when using our SFC-based load balancing scheme during the dynamic balancing stage. Since LBM-based simulations require per-level balancing, a global synchronization of all block IDs using an MPI `allgather` operation is necessary (cf. Section 2.4.1). Moreover, we also make use of hybrid parallel execution. On JUQUEEN, we use eight threads per MPI process in order to reduce the total number of processes. Fewer processes result in a smaller number of globally available blocks, since the more threads are assigned to one process, the more cells are stored at each block (cf. second paragraph of Section 3.1). Using fewer processes for the same number of allocated cores also results in more memory available for one process. Dealing with fewer processes, fewer blocks, and more memory per process is crucial

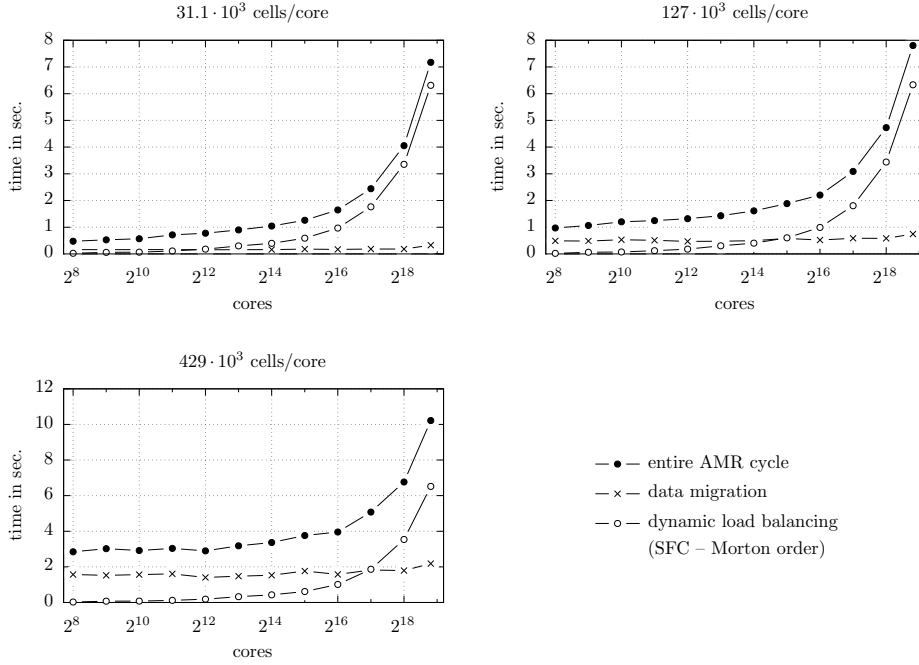


FIG. 9. Detailed runtime for the entire AMR cycle on JUQUEEN when using SFC-based dynamic load balancing for three different benchmarks that only vary in the amount of data.

for the SFC-based balancing performance (see Section 3.1.4). Moreover, for hybrid parallel execution with eight threads per process, the implementation of the LBM still achieves close to peak performance [60]. On JUQUEEN, Morton order-based balancing is approximately twice as fast as Hilbert order-based balancing. The irregular, indirect memory access caused by an additional access order lookup required for Hilbert order-based balancing results in a notable penalty on the Blue Gene/Q architecture.

Figure 9 illustrates the performance of the AMR procedure when employing the SFC-based balancing scheme using Morton order. As expected, the runtime of the balancing algorithm is independent of the amount of data stored on each block, but it increases significantly the more processes are used. With the large number of cores on JUQUEEN, the disadvantages of a global synchronization based on an MPI `allgather` operation becomes clearly visible in the timings. This approach does not scale to extreme numbers of processes and, as a consequence, will not be feasible if the number of cores continues to increase. The time required for the migration procedure is, as expected, proportional to the amount of data stored on each block. The communication network on JUQUEEN shows homogeneous performance across almost the entire system. As a result, for the same amount of data per process, the runtime of the migration procedure is almost completely independent of the number of processes. Ultimately, in large simulations, the runtime of the entire AMR algorithm is dominated by the SFC-based dynamic balancing step.

Results on SuperMUC are similar. The $O(N \log N)$ scaling properties of the SFC-based balancing can be seen in the measurements. However, since we only use up

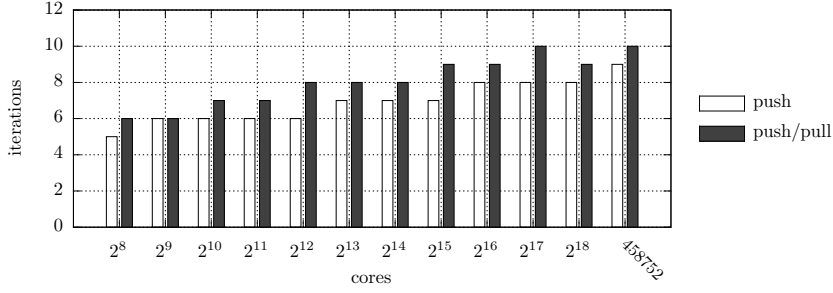


FIG. 10. Number of main iterations that are required for the diffusion procedure until perfect balance is achieved on JUQUEEN.

to 2^{16} cores on SuperMUC, the impact of the global synchronization is much less severe than with larger numbers of cores on JUQUEEN. Moreover, due to the different architecture of the processor cores, there is barely any difference between using Morton or Hilbert order for the SFC-based balancing algorithm. For 13.8 billion cells (≈ 261 billion unknowns) distributed to 65,536 cores ($\approx 210 \cdot 10^3$ cells/core), the entire AMR procedure is finished in less than one second on SuperMUC.

3.1.3. Diffusion-based load balancing. In this subsection, we evaluate the runtime of the AMR procedure when using the diffusion-based load balancing approach from Section 2.4.2. We compare two different configurations. The first configuration exclusively uses the push scheme, with 15 flow iterations during each execution of the push algorithm. The other configuration alternates between calling the push and the pull algorithm for every execution of one main iteration of the diffusion procedure. Here, each call to the push or pull algorithm only executes 5 flow iterations. For the rest of this article, we refer to these two configurations as “push” and “push/pull” configuration, respectively. Both variants always converge towards perfect balance as shown in Table 3, for every number of cores and on both systems, SuperMUC and JUQUEEN. When using the “push” configuration, executing fewer flow iterations (5, 8, 10, 12, etc.) does not always result in perfect balance, hence the 15 flow iterations that are used for this configuration.

We now use four threads per process on JUQUEEN, different from the eight threads that were employed for SFC-based balancing. For diffusion-based balancing, we do not need to restrict the total number of processes and the version utilizing four instead of eight threads per process results in the best performance for the entire simulation, including the algorithm for the LBM on nonuniform grids [60]. Figure 10 lists the number of main iterations that are required for the diffusion procedure until perfect balance is achieved³. The number of iterations slightly increases as the number of processes/utilized cores increases exponentially. The push/pull version, on average, requires one more iteration than the push only version. Moreover, executing the pull algorithm is more expensive than executing the push algorithm (see Algorithms 3 and 4). However, the push/pull version performs considerably fewer flow iterations (5 instead of 15). Ultimately, both versions result in almost identical times for the entire AMR procedure.

For a more detailed analysis of the performance results, we therefore only focus

³number of main iterations $\hat{=}$ number of times Algorithm 2 is executed during the dynamic load balancing stage

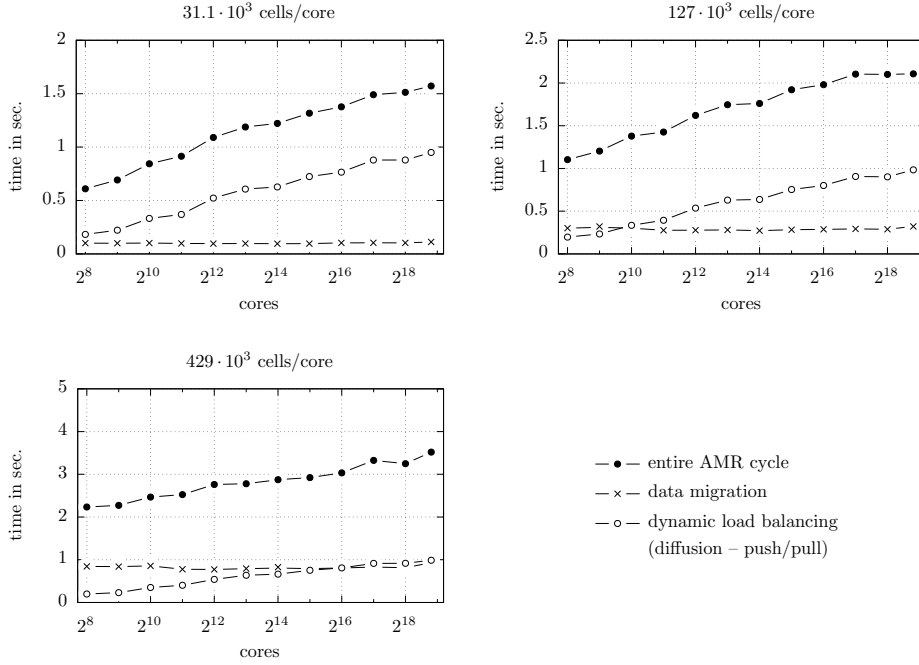


FIG. 11. Detailed runtime for the entire AMR cycle on JUQUEEN when using diffusion-based dynamic load balancing for three different benchmarks that only vary in the amount of data.

on the push/pull version. These detailed results are presented in Figure 11. The time required for the balancing algorithm is, again, independent of the amount of simulation data stored on each actual block. Contrary to SFC-based balancing, however, the time required for the diffusion-based balancing increases much slower and mainly depends on the number of main iterations required for the diffusion approach. If the number of main iterations is identical (as is the case for 2^{15} and 2^{16} cores, for example), the time required for the dynamic load balancing stage also remains almost identical. Furthermore, since diffusion-based balancing requires only communication between neighboring processes, the time required for the data migration stage proves to be virtually independent of the total number of processes and it remains nearly unchanged from 256 up to 458,752 cores. Consequently, for massively parallel simulations, the diffusion-based dynamic load balancing approach promises superior scaling characteristics as compared to an SFC-based, global load balancing scheme. When using the diffusion-based approach, executing one entire AMR cycle for a simulation that consists of 197 billion cells (≈ 3.7 trillion unknowns) and runs on all 458,752 cores of JUQUEEN only takes 3.5 seconds to complete, as opposed to the 10 seconds when using the SFC-based balancing scheme.

Results on SuperMUC are again similar. Just as on JUQUEEN, the number of main iterations required to achieve perfect balance slightly increases while the number of processes increases exponentially. Also, the runtime of the entire AMR procedure shows hardly any difference between the push/pull and the push only version. Ultimately, the time required for the dynamic load balancing stage always remains below 90 ms. For 13.8 billion cells (≈ 261 billion unknowns) on 65,536 cores ($\approx 210 \cdot 10^3$

cells/core) of SuperMUC, the entire AMR procedure completes in half a second. This is almost twice as fast when compared to the SFC-based balancing scheme.

3.1.4. Comparison between different strategies. Finally, Figure 12 provide a direct comparison between different parallelization and dynamic load balancing strategies on JUQUEEN. The advantages of the diffusion-based balancing approach are obvious. AMR that relies on the SFC-based balancing scheme presented in Section 2.4.1 suffers from the scheme’s $O(N \log N)$ scaling properties, whereas, in direct comparison, AMR that relies on diffusion-based balancing shows nearly constant run-time. For the SFC-based balancing and MPI only parallelization, some results are missing in the graphs of Figure 12. The corresponding simulations cannot complete the AMR procedure successfully since they run out of memory during the global `allgather` synchronization of the balancing stage. In order to successfully use the SFC-based balancing algorithm on all 458,752 cores of JUQUEEN, we must use an OpenMP/MPI hybrid parallelization scheme in order to reduce the number of processes. As a consequence, more memory is available for each individual process and the `allgather` operation can be executed successfully. Ultimately, the diagrams in Figure 12 show the superior performance and scaling characteristics of a fully distributed AMR pipeline that relies on our diffusion-based instead of our SFC-based dynamic load balancing algorithm. On SuperMUC, we observed the same behavior. Since SuperMUC consists of considerably fewer cores with more memory per core, the differences between the two load balancing strategies are, however, not as large. Still, the more cores are used for a single simulation, the greater the benefits of the distributed AMR pipeline that relies on diffusion-based balancing.

However, since the diffusion-based balancing approach represents an iterative, local balancing scheme, perfect balance cannot be guaranteed, as opposed to a global, SFC-based balancing scheme. Consequently, although the chosen benchmark puts a lot of pressure on the repartitioning procedure by triggering refinement/coarsening for more than two thirds of the global mesh, other scenarios might not result in perfect balance for the two configurations used in Section 3.1.3. The advantage of the iterative, fully distributed balancing approach is, however, that it makes the memory requirement of the entire AMR pipeline completely independent of the total number of processes. The memory required by a process then only depends on the amount of simulation data that is assigned to this process, but it is independent of the global amount of data and the total number of processes. In consequence, as our results indicate, this approach will scale to millions of processes and beyond and is thus well prepared for the upcoming generation of exascale supercomputers. Furthermore, our results indicate that even if perfect balance is not achieved, the peak workloads are greatly reduced already after the first few main iterations of the diffusion algorithm.

For the LBM-based simulations that we study in our work, the sizes of the blocks are always chosen such that each process ends up with only few blocks per level. Typically, each process will contain no more than around four blocks of each level. As a result, most of the time required for one iteration of the LBM is spent executing the compute kernel that updates the 19 (for the D3Q19 model) or 27 (D3Q27) values stored in each cell of the grid contained within each block. Less time is spent for communication and the synchronization of data between neighboring blocks. These kinds of simulations that only contain very few blocks per process (with hundreds to thousands of cells per block) do not face the same partitioning quality challenges that unstructured, cell-based codes are facing where for each process thousands of individual cells must be kept as compact agglomerations with low surface to volume

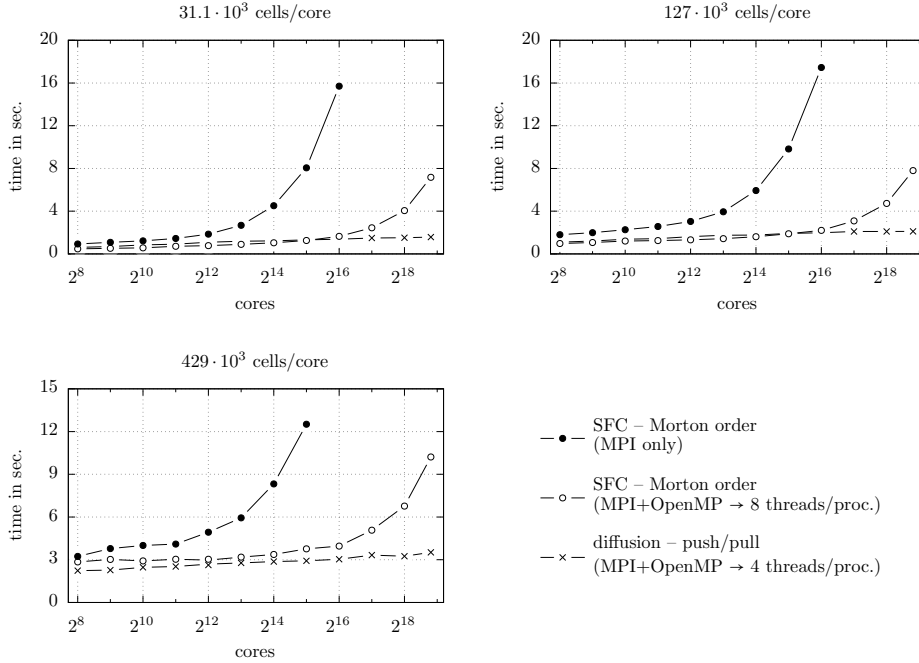


FIG. 12. Runtime of one entire AMR cycle, including dynamic load balancing and data migration, on JUQUEEN. Results are compared for different parallelization and balancing strategies.

ratios. For the LBM-based simulations with few blocks per process, the partitioning quality is mainly determined by the balance/imbalance of the number of blocks per process. As shown in the previous benchmark, few iterations (≤ 10) of the iterative scheme have been enough to eliminate all imbalances. Future work that builds on the AMR pipeline presented in this article will further study and analyze the current partitioning quality and its influence on different simulations. Furthermore, as noted in [Section 2.4](#), future work will include the integration of and comparison with other specialized dynamic load balancing libraries.

3.2. Example application. In order to demonstrate the capability of the presented algorithms, we finally turn to an application-oriented example. [Figure 13](#) illustrates a phantom geometry of the vocal fold as it is used to study the voice generation within the human throat [\[4\]](#). For this direct numerical simulation with a Reynolds number of 2,500, we use the LBM with the D3Q27 lattice and the two-relaxation-time (TRT) collision model. The simulation runs on 3,584 cores of SuperMUC and starts with a completely uniform partitioning of the entire domain into a total of 23.8 million fluid cells. AMR with a refinement criterion based on velocity gradients causes the simulation to end up with 308 million fluid cells distributed to 5 different levels. The time spent executing the AMR algorithm (see [Algorithm 1](#)) accounts for 17% of the total runtime. 95% of that time is spent on the first AMR pipeline stage and evaluating the refinement criterion, i.e., deciding whether blocks require refinement. Consequently, only 5% of the time spent for AMR ($\approx 1\%$ of the total runtime) is consumed by dynamic load balancing and data migration. During the final phase of the simulation depicted in [Figure 13](#), 311 times less memory is required and 701 times

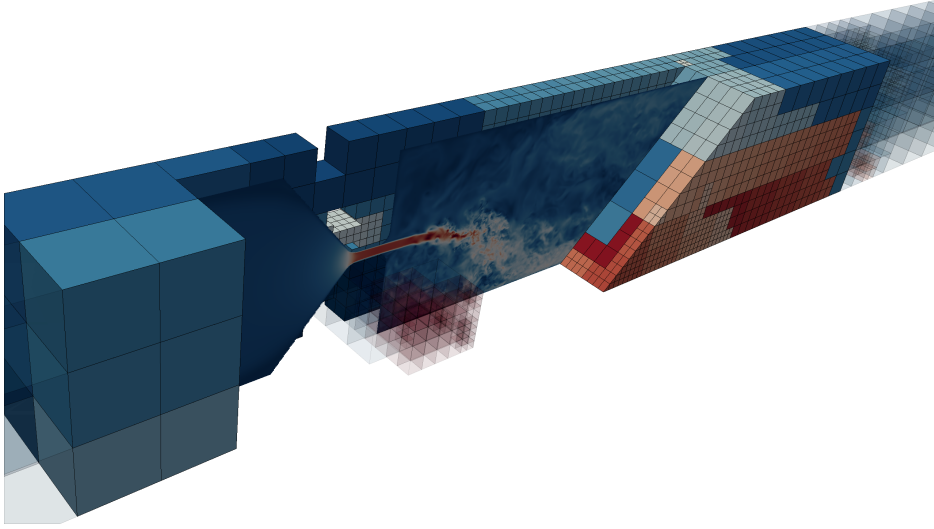


FIG. 13. 3D simulation using a phantom geometry of the human vocal fold. The figure corresponds to the final time step 180,000 ($\hat{=}$ time step 2,880,000 on the finest level) and only shows the domain partitioning into blocks. Each block consists of $34 \times 34 \times 34$ cells. The different colors of the blocks depict their process association.

less workload is generated as compared to the same simulation with the entire domain refined to the finest level.

4. Conclusion. In this article, we have presented an approach for SAMR that exploits the hierarchical nature of a block-structured domain partitioning by using a lightweight, temporary copy of the core data structure during the AMR process. The temporarily created data structure does not contain any of the simulation data and only acts as a proxy for the actual data structure. This proxy data structure enables inexpensive, iterative, diffusion-based dynamic load balancing schemes that do not require to communicate the actual simulation data during the entire load balancing phase. All data structures are stored in a perfectly distributed manner, i.e., meta data memory consumption is completely independent of the total number of processes. Ultimately, the SAMR approach presented in this article allows simulations that make use of dynamic AMR to efficiently scale to extreme-scale parallel machines.

We have demonstrated that an entire AMR cycle can be executed in half a second for a mesh that consists of 13.8 billion cells when using 65,536 processor cores. We have also confirmed the applicability of our AMR approach for meshes with up to 197 billion cells, distributed to almost half a million cores. As such, the approach demonstrates state-of-the-art scalability. To the best knowledge of the authors, the scale as well as the performance of the benchmarks presented in this article significantly exceed the data previously published for LBM-based simulations capable of AMR [29, 35, 40, 44, 47, 59, 65].

For future work, we will use the underlying distributed data structures combined with the presented AMR algorithm for meshfree simulation methods that work fundamentally different to the LBM. Future work will also look into further improving the current implementation of the load balancing schemes and see the integration of additional dynamic load balancing algorithms based on specialized libraries like ParMETIS [52, 57], Zoltan [6, 66], or PT-Scotch [19, 54].

Reproducibility. All concepts and algorithms presented in this article are implemented in the WALBERLA software framework. The LBM-based benchmark application presented in Section 3.1 was also added to the framework and is available as part of the software. The WALBERLA software framework is available under an open source license and can be freely downloaded at <http://walberla.net>.

Acknowledgments. The authors would like to thank Prof. Henning Meyerhenke, Christian Godenschwager, and Martin Bauer for valuable discussions. We are also grateful to the Jülich Supercomputing Center and the Leibniz Rechenzentrum in Munich for providing access to the supercomputers JUQUEEN and SuperMUC.

REFERENCES

- [1] M. ADAMS, P. COLELLA, D. T. GRAVES, J. N. JOHNSON, H. S. JOHANSEN, N. D. KEEN, T. J. LIGOCKI, D. F. MARTIN, P. W. MCCORQUODALE, D. MODIANO, P. O. SCHWARTZ, T. D. STERNBERG, AND B. V. STRAALEN, *Chombo Software Package for AMR Applications – Design Document*, tech. report, Lawrence Berkeley National Laboratory, 2015.
- [2] C. K. AIDUN AND J. R. CLAUSEN, *Lattice-Boltzmann method for complex flows*, Annual Review of Fluid Mechanics, 42 (2010), pp. 439–472, <https://doi.org/10.1146/annurev-fluid-121108-145519>.
- [3] D. BARTUSCHAT AND U. RÜDE, *Parallel Multiphysics Simulations of Charged Particles in Microfluidic Flows*, Journal of Computational Science, 8 (2015), pp. 1–19, <https://doi.org/10.1016/j.jocs.2015.02.006>.
- [4] S. BECKER, S. KNIESBURGES, S. MÜLLER, A. DELGADO, G. LINK, M. KALTENBACHER, AND M. DÖLLINGER, *Flow-structure-acoustic interaction in a human voice model*, The Journal of the Acoustical Society of America, 125 (2009), pp. 1351–1361, <https://doi.org/10.1121/1.3068444>.
- [5] J. E. BOILLAT, *Load balancing and Poisson equation in a graph*, Concurrency: Practice and Experience, 2 (1990), pp. 289–313, <https://doi.org/10.1002/cpe.4330020403>.
- [6] E. G. BOMAN, U. V. CATALYUREK, C. CHEVALIER, AND K. D. DEVINE, *The Zoltan and Isoropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring*, Scientific Programming, 20 (2012), pp. 129–150, <https://doi.org/10.3233/SPR-2012-0342>.
- [7] BOOST C++ LIBRARIES, <http://www.boost.org> (accessed 2018-04-15).
- [8] J. BORDNER AND M. L. NORMAN, *Enzo-P / Cello: Scalable Adaptive Mesh Refinement for Astrophysics and Cosmology*, in Proceedings of the Extreme Scaling Workshop, BW-XSEDE ’12, University of Illinois at Urbana-Champaign, 2012, pp. 4:1–4:11.
- [9] BOXLIB, <https://ccse.lbl.gov/BoxLib/> (accessed 2017-08-31).
- [10] G. L. BRYAN, M. L. NORMAN, B. W. O’SHEA, T. ABEL, J. H. WISE, M. J. TURK, D. R. REYNOLDS, D. C. COLLINS, P. WANG, S. W. SKILLMAN, B. SMITH, R. P. HARKNESS, J. BORDNER, J. KIM, M. KUHLIN, H. XU, N. GOLDBAUM, C. HUMMELS, A. G. KRITSUK, E. TASKER, S. SKORY, C. M. SIMPSON, O. HAHN, J. S. OISHI, G. C. SO, F. ZHAO, R. CEN, AND Y. LI, *ENZO: An Adaptive Mesh Refinement Code for Astrophysics*, The Astrophysical Journal Supplement Series, 211 (2014), p. 19, <https://doi.org/10.1088/0067-0049/211/2/19>.
- [11] H.-J. BUNGARTZ, M. MEHL, T. NECKEL, AND T. WEINZIERL, *The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids*, Computational Mechanics, 46 (2010), pp. 103–114, <https://doi.org/10.1007/s00466-009-0436-x>.
- [12] C. BURSTEDDE, D. CALHOUN, K. MANDLI, AND A. R. TERREL, *ForestClaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws*, Advances in Parallel Computing, 25 (2014), pp. 253–262, <https://doi.org/10.3233/978-1-61499-381-0-253>.
- [13] C. BURSTEDDE, L. C. WILCOX, AND O. GHATTAS, *p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees*, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103–1133, <https://doi.org/10.1137/100791634>.
- [14] CACTUS, <http://cactuscode.org/> (accessed 2018-04-15).
- [15] P. M. CAMPBELL, K. D. DEVINE, J. E. FLAHERTY, L. G. GERVASIO, AND J. D. TERESCO, *Dynamic Octree Load Balancing Using Space-Filling Curves*, Tech. Report CS-03-01, Williams College Department of Computer Science, 2003.
- [16] CARPET, <https://carpetcode.org/> (accessed 2018-04-15).

- [17] H. CHEN, O. FILIPPOVA, J. HOCH, K. MOLVIG, R. SHOCK, C. TEIXEIRA, AND R. ZHANG, *Grid refinement in lattice Boltzmann methods based on volumetric formulation*, Physica A: Statistical Mechanics and its Applications, 362 (2006), pp. 158–167, <https://doi.org/10.1016/j.physa.2005.09.036>.
- [18] S. CHEN AND G. D. DOOLEN, *Lattice Boltzmann method for fluid flows*, Annual Review of Fluid Mechanics, 30 (1998), pp. 329–364, <https://doi.org/10.1146/annurev.fluid.30.1.329>.
- [19] C. CHEVALIER AND F. PELLEGRINI, *PT-Scotch: A tool for efficient parallel graph ordering*, Parallel Computing, 34 (2008), pp. 318–331, <https://doi.org/10.1016/j.parco.2007.12.001>.
- [20] G. CYBENKO, *Dynamic load balancing for distributed memory multiprocessors*, Journal of Parallel and Distributed Computing, 7 (1989), pp. 279–301, [https://doi.org/10.1016/0743-7315\(89\)90021-X](https://doi.org/10.1016/0743-7315(89)90021-X).
- [21] J.-C. DESPLAT, I. PAGONABARRAGA, AND P. BLADON, *LUDWIG: A parallel Lattice-Boltzmann code for complex fluids*, Computer Physics Communications, 134 (2001), pp. 273–290, [https://doi.org/10.1016/S0010-4655\(00\)00205-8](https://doi.org/10.1016/S0010-4655(00)00205-8).
- [22] A. DUBEY, A. ALMGREN, J. BELL, M. BERZINS, S. BRANDT, G. BRYAN, P. COLELLA, D. GRAVES, M. LIJEWSKI, F. LÖFFLER, B. O’SHEA, E. SCHNETTER, B. V. STRAALEN, AND K. WEIDE, *A survey of high level frameworks in block-structured adaptive mesh refinement packages*, Journal of Parallel and Distributed Computing, 74 (2014), pp. 3217–3227, <https://doi.org/10.1016/j.jpdc.2014.07.001>.
- [23] A. DUBEY, K. ANTYPAS, M. K. GANAPATHY, L. B. REID, K. RILEY, D. SHEELER, A. SIEGEL, AND K. WEIDE, *Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code*, Parallel Computing, 35 (2009), pp. 512–522, <https://doi.org/10.1016/j.parco.2009.08.001>, <https://arxiv.org/abs/0903.4875>.
- [24] ENZO-P/CELLO, <http://client64-249.sdsc.edu/cello/> (accessed 2018-04-15).
- [25] A. FAKHARI, M. GEIER, AND T. LEE, *A mass-conserving lattice Boltzmann method with dynamic grid refinement for immiscible two-phase flows*, Journal of Computational Physics, 315 (2016), pp. 434–457, <https://doi.org/10.1016/j.jcp.2016.03.058>.
- [26] A. FAKHARI AND T. LEE, *Finite-difference lattice Boltzmann method with a block-structured adaptive-mesh-refinement technique*, Physical Review E, 89 (2014), p. 033310, <https://doi.org/10.1103/PhysRevE.89.033310>.
- [27] J. FIETZ, M. J. KRAUSE, C. SCHULZ, P. SANDERS, AND V. HEUVELINE, *Euro-Par 2012 Parallel Processing*, Springer Berlin Heidelberg, 2012, ch. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries, pp. 818–829, https://doi.org/10.1007/978-3-642-32820-6_81.
- [28] FLASH, <http://flash.uchicago.edu/site/flashcode/> (accessed 2018-04-15).
- [29] S. FREUDIGER, J. HEGEWALD, AND M. KRAFCZYK, *A parallelisation concept for a multi-physics lattice Boltzmann prototype based on hierarchical grids*, Progress in Computational Fluid Dynamics, 8 (2008), pp. 168–178, <https://doi.org/10.1504/PCFD.2008.018087>.
- [30] C. GODENSWAGER, F. SCHORNBAUM, M. BAUER, H. KÖSTLER, AND U. RÜDE, *A Framework for Hybrid Parallel Flow Simulations with a Trillion Cells in Complex Geometries*, in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’13, ACM, 2013, pp. 35:1–35:12, <https://doi.org/10.1145/2503210.2503273>.
- [31] T. GOODALE, G. ALLEN, G. LANFERMANN, J. MASSÓ, T. RADKE, E. SEIDEL, AND J. SHALF, *The Cactus Framework and Toolkit: Design and Applications*, in Vector and Parallel Processing – VECPAR’2002, 5th International Conference, Lecture Notes in Computer Science, Springer, 2003, <http://edoc.mpg.de/3341>.
- [32] J. GÖTZ, K. IGLBERGER, M. STÜRMER, AND U. RÜDE, *Direct numerical simulation of particulate flows on 294912 processor cores*, in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–11, <https://doi.org/10.1109/SC.2010.20>.
- [33] D. GROEN, O. HENRICH, F. JANOSCHEK, P. COVENEY, AND J. HARTING, *Lattice-Boltzmann Methods in Fluid Dynamics: Turbulence and Complex Colloidal Fluids*, in Juelich Blue Gene/P Extreme Scaling Workshop 2011, Juelich Supercomputing Centre, 2011.
- [34] D. GROEN, J. HETHERINGTON, H. B. CARVER, R. W. NASH, M. O. BERNABEU, AND P. V. COVENEY, *Analysing and modelling the performance of the HemeLB lattice-Boltzmann simulation environment*, Journal of Computational Science, 4 (2013), pp. 412–422, <https://doi.org/10.1016/j.jocs.2013.03.002>.
- [35] M. HASERT, K. MASILAMANI, S. ZIMNY, H. KLIMACH, J. QI, J. BERNSDORF, AND S. ROLLER, *Complex fluid simulations with the parallel tree-based Lattice Boltzmann solver Musubi*, Journal of Computational Science, 5 (2014), pp. 784–794, <https://doi.org/10.1016/j.jocs.2013.11.001>.

- [36] V. HEUVELINE AND J. LATT, *The OpenLB Project: An Open Source and Object Oriented Implementation of Lattice Boltzmann Methods*, International Journal of Modern Physics C, 18 (2007), pp. 627–634, <https://doi.org/10.1142/S0129183107010875>.
- [37] D. HILBERT, *Ueber die stetige Abbildung einer Linie auf ein Flächenstück*, Mathematische Annalen, 38 (1891), pp. 459–460.
- [38] L. V. KALE AND G. ZHENG, *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, John Wiley & Sons, Inc., 2009, ch. Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects, pp. 265–282, <https://doi.org/10.1002/9780470558027.ch13>.
- [39] D. LAGRAVA, O. MALASPINAS, J. LATT, AND B. CHOPARD, *Advances in multi-domain lattice Boltzmann grid refinement*, Journal of Computational Physics, 231 (2012), pp. 4808–4822, <https://doi.org/10.1016/j.jcp.2012.03.015>.
- [40] M. LAHNERT, C. BURSTEDDE, C. HOLM, M. MEHL, G. REMPFER, AND F. WEIK, *Towards Lattice-Boltzmann on Dynamically Adaptive Grids – Minimally-Invasive Grid Exchange in ESPResSo*, in Proceedings of the ECCOMAS Congress 2016, VII European Congress on Computational Methods in Applied Sciences and Engineering, 2016, pp. 1–25.
- [41] LB3D, <http://ccs.chem.ucl.ac.uk/lb3d> (accessed 2018-04-15).
- [42] P. MACNEICE, K. M. OLSON, C. MOBARRY, R. DE FAINCHEIN, AND C. PACKER, *PARAMESH: A parallel adaptive mesh refinement community toolkit*, Computer Physics Communications, 126 (2000), pp. 330–354, [https://doi.org/10.1016/S0010-4655\(99\)00501-9](https://doi.org/10.1016/S0010-4655(99)00501-9).
- [43] R. C. MARTIN, *The Open-Closed Principle*, C++ Report, (1996).
- [44] M. MEHL, T. NECKEL, AND P. NEUMANN, *Navier-Stokes and Lattice-Boltzmann on octree-like grids in the Peano framework*, International Journal for Numerical Methods in Fluids, 65 (2011), pp. 67–86, <https://doi.org/10.1002/flid.2469>.
- [45] B. MEYER, *Object-Oriented Software Construction*, Prentice Hall, 1988.
- [46] G. M. MORTON, *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, tech. report, IBM Ltd., 1966.
- [47] P. NEUMANN AND T. NECKEL, *A dynamic mesh refinement technique for Lattice Boltzmann simulations on octree-like grids*, Computational Mechanics, 51 (2013), pp. 237–253, <https://doi.org/10.1007/s00466-012-0721-y>.
- [48] OPENLB, <http://optilb.org/openlb/> (accessed 2018-04-15).
- [49] PALABOS, <http://www.palabos.org/> (accessed 2018-04-15).
- [50] PARAMESH, <https://opensource.gsfc.nasa.gov/projects/paramesh/index.php> (accessed 2018-04-15).
- [51] S. G. PARKER, *A component-based architecture for parallel multi-physics PDE simulation*, Future Generation Computer Systems, 22 (2006), pp. 204–216, <https://doi.org/10.1016/j.future.2005.04.001>.
- [52] PARMETIS, <http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview> (accessed 2018-04-15).
- [53] T. PRECLIK AND U. RÜDE, *Ultrascale simulations of non-smooth granular dynamics*, Computational Particle Mechanics, 2 (2015), pp. 173–196, <https://doi.org/10.1007/s40571-015-0047-6>.
- [54] PT-SCOTCH, <http://www.labri.fr/perso/pelegrin/scotch/> (accessed 2018-04-15).
- [55] A. RANGLES, *Modeling Cardiovascular Hemodynamics Using the Lattice Boltzmann Method on Massively Parallel Supercomputers*, PhD thesis, Harvard University, 2013.
- [56] M. ROHDE, D. KANDHAI, J. J. DERKSEN, AND H. E. A. VAN DEN AKKER, *A generic, mass conservative local grid refinement technique for lattice-Boltzmann schemes*, International Journal for Numerical Methods in Fluids, 51 (2006), pp. 439–468, <https://doi.org/10.1002/flid.1140>.
- [57] K. SCHLOEGEL, G. KARYPIS, AND V. KUMAR, *Parallel static and dynamic multi-constraint graph partitioning*, Concurrency and Computation: Practice and Experience, 14 (2002), pp. 219–240, <https://doi.org/10.1002/cpe.605>.
- [58] E. SCHNETTER, S. H. HAWLEY, AND I. HAWKE, *Evolutions in 3D numerical relativity using fixed mesh refinement*, Classical and Quantum Gravity, 21 (2004), pp. 1465–1488, <https://doi.org/10.1088/0264-9381/21/6/014>, <https://arxiv.org/abs/gr-qc/0310042>.
- [59] M. SCHÖNHERR, K. KUCHER, M. GEIER, M. STIEBLER, S. FREUDIGER, AND M. KRAFCZYK, *Multi-thread implementations of the lattice Boltzmann method on non-uniform grids for CPUs and GPUs*, Computers & Mathematics with Applications, 61 (2011), pp. 3730–3743, <https://doi.org/10.1016/j.camwa.2011.04.012>.
- [60] F. SCHORNBaum AND U. RÜDE, *Massively Parallel Algorithms for the Lattice Boltzmann Method on Nonuniform Grids*, SIAM Journal on Scientific Computing, 38 (2016), pp. C96–C126, <https://doi.org/10.1137/15M1035240>, <https://arxiv.org/abs/1508.07982>.

- [61] THE ENZO PROJECT, <http://enzo-project.org/> (accessed 2018-04-15).
- [62] J. TÖLKE, S. FREUDIGER, AND M. KRAFCZYK, *An adaptive scheme using hierarchical grids for lattice Boltzmann multi-phase flow simulations*, Computers & Fluids, 35 (2006), pp. 820–830, <https://doi.org/10.1016/j.compfluid.2005.08.010>.
- [63] UINTAH, <http://uintah.utah.edu/> (accessed 2018-04-15).
- [64] WALBERLA, <http://walberla.net> (accessed 2018-04-15).
- [65] Z. YU AND L.-S. FAN, *An interaction potential based lattice Boltzmann method with adaptive mesh refinement (AMR) for two-phase flow simulation*, Journal of Computational Physics, 228 (2009), pp. 6456–6478, <https://doi.org/10.1016/j.jcp.2009.05.034>.
- [66] ZOLTAN, <http://www.cs.sandia.gov/Zoltan> (accessed 2018-04-15).

Appendix A. Additional timing details for benchmarks on JUQUEEN in Section 3.1.2. Table A1 lists the exact runtime for one entire AMR cycle on JUQUEEN when using SFC-based dynamic load balancing. These timings show that on JUQUEEN Morton order-based balancing is approximately twice as fast as Hilbert order-based balancing. Table A2 lists a breakdown of all times that were used to generate the graphs in Figure 9.

TABLE A1

Time for one entire AMR cycle, including dynamic load balancing and data migration on JUQUEEN (in seconds). The table compares SFC-based load balancing using Hilbert order with SFC-based load balancing using Morton order. Times are listed for three benchmarks that only vary in the amount of data assigned to each block (and therefore each core).

cores	31.1 · 10 ³ cells/core		127 · 10 ³ cells/core		429 · 10 ³ cells/core	
	Hilbert	Morton	Hilbert	Morton	Hilbert	Morton
256	0.48	0.47	1.02	0.97	2.83	2.85
512	0.57	0.53	1.18	1.06	2.64	3.02
1 024	0.60	0.57	1.23	1.20	2.93	2.92
2 048	0.75	0.71	1.39	1.25	3.12	3.04
4 096	0.88	0.78	1.52	1.32	3.14	2.90
8 192	1.16	0.90	1.68	1.43	3.25	3.18
16 384	1.51	1.04	2.23	1.61	3.85	3.37
32 768	2.12	1.26	2.78	1.88	4.73	3.76
65 536	3.35	1.65	3.93	2.20	5.82	3.97
131 072	5.82	2.44	6.50	3.09	8.63	5.08
262 144	10.63	4.05	11.41	4.73	13.57	6.77
458 752	18.54	7.17	19.22	7.80	21.62	10.22

TABLE A2

Breakdown of all times (in sec.) from the benchmark outlined in Figure 9: One complete AMR cycle with SFC-based (Morton order) load balancing on JUQUEEN.

cores	entire AMR			load balancing			data migration		
	cells/core (in 10 ³)			cells/core (in 10 ³)			cells/core (in 10 ³)		
	31.1	127	429	31.1	127	429	31.1	127	429
256	0.47	0.97	2.85	0.02	0.02	0.02	0.16	0.49	1.57
512	0.53	1.06	3.02	0.06	0.06	0.07	0.16	0.48	1.53
1 024	0.57	1.20	2.92	0.06	0.07	0.08	0.16	0.53	1.56
2 048	0.71	1.25	3.04	0.11	0.12	0.12	0.16	0.51	1.61
4 096	0.78	1.32	2.90	0.18	0.18	0.19	0.15	0.47	1.40
8 192	0.90	1.43	3.18	0.30	0.30	0.33	0.16	0.48	1.48
16 384	1.04	1.61	3.37	0.39	0.40	0.43	0.16	0.50	1.53
32 768	1.26	1.88	3.76	0.59	0.60	0.62	0.18	0.58	1.77
65 536	1.65	2.20	3.96	0.97	0.99	1.01	0.17	0.52	1.58
131 072	2.44	3.09	5.08	1.76	1.80	1.86	0.18	0.59	1.82
262 144	4.05	4.73	6.77	3.35	3.44	3.54	0.18	0.58	1.79
458 752	7.17	7.80	10.22	6.31	6.33	6.52	0.33	0.74	2.19

Appendix B. Results of benchmarks on SuperMUC of Section 3.1.2.

Table A3 lists the exact runtime for one entire AMR cycle on SuperMUC when using SFC-based dynamic load balancing. These timings show that on SuperMUC there is barely any difference between using Morton or Hilbert order for the SFC-based balancing algorithm. If a small difference in execution time can be measured, the AMR procedure that uses the Hilbert order-based balancing is a bit slower due to the additional effort that is required for accessing a lookup table (cf. Section 2.4.1). On SuperMUC, the benchmark always makes use of hybrid parallel execution with four OpenMP threads per MPI process.

Figure A1 shows detailed results for SFC-based balancing using Morton order. The time required for the load balancing stage only depends on the total number of globally available proxy blocks, but is independent of the amount of simulation data stored in each actual block. Consequently, the time required for the load balancing stage is identical in all three scenarios. As expected, the runtime of the SFC-based balancing algorithm increases with the number of processes due to the `allgather` operation and the subsequent sorting of all block IDs (cf. Section 2.4.1). Just as on JUQUEEN, if the amount of data per block increases, the time required for the migration stage increases proportionally. Furthermore, migration time also increases with the number of processes since SFC-based balancing results in a global reassignment of all blocks regardless of the blocks' previous process associations. As a consequence, some of the data must be migrated between distant processes, and this distance increases the more processes are utilized⁴. Ultimately, SFC-based dynamic load balancing shows good performance (140 ms on 65,536 cores), with the runtime of the entire AMR cycle being dominated by the migration, refinement, and coarsening of the cell data. A breakdown of all times that were used to generate the graphs in Figure 9 is also presented in Table A4. For 13.8 billion cells ($\hat{=}$ 261 billion unknowns) on 65,536 cores ($\hat{=}$ $210 \cdot 10^3$ cells/core), the entire AMR procedure is finished in less than one second.

TABLE A3

Time for one entire AMR cycle, including dynamic load balancing and data migration on SuperMUC (in seconds). The table compares SFC-based load balancing using Hilbert order with SFC-based load balancing using Morton order. Times are listed for three benchmarks that only vary in the amount of data assigned to each block (and therefore each core).

cores	$62.1 \cdot 10^3$ cells/core		$210 \cdot 10^3$ cells/core		$971 \cdot 10^3$ cells/core	
	Hilbert	Morton	Hilbert	Morton	Hilbert	Morton
512	0.15	0.15	0.40	0.40	1.77	1.75
1 024	0.17	0.17	0.43	0.43	1.86	1.86
2 048	0.18	0.18	0.45	0.45	1.96	1.95
4 096	0.20	0.20	0.48	0.47	2.01	2.02
8 192	0.22	0.20	0.50	0.48	2.06	2.04
16 384	0.28	0.26	0.63	0.61	2.56	2.50
32 768	0.33	0.29	0.75	0.71	3.01	2.93
65 536	0.57	0.42	1.04	0.93	3.50	3.38

⁴On SuperMUC, more bandwidth is available for communication within the same compute island (1 island $\hat{=}$ 8,192 cores) than for inter-island communication.

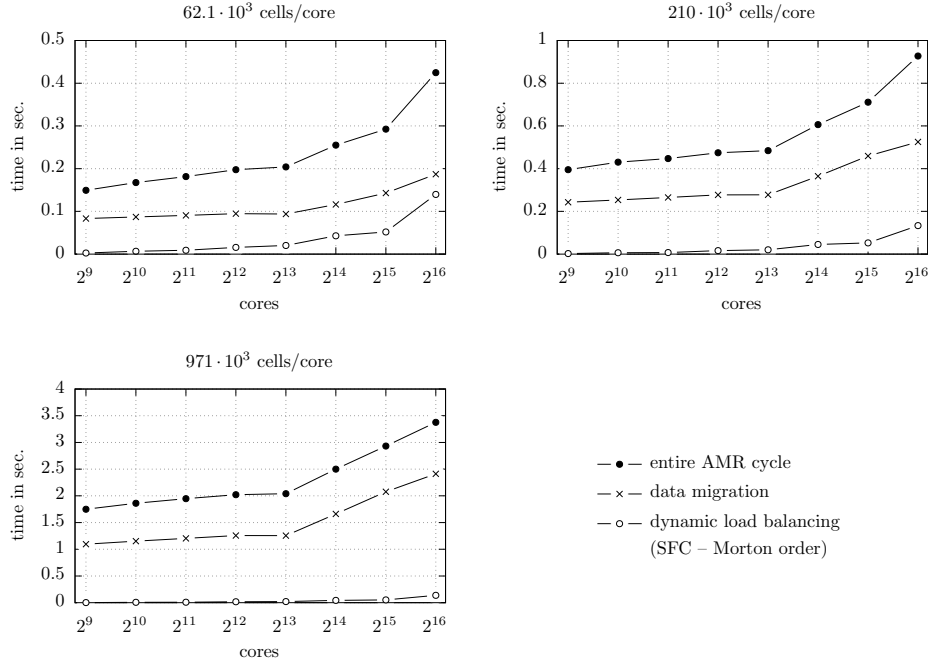


FIG. A1. Detailed runtime for the entire AMR cycle on SuperMUC when using SFC-based dynamic load balancing for three different benchmarks that only vary in the amount of data. A breakdown of all times used to generate these graphs is available in Table A4.

TABLE A4

Breakdown of all times (in sec.) from the benchmark outlined in Figure A1: One complete AMR cycle with SFC-based (Morton order) load balancing on SuperMUC.

cores	entire AMR			load balancing			data migration		
	cells/core (in 10^3)			cells/core (in 10^3)			cells/core (in 10^3)		
	62.1	210	971	62.1	210	971	62.1	210	971
512	0.15	0.40	1.75	0.003	0.003	0.003	0.08	0.24	1.09
1024	0.17	0.43	1.86	0.007	0.007	0.006	0.09	0.25	1.15
2048	0.18	0.45	1.95	0.009	0.008	0.009	0.09	0.27	1.20
4096	0.20	0.47	2.02	0.016	0.017	0.017	0.09	0.28	1.26
8192	0.20	0.48	2.04	0.020	0.020	0.020	0.09	0.28	1.25
16384	0.26	0.61	2.50	0.043	0.045	0.043	0.12	0.36	1.66
32768	0.29	0.71	2.93	0.052	0.053	0.052	0.14	0.46	2.08
65536	0.42	0.93	3.38	0.140	0.133	0.138	0.19	0.53	2.41

Appendix C. Additional timing details for benchmarks on JUQUEEN in Section 3.1.3. Table A5 lists the exact runtime for one entire AMR cycle on JUQUEEN when using diffusion-based dynamic load balancing. These timings show that the push scheme as well as the push/pull scheme both result in almost identical runtimes. Table A6 lists a breakdown of all times that were used to generate the graphs in Figure 11.

TABLE A5

Time for one entire AMR cycle, including dynamic load balancing and data migration on JUQUEEN (in seconds). The table compares two different versions for the diffusion-based load balancing. Times are, again, listed for three benchmarks that only vary in the amount of data assigned to each block.

cores	$31.1 \cdot 10^3$ cells/core		$127 \cdot 10^3$ cells/core		$429 \cdot 10^3$ cells/core	
	push	push/pull	push	push/pull	push	push/pull
256	0.60	0.61	1.05	1.10	2.17	2.23
512	0.71	0.69	1.22	1.20	2.40	2.27
1024	0.80	0.84	1.34	1.38	2.48	2.47
2048	0.91	0.91	1.44	1.43	2.52	2.52
4096	1.03	1.09	1.56	1.62	2.67	2.76
8192	1.09	1.19	1.65	1.75	2.67	2.78
16384	1.18	1.22	1.74	1.76	2.80	2.87
32768	1.23	1.32	1.85	1.92	2.87	2.92
65536	1.35	1.38	1.86	1.98	3.02	3.03
131072	1.36	1.49	1.79	2.10	3.04	3.33
262144	1.48	1.51	1.94	2.10	3.15	3.25
458752	1.55	1.57	2.06	2.11	3.26	3.52

TABLE A6

Breakdown of all times (in sec.) from the benchmark outlined in Figure 11: One complete AMR cycle with diffusion-based (push/pull scheme with 5 flow iterations) load balancing on JUQUEEN.

cores	entire AMR			load balancing			data migration		
	cells/core (in 10^3)			cells/core (in 10^3)			cells/core (in 10^3)		
	31.1	127	429	31.1	127	429	31.1	127	429
256	0.61	1.10	2.23	0.18	0.20	0.20	0.10	0.30	0.84
512	0.69	1.20	2.27	0.22	0.23	0.23	0.10	0.30	0.84
1024	0.84	1.38	2.47	0.33	0.33	0.35	0.10	0.30	0.86
2048	0.91	1.43	2.53	0.37	0.39	0.40	0.10	0.28	0.78
4096	1.09	1.62	2.76	0.52	0.54	0.54	0.10	0.28	0.77
8192	1.19	1.75	2.78	0.61	0.63	0.64	0.10	0.28	0.78
16384	1.22	1.76	2.87	0.63	0.64	0.66	0.10	0.27	0.77
32768	1.32	1.92	2.92	0.72	0.75	0.75	0.10	0.28	0.79
65536	1.38	1.98	3.03	0.77	0.80	0.81	0.10	0.29	0.81
131072	1.49	2.10	3.33	0.88	0.91	0.92	0.10	0.29	0.83
262144	1.51	2.10	3.25	0.88	0.90	0.92	0.10	0.29	0.81
458752	1.57	2.11	3.52	0.95	0.98	0.99	0.11	0.32	0.93

Appendix D. Results of benchmarks on SuperMUC of Section 3.1.3.

Figure A2 lists the number of main iterations that are required for the diffusion procedure until perfect balance is achieved on SuperMUC⁵. The number of iterations increases very little as the number of processes/utilized cores increases exponentially. Just as on JUQUEEN, the push/pull version, on average, requires one more iteration than the push only version. Ultimately, both versions result in almost identical times for the entire AMR procedure as shown in Table A7.

Detailed results when using the push/pull scheme are presented in Figure A3. Here, we again use hybrid parallelization with four threads per process. Contrary to SFC-based balancing, the time required for the diffusion-based balancing increases much slower and mainly depends on the number of main iterations required for the diffusion approach. Just as on JUQUEEN, if the number of main iterations is identical, the time required for the dynamic load balancing stage also remains almost identical. Contrary to SFC-based balancing on SuperMUC (see Figure A1), the time required for the data migration stage proves to be virtually independent of the total number of processes since diffusion-based balancing requires only communication between neighboring processes. For simulations with large amounts of data that must be communicated during the AMR procedure, the runtime of the entire AMR algorithm remains almost constant for any number of processes.

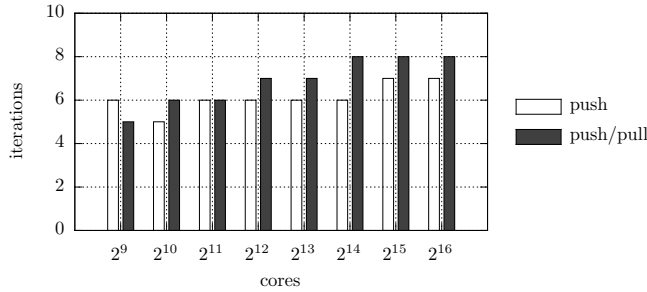


FIG. A2. Number of main iterations that are required for the diffusion procedure until perfect balance is achieved on SuperMUC.

TABLE A7

Time for one entire AMR cycle, including dynamic load balancing and data migration on SuperMUC (in seconds). The table compares two different versions for the diffusion-based load balancing. Times are listed for three benchmarks that only vary in the amount of data assigned to each block (and therefore each core).

cores	62.1 · 10 ³ cells/core		210 · 10 ³ cells/core		971 · 10 ³ cells/core	
	push	push/pull	push	push/pull	push	push/pull
512	0.14	0.15	0.36	0.36	1.53	1.58
1 024	0.16	0.17	0.38	0.39	1.60	1.62
2 048	0.18	0.18	0.41	0.40	1.63	1.65
4 096	0.20	0.20	0.42	0.43	1.65	1.69
8 192	0.21	0.21	0.43	0.44	1.65	1.69
16 384	0.23	0.23	0.46	0.46	1.69	1.80
32 768	0.24	0.26	0.48	0.49	1.75	1.86
65 536	0.27	0.26	0.53	0.53	1.80	1.90

⁵number of main iterations $\hat{=}$ number of times Algorithm 2 is executed during the dynamic load balancing stage

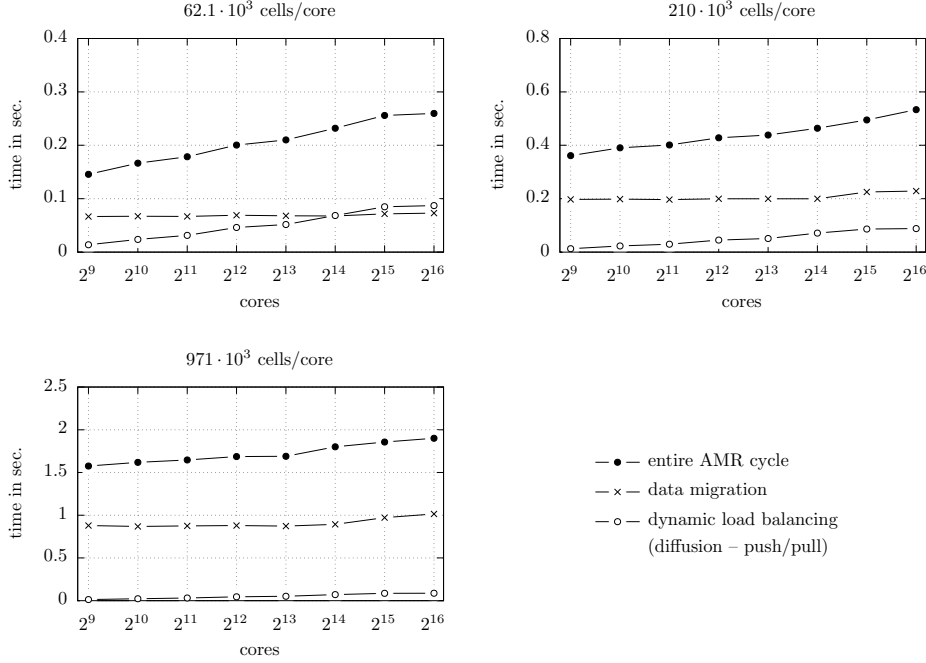


FIG. A3. Detailed runtime for the entire AMR cycle on SuperMUC when using diffusion-based dynamic load balancing for three different benchmarks that only vary in the amount of data. A breakdown of all times used to generate these graphs is available in [Table A8](#).

TABLE A8

Breakdown of all times (in sec.) from the benchmark outlined in [Figure A3](#): One complete AMR cycle with diffusion-based (push/pull scheme with 5 flow iterations) load balancing on SuperMUC.

cores	entire AMR			load balancing			data migration		
	cells/core (in 10^3)			cells/core (in 10^3)			cells/core (in 10^3)		
	62.1	210	971	62.1	210	971	62.1	210	971
512	0.15	0.36	1.58	0.014	0.013	0.013	0.07	0.20	0.88
1 024	0.17	0.39	1.62	0.024	0.023	0.023	0.07	0.20	0.87
2 048	0.18	0.40	1.65	0.031	0.030	0.031	0.07	0.20	0.88
4 096	0.20	0.43	1.69	0.046	0.045	0.045	0.07	0.20	0.88
8 192	0.21	0.44	1.69	0.052	0.051	0.052	0.07	0.20	0.87
16 384	0.23	0.46	1.80	0.069	0.071	0.072	0.07	0.20	0.89
32 768	0.26	0.49	1.86	0.085	0.086	0.086	0.07	0.23	0.97
65 536	0.26	0.53	1.90	0.087	0.088	0.087	0.07	0.23	1.01

Appendix E. Time evolution of the application in Section 3.2. Figure A4 provides an illustration of the evolution of the simulation introduced in Section 3.2 over time. The simulation runs for approx. 24 hours on 3,584 cores of SuperMUC and spans 180,000 time steps on the coarsest and 2,880,000 time steps on the finest grid.

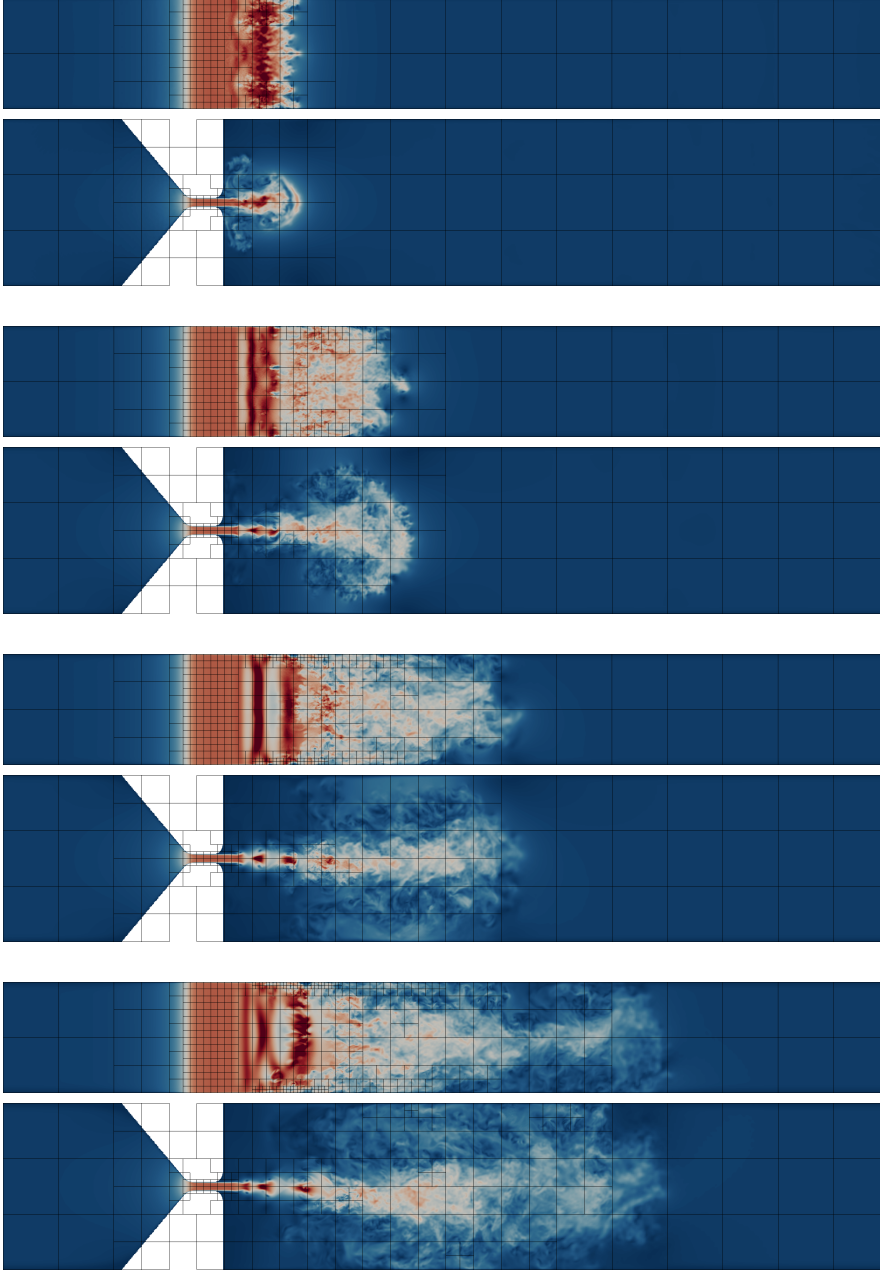


FIG. A4. Evolution of the simulation introduced in Section 3.2 over time. The figures represent 2D slices through the simulation as viewed from the top and from the side. They show the state of the simulation during the build-up phase at time step 6,000, 11,000, 20,000, and 36,000. These figures, however, do not show the entire domain. The actual domain extends slightly further to the left and a lot further to the right.

Of the 308 million fluid cells at the end of the simulation, almost 50 % are located on the second finest level. Over the course of the simulation, the total number of blocks increases from 612 (all on the coarsest level) to 8030 (distributed among all 5 levels). Even though the AMR algorithm is executed in every time step of the 180,000 coarse time steps, actual dynamic refinement/coarsening of the grid data is only triggered 537 times. Ultimately, executing the entire AMR pipeline, including dynamic load balancing and the migration of the data, only happens every 335 time steps (on average).