A fully distributed CFD framework for massively parallel systems

J. Zudrop, H. Klimach, M. Hasert, K. Masilamani, S. Roller
Applied Supercomputing for Engineering, German Research School for Simulation Sciences GmbH and RWTH Aachen University
Aachen, Germany
j.zudrop@grs-sim.de, h.klimach@grs-sim.de, m.hasert@grs-sim.de, k.masilamani@grs-sim.de, s.roller@grs-sim.de

***Abstract*—A highly scalable solver framework, based on a linearized octree is presented. It allows for fully distributed computations and avoids special needs on single processes with potential bottlenecks, while enabling simulations with complex geometries. Scaling results on the Cray XE6 system Hermit at HLRS in Stuttgart are presented with runs up to 3072 nodes with 98304 MPI processes. Even with a fully indirect addressing a high sustained performance of more than 9% can be achieved on the system, enabling very large simulations. Two flow simulation methods are shown, a Finite Volume Method for compressible flows, and a Lattice Boltzmann Method for incompressible flows in complex geometries.**

***Keywords*-Octree; Computational fluid dynamics; Finite Volume; Lattice Boltzmann; High Performance Computing; Performance; Scaling**

I. Introduction

An efficient usage of modern massively parallel hardware is non-trivial, and simulation frameworks often suffer from bottleneck problems at some point in the process chain. A lack of scalability imposes a severe limitation, as the development of modern computing architectures follows a trend to ever more distributed resources.

Scalability bottlenecks often only become visible after surpassing a larger number of processes. Methods which increase in running time or consumed memory with the number of processes in a $\mathcal{O}(p)$ manner will eventually begin to dominate the simulation on thousands of processes, while they might not even be noticed on smaller counts of processes.

In the following we present a simulation framework designed for the distributed computation of mesh-based methods. We consider two different numerical algorithms (Finite Volume and Lattice Boltzmann method) in the field of computational fluid dynamics (CFD) on top of a common infrastructure, based on an octree mesh representation. Octree mesh based simulations are not new and have been proposed for example by Flaherty et al. in 1997 [1]. However, to the authors' knowledge, our approach is for the first time fully exploiting the features of such meshes for massively distributed computing. Several implementations for octree meshes exist, but they are often designated to a single numerical method like the Dendro [2] library for FEM, or impose a cell-wise computation approach prohibiting the exploitation of vectorization like the Peano framework [3]. In our opinion, a consideration of the full simulation pipeline as described by Tu et al. [4] is essential for successful large scale simulations on distributed systems. The described framework addresses the complete toolchain from mesh generation to simulation with different solvers for different equation systems to post-processing.

The outline of this paper is as follows: In section II we describe our approach to a highly scalable solver framework called APES (Adaptable Poly-Engineering Simulator). Section II-A describes the partitioning and parallelization strategy of the mesh. Different requirements of the solvers and our approach to harmonize them are described in section II-C. Section III outlines the algorithmic approaches to computational fluid dynamics in the implemented solvers. In the final section V we give a conclusion and outline future plans for the APES simulation framework.

II. Octree framework

Many modern simulation techniques aim to solve partial differential equations on a mesh, i.e. they decompose the domain of interest into smaller discrete elements. We consider two popular approaches of mesh-based numerical algorithms: the Finite Volume (FVM) and Lattice Boltzmann method (LBM). An attractive feature of these two methods is their ability to handle arbitrarily complex geometries. However, the type of meshes that are typically used are different. Unstructured meshes are usually used with the FVM, while the LBM operates on Cartesian voxel meshes. Although an unstructured mesh nicely solves the problem of complex geometries, its arbitrary nature imposes problems on massively parallel and distributed computing systems. In fully unstructured meshes, the relation of each element to adjacent elements has to be identified explicitly and these neighbors might be arbitrarily distributed across remote computing partitions. If the elements of the unstructured mesh are not sorted and neighborhood relations are not identified globally beforehand, each partition potentially requires complete mesh information from all remote processes introducing a severe bottleneck. This can be remedied by storing neighbor information and using a global ordering of the elements [5]. However, such an approach is merely shifting the problem to the preprocessing step, which eventually fails for very large problem sizes.

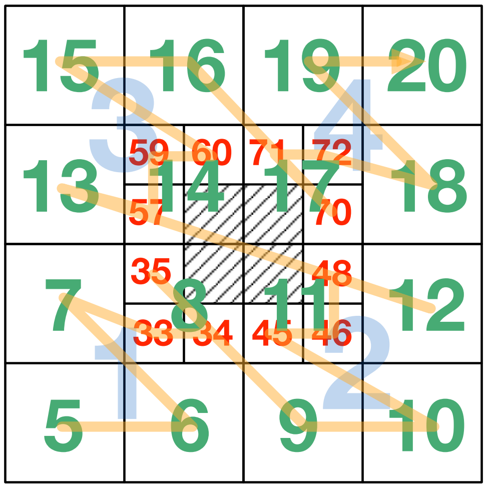

Figure 1. Schematic 2D (quad-) tree mesh example.

Therefore, we propose to solve this problem by octree-based meshes for both methods. We utilize a global, breadth-first numbering scheme for all the nodes of the full octree with a geometric ordering given by a space filling curve (SFC) [6]. The tree topology information of each element is held in the so-called treeID, an integer number, and allows each process to compute neighbors locally with minimal data about remote partitions.

Figure 1 shows a sample discretization of a cubic physical domain. For the sake of simplicity, we restrict the illustrations to a quadtree in two dimensions, where the same rules apply to the equivalent octree in three dimensions. An obstacle is located at the center of the bounding cube. This bounding cube corresponds to the root cell of the tree, and its children are refined towards the obstacle. The mesh generation is described in detail in Harlacher et al. [7].

The mesh is generated in a recursive fashion, starting from the root cell on level 0. Surface geometry is provided in the form of triangles with the help of STL files. Attached to the geometrical objects is a definition of the resolution level, and the tree is recursively refined accordingly. The result is a hierarchical mesh, i.e. each cell has one parent cell on the next coarser level and a uniform amount of eight children on the next finer level. The spatial ordering of the children on each level is obtained by the space-filling Z-Curve, indicated by the orange line in figure 1. This introduces a linearization of the hierarchical tree into a linear representation. Only the leaf nodes of the tree are part of the final mesh.

An example of a fully refined mesh for a porous medium is presented in Figure 2.

A. Parallelization and domain decomposition

With the linearization of the tree by means of the SFC, we obtain a one-dimensional list of elements. This list can be divided easily into chunks of similar size to achieve a reasonable partitioning for parallel runs. Figure 3 shows this partitioning strategy. Only leaf nodes of the tree are stored in the list. The colored boxes indicate the distribution of elements to the processes. This is important in order to achieve a balanced method for reading the mesh efficiently in parallel on a distributed system. The partitioning might be refined afterwards, to take into account different computational costs of certain elements.

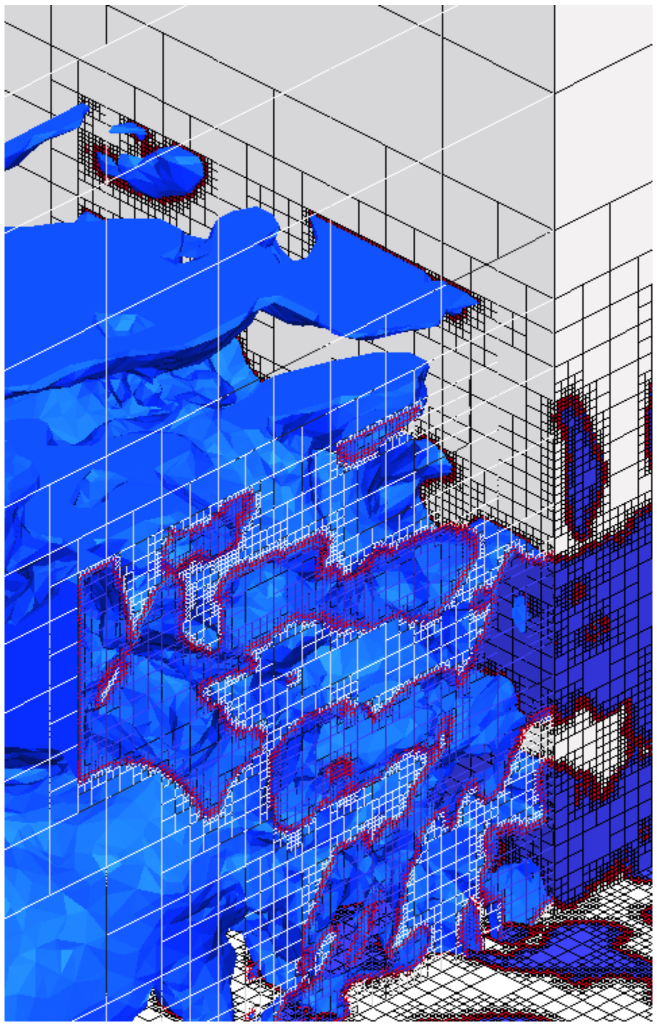

Figure 2. Detail of a porous medium approximated by an octree mesh. The grey cells build up the fluid domains, the red boundaries contain geometries, and the blue cells are not part of the computational domain.

The treeIDs are stored in 8 byte signed integers for best portability. This allows for meshes of up to $1.8 \cdot 10^{18}$ elements which corresponds to one million elements in each coordinate direction. Each element is identified by the numbering scheme as pointed out in Figure 1 and 3.

Mesh-based numerical solvers for partial differential equations typically work with a stencil to include values of the neighboring cells for computations of the current cell. With the previous definitions and the topology of the octree, we

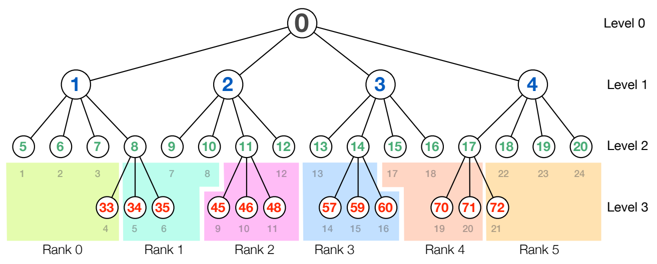

Figure 3. Schematic 2D tree representation of the mesh in Figure 1.

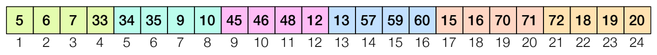

Figure 4. Linearization of the cells of the mesh of Figure 1. Offsets indicated by smaller numbers below the colored boxes. Distribution over processes is indicated by the color.

can efficiently find these stencils in the distributed mesh. In the APES framework, the solvers provide information of the used stencil. The framework then looks up the requested information in the distributed mesh of actually existing neighboring elements. This includes building up so called *halo* elements to receive data from remote processes.

Our approach for stencil-based neighbor identification is as follows: From the unique treeID, the element position and its size can be determined in terms of the tuple of the spatial coordinates and the level it resides on.

$$\text{treeID} \Longleftrightarrow \text{coord}(x, y, z, level) \quad (1)$$

This definition of cell indices defines a one-to-one relation between cell index and the treeID in the complete tree. With this relation it is straight forward to find treeIDs of arbitrary neighbors given by offsets relative to the current cell position. This identification of the required treeIDs within the stencil is purely local.

To obtain the processes, which are holding data for a desired treeID, we take advantage of the introduced global ordering. Since we cut the linearized information into partitions maintaining the ordering, it is sufficient to store the first and last treeID each process holds to obtain the partition in which a sought treeID resides. This partition identification of a treeID is then a purely local operation. The complete algorithm can be decomposed into the following steps:

- get the required treeID
- check if the required treeID is local
- create path from this treeID to the root element
- iterate over the remote processes and check if treeID is between first and last for one of these processes

Note, that the storage of each partition's first and last treeID requires $\mathcal{O}(p)$ memory consumption with a relatively low constant of 336bytes/process. Such a limitation does not hurt yet on a system like the Cray XE6 Hermit with around 10^5 cores and 1 GB of memory per core, but is already notable on a Blue Gene System like the Jugene with roughly $3 \cdot 10^5$ cores and only 512 MB of main memory per core. It will be most likely not feasible on systems with millions of cores. This is similar to the limitation of current MPI communicators, and might be overcome with the help of a distributed hash-like data structure at the cost of additional communication.

Only a single MPI Allgather communication is needed to exchange requirements from remote partitions before the local distributed search. Compared to fully unstructured meshes, our approach requires only very little communication with neighbor processes due to the implicit topology of the tree.

This strategy is not affected by local refinements and also works for arbitrarily refined meshes. The same applies for the following section which describes our approach to fully parallel input and output of mesh-related data.

B. *Parallel I/O*

The exploitation of truly parallel input and output (I/O) operations is a key feature to achieve high scalability on distributed systems. If we order the computing processes linearly (e.g. by the rank identifier in the context of MPI), each process gets a contiguous piece of data in a known region to read from disk. This enables each process to handle the I/O exclusively in its partition. Figure 4 shows a schematic representation of our I/O format for the mesh and related files, such as restart files.

The mesh file consists of a contiguous binary file with 16 bytes for each element. An 8 byte integer holds the treeID and another one holds the property information for each element. The property integer of each element in the mesh file is used as a bitmask to assign additional properties, e.g. boundary conditions, to each cell. Each bit indicates, if a certain property is active for the element. If so, there might be additional information attached to the element, which is stored in a separate file. Additional property data is only stored for those elements, where the property is active. This approach only requires once a counting of all elements with a given property on all processes and their global reduction with the help of a prefix summation, to find the offsets in the property data. Parallel prefix summation is provided by the MPI Exscan operation, which has a running time complexity of only $\mathcal{O}(\log p)$.

A text formatted header provides general information on the mesh. This includes the overall number of cells and a description of the spatial origin and length of the bounding cube. In case of a periodic domain without boundary informations or other properties, we end up with a total storage amount of 16 Bytes/cell. Our algorithmic approach for octree data storage is not restricted to the mesh itself. We can use it in a similar way for writing simulation results to disk.

C. *Common features of the solver framework*

On top of the octree framework our implementation of the solvers is organized in two additional algorithmic layers. The first provides generic functionality of common tasks in the octree and is common to all solvers. On top of this we introduce solver specific algorithms in a second layer. To summarize we have three levels of algorithms in our framework:

- basic algorithms of the octree mesh itself
- common algorithms of the solvers (often related to the octree mesh)
- solver-specific algorithms

It is worth mentioning that all the data structures in our framework are created such that they offer the possibility to operate in a level-wise manner. Clearly, this is an advantage in time-dependent explicit simulations where the size of a cell leads to a restriction of the maximum time step. Therefore, this strategy allows for local time-stepping algorithms to operate with optimal time-steps. To provide a detailed analysis of the performance of our solver framework it is necessary to consider all layers of our approach. Therefore, we also describe the solver-specific algorithms briefly in the following section.

III. Numerical Schemes

In the following we focus on transient flows, i.e. flows that involve time-dependent phenomena. We show how solvers based on FVM and LBM are integrated in the APES framework. The former is used to solve compressible flows while the latter is used for incompressible flow problems.

A. *Finite Volume Method for (non-)linear conservation laws*

The Finite Volume Method is a popular method for the numerical solution of conservation laws [8]. Conservation laws are special types of partial differential equations of the following type (where we assume Ω to be an open and bounded subset of three-dimensional Euclidean space):

$$\partial_t \mathbf{u}(x,t) + \nabla \cdot \mathcal{F}\left(\mathbf{u}(x,t)\right) = 0 \qquad \forall x \in \Omega \tag{2}$$

These types of equations cover a wide range of physical processes, such as electrodynamics and fluid dynamics. The equation type is determined by the flux function $\mathcal{F}$ and in case of Euler or Navier-Stokes equations the non-linearity of this function causes major problems. The discrete formulation of the FVM can be obtained from (2) by a tessellation $\mathcal{T}$ of Ω into a finite set of N cells (we denote the cells by $\mathcal{T}_i$ with $i = 1, \cdots, N$), integration over each of these cells and applying the Gaussian theorem:

$$\partial_t \int_{\mathcal{T}_i} \mathbf{u} dV = -\int_{\mathcal{T}_i} \nabla \cdot \mathcal{F}\left(\mathbf{u}\right) dV = -\int_{\partial \mathcal{T}_i} \mathcal{F}^* \cdot \mathbf{n} dS \tag{3}$$

On the right hand side we replaced the physical flux function $\mathcal{F}$ by its numerical counterpart $\mathcal{F}^*$ that is no longer depending on the state within the cell alone. Instead, it is a function of the state left and right of the faces of $\mathcal{T}_i$:

$$\mathcal{F}^* = \mathcal{F}^*(\mathbf{u}^-, \mathbf{u}^+) \tag{4}$$

Defining the integral mean for cell i by

$$\bar{\mathbf{u}}_i = \int_{\mathcal{T}_i} \mathbf{u} dV \tag{5}$$

we obtain a semi-discrete scheme for the integral mean of each cell:

$$\partial_t \bar{\mathbf{u}}_i = -\int_{\partial \mathcal{T}_i} \mathcal{F}^*\left(\mathbf{u}_i^-, \mathbf{u}_i^+\right) \cdot \mathbf{n} dS \tag{6}$$

The right hand side of the equation above is typically integrated by a quadrature rule (with quadrature points $x^{(1)}, \cdots, x^{(M)}$ and weights $w^{(1)}, \cdots, w^{(M)}$):

$$\partial_t \bar{\mathbf{u}}_i = -\sum_{k=1}^{M} w^{(k)} \mathcal{F}^*\left(\mathbf{u}_i^-\left(x^{(k)}\right), \mathbf{u}_i^+\left(x^{(k)}\right)\right) \tag{7}$$

In case of higher order Finite Volume Methods the values for u^- and u^+ on the right hand side are obtained with higher accuracy by a reconstruction step. The higher the accuracy, the larger the stencil has to be. In the following we focus on the MUSCL scheme [9]. This is a second order scheme and is the following fully discrete version of equation (7), where s denotes a properly defined (limited) slope:

$$\begin{aligned}
\mathbf{u}_i^{\pm,n} &= \mathbf{u}_i^n \pm \frac{\Delta x}{2}\mathbf{s} \\
\mathbf{u}_i^{\pm,n+\frac{1}{2}} &= \mathbf{u}_i^{\pm,n} - \frac{\Delta t}{2\Delta x}\left(\mathcal{F}(\mathbf{u}_i^{-,n}) - \mathcal{F}(\mathbf{u}_{i-1}^{+,n})\right) \\
\mathbf{u}_i^{n+1} &= \mathbf{u}_i^n - \frac{\Delta t}{\Delta x}\left(\mathcal{F}^*(\mathbf{u}_i^{+,n+\frac{1}{2}}) - \mathcal{F}^*(\mathbf{u}_i^{-,n+\frac{1}{2}})\right)
\end{aligned} \tag{8}$$

In case of the Euler equations we have the following set of conserved quantities: $\mathbf{u} = (\rho, \mathbf{j}, E)^T$, where ρ denotes density, $\mathbf{j}$ momentum and E energy. With the auxiliary definition of pressure, denoted by p and given by the equation of state, the Euler equations read:

$$\begin{aligned}
\partial_t \rho + \nabla \cdot \mathbf{j} &= 0 \\
\partial_t \mathbf{j} + \nabla \cdot \left(\mathbf{j} \otimes \frac{\mathbf{j}}{\rho} + p\mathbf{I}\right) &= 0 \\
\partial_t E + \nabla \cdot \left((E + p)\frac{\mathbf{j}}{\rho}\right) &= 0
\end{aligned} \tag{9}$$

One way to obtain the numerical flux over $\partial \mathcal{T}_i$ is to solve the so called Riemann problem for the Euler equations. This solves the above equation system subject to the initial condition

$$\mathbf{u}\left(x, t=0\right) = \begin{cases} \mathbf{u}_l & \text{, if } x < 0 \\ \mathbf{u}_r & \text{, if } x > 0 \end{cases} \tag{10}$$

An overview of available Riemann solvers is given in [10]. In the following we always use the so called HLLE-flux. Indeed, the flux calculation is one of the most expensive

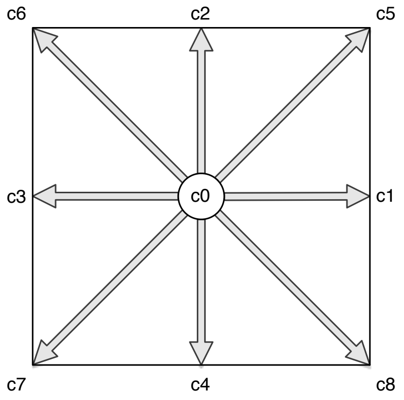

Figure 5. Schematic representation of a cell for the Lattice Boltzmann method in 2D for the D2Q9 model. The arrows indicate the direction of propagation in the streaming step.

parts of the solver, especially for non-linear conservation laws. To avoid redundant flux calculations in the FVM solver, we iterate over the faces instead of the cells. This saves roughly a factor of two in the effort for the flux calculation.

B. Lattice Boltzmann Method for incompressible flows

The Lattice Boltzmann Method (LBM) is an alternative numerical scheme for the simulation of incompressible flows. It has been shown that the LBM can be derived from Cellular Automata or as a special discretization of the Boltzmann equation [11]. The latter recovers the hydrodynamic macroscopic equations (i.e. the Navier-Stokes equation) in the limit of small Knudsen numbers. The LBM algorithm has the following form

$$f_i(x+1,t+1) = f_i(x,t) + \frac{1}{\tau}\left(f_i^{eq}(x,t) - f_i(x,t)\right) \quad (11)$$

Here, f denotes a probability density function, which describes the probability to find a particle with a certain velocity at $\vec{x}$. The upper algorithm is operating on a mesh of cubic elements, as we access the data at integer spatial and temporal coordinates. The index i is running over all the links of the lattice, i.e. 19 for the standard D3Q19 model in three dimensions. Figure 5 gives a typical shape of a cell in the LBM for the standard D2Q9 model. The links are numbered and denoted by c_i. In the following, Q denotes the number of links per cell.

The probability density function f is related to the macroscopic quantities of the incompressible Navier-Stokes equation by

$$\rho(x,t) = \sum_i^Q f_i(x,t)$$
$$\mathbf{j}(x,t) = \sum_i^Q \mathbf{c}_i f_i(x,t) \quad (12)$$

Algorithm (11) is usually decomposed into a collision

$$\hat{f}_i(x,t) \leftarrow \frac{1}{\tau}\left(f_i^{eq}(x,t) - f_i(x,t)\right) \quad (13)$$

and streaming step

$$f_i(x+1,t+1) \leftarrow f_i(x,t) + \hat{f}_i(x,t). \quad (14)$$

The collision step is a fully local operation and given by the so called BGK approximation. The streaming step involves pure memory copies of all Q local f values along the links to adjacent cells.

There are also a number of generalizations of the LBM. One is the so called multiple relaxation time (MRT) model. It replaces the scalar relaxation parameter τ in front of the collision step by a matrix M. The complete LBM-MRT method has the following algorithmic form (where the vector $\mathbf{f} = (f_1, \cdots, f_Q)^T$ is the collection of the linkwise probability density function f_i)

$$\mathbf{f}(x+1,t+1) = \mathbf{f}(x,t) + M\left(\mathbf{f}^{eq}(x,t) - \mathbf{f}(x,t)\right) \quad (15)$$

Although the MRT algorithm involves more floating point operations due to the matrix-vector multiplication, the general structure of the LBM is unmodified. In the following section we consider both types of the LBM with respect to performance.

IV. Performance Results

We present performance studies of our implementations on the Cray XE6 system Hermit at the HLRS in Stuttgart. The Hermit system (Cray XE6) consists of 3552 computing nodes with 32 cores each. We investigate strong and weak scaling behavior for a number of problem sizes. As a test case we use a fully cubic domain and a Gaussian pulse. Our test case for the performance and scaling measurements consists of a fully periodic, cubic simulation domain. Refining the overall simulation domain by one level increases the number of cells by a factor of eight. We consider runs with up to 3072 nodes of our solver and cover refinement levels three to eleven (i.e. 8^3 to 8^{11} cells) with uniform cell sizes.

1) Performance Measure: For the following discussion, we define *MCUPS* (million cell updates per second) as a measure of the system performance. Update refers to a complete time step cycle of the computing algorithm. Obviously, this defines an absolute performance measure (i.e. it depends on the number of cores we use for parallel execution). Usually this performance depends on the problem size in serial executions, as it is influenced by cache usage, non-computational implementation overheads and so on. On the other hand, ideal parallel execution is expected to just replicate the serial behavior on each execution unit. Thus, the MCUPS are to be proportional to the number of processes in the parallel simulation run. It is therefore useful to define a relative performance measure, which indicates the behavior of the application per execution unit. On supercomputing systems it is usually not useful, or even possible to use less than a dedicated node. Due to this, a meaningful execution unit is given by a single node. For this purpose we define the number of million cell updates per second per node (MCUPSN) as

$$\text{MCUPSPN} = \frac{\text{MCUPS}}{\#\text{nodes}}. \tag{16}$$

As already pointed out, this measure would be constant over the number of processes in case of an ideal scaling. Combining the baseline serial behavior over the problem size with the replicated execution on parallel units in a single graph, we end up with a performance map (see e.g. Figure 6) characterizing the overall runtime behavior of the application. This is achieved by plotting the MCUPS per execution unit over the problem size per execution unit with one data series per process count. The expectation is, that all graphs for multiple processing units are below that of the single processing unit, since they have additional communication costs attached to them.

For the overall execution it is important to first understand the application behavior within a single node, and then make optimal use of each node in runs with multiple nodes. We first performed these intranode performance runs with $1, 2, 4, 8, 16$ and 32 cores. To obtain the best performance of a single node, we use pinning to distribute the processes within the node to a specific core.

2) FVM Performance and Scaling: The physical test case in the cubic, periodic domain is based on a Gaussian pulse in density and constant values for all other variables. Due to the nature of the octree and the periodicity, we gain self-similar problems, whenever the domain is subdivided into smaller partitions by a factor of eight. We make use of this fact to obtain perfect weak scaling problems, where the work done by each processor is exactly the same for the different processor counts. That is, number of cells to compute, amount of data to communicate and number of neighbors to communicate with, stays constant within the weak scaling experiment.

In each time step we start with a communication of the integral mean values which are used for the reconstruction in space and are covered by stencils of cells on other processes. Each cell reconstructs a linear interpolation of the numerical solution by means of the neighboring six nodes in a dimension-by-dimension approach as suggested by the MUSCL approach. Therefore, this communication involves exactly five floating point numbers per communicated cell. Finally, we extrapolate the reconstructed values to faces of the cell and extrapolate in time. Another communication of the processes exchanges information about these face values. In this communication it is sufficient to communicate only those faces that are really requested by the neighboring process. Then the flux calculation over all the faces for each cell can be performed as well as the actual update of the

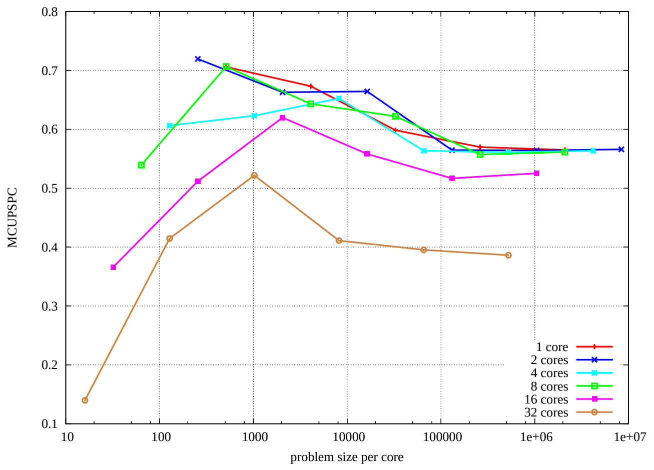

Figure 6. Performance map of the FVM within one node of the Cray XE6 system Hermit with AMD Interlagos processors

cell values for the current time step. All in all we obtain two communications in our simulation loop, all other operations are purely local. One cycle of this whole simulation loop requires roughly 630 floating point operations per element.

The results obtained for a scaling test within a single node are shown in Figure 6 with a performance map with respect to a single core as execution unit.

This already exposes some features of our implementation: For a very small number of cells per core we clearly see additional overhead of the code. In parallel it is mainly due to the relatively large communication surface for the small partitions. As this overhead is diminishing relatively for larger problem sizes per core, we see a strong dependency of the performance on the problem size. The highest performance per core is obtained when the system is able to hold the complete memory of the computing loop in its cache(s). Once the cache size is exceeded the performance per core flattens out. Thus we can mainly distinguish three different regions:

1) Overhead dominated very small problem sizes with a strong gradient in the performance with increasing problem size.
2) Problem sizes which fit into cache.
3) Problem sizes which require frequent access to the main memory.

Furthermore we notice that due to the hardware architecture of the AMD Interlagos CPUs of the Hermit system the performance per core in case of 16 core execution is significantly higher than the one with 32 core execution. Nevertheless the higher the number of cores the higher the overall performance becomes. Our FVM implementation achieves the highest single node performance when using all 32 cores.

For the performance analysis beyond one node, we there-

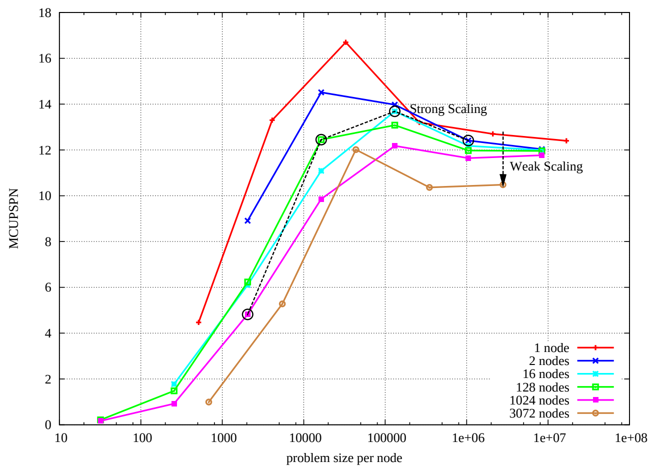

Figure 7. Performance map of the FVM implementation between nodes on Hermit with indication to weak and strong scaling

fore use all 32 cores on each node. This is shown in Figure 7, where we plot the problem size per node on the horizontal axis and the value of MCUPSPN on the vertical axis. Compared to the intranode performance analysis, the communication now involves network transfers, which drastically increases the time spent on communication. We see this effect clearly for small problem sizes per node with a large ratio of communication to computation and low computing efficiency. The larger the number of cells per node, the smaller this ratio becomes. Figure 7 represents this fact by the very steep slope on the left of the graph. Similar to the intranode performance analysis, we reach a maximum when the required memory fits exactly into the cache of each node. However it is diminished by an increasingly large communication overhead for larger node counts. In Figure 7 we can see this optimal point around 16384 cells per node. Once we have exceeded the cache we see again that the performance flattens out.

Please note, that Figure 7 also contains the information about weak and strong scaling as indicated by the dashed lines. The weak scaling can be obtained by the vertical connection of points as this corresponds to a fixed number of cells per node. The closer the points are located to each other the better the weak scaling. Strong scaling can be seen by connecting corresponding points of same overall problem sizes over the various lines with different node counts. That is connecting the rightmost point on 2 nodes with the next less problem size on 16 nodes and so on. It can be seen, that the scaling works fine on most process counts for all problems, that fit into memory down to the cache-sized problems, where the communication gets dominant and we see a steep decreasing of the performance per node with smaller problems per node.

In Figure 8 we provide an explicit overview for the weak

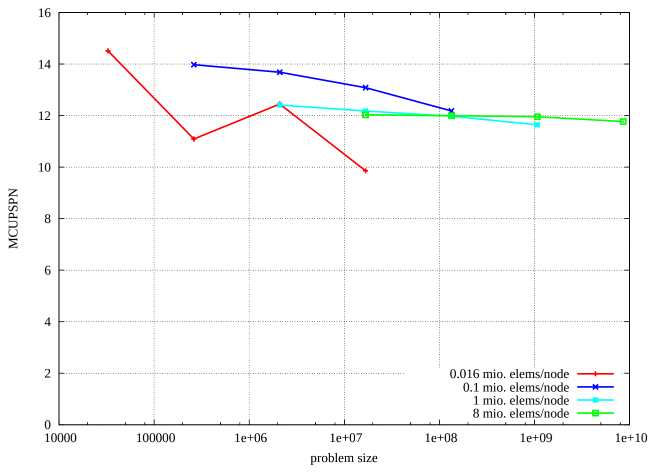

Figure 8. Weak scaling of the FVM implementation. The vertical axis represents the number of cell updates per second and node. The points for each line in the graph represents executions of the solver with $2, 16, 128$ and 1024 nodes.

scaling with different cell counts per node. For a very small problem with 16384 cells per node, fitting into the cache, the scaling is quite bad. In the performance map, Figure 7, this is indicated by the large vertical distance between the node counts at this point. For larger cell counts per node, the scaling gets better and is nearly ideal for problems with more than 1 million cells per node. The run on 3072 nodes is included for completeness, even so it imposes a less ideal partitioning. It shows a drop for a node count which is not a power of two, however the performance achieved per node is still acceptable.

A dedicated graph of the strong scaling is shown in Figure 9. We consider testcases with 8^4 to 8^9 cells total domain size. As already pointed out previously we can find information about strong scaling in Figure 7 by consideration of *diagonal* points. We start with our analysis for 2 million cells in total. It is clearly visible that the performance on 16 nodes is even better than on two nodes. Comparing both points with our performance map in Figure 7 we can immediately recognize that the 16 node test case (resulting in 131072 cells per node) reaches a region with caching effects, which explains the super-linear speedup. Increasing the number of computing nodes by a factor of 64 again, leading to 2048 cells per node, drastically decreases the performance per node resulting in approximately 40% of parallel efficiency. For smaller problem sizes per node the situation is even worse. In Figure 9 we observe that the scaling improves with the overall problem size. With 16 million cells, which is the largest problem in our series, we can fit on 2 nodes, we can scale well up to 1024 nodes. All in all, the analysis shows, that we can speedup a problem that fits onto a single computing node roughly by a factor of 1000. This is also

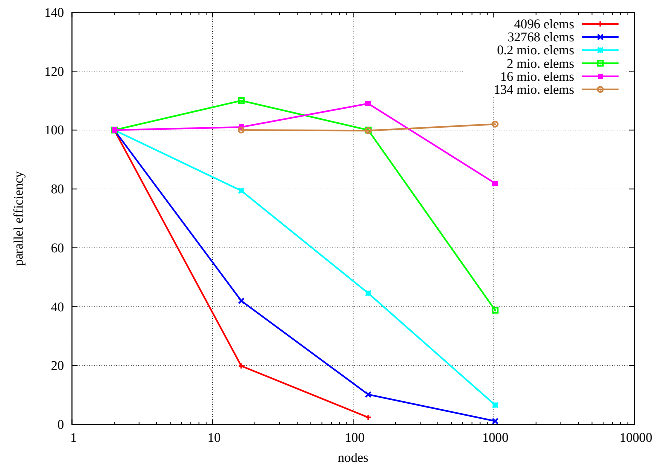

Figure 9. Parallel efficiency for strong scaling of the FVM implementation

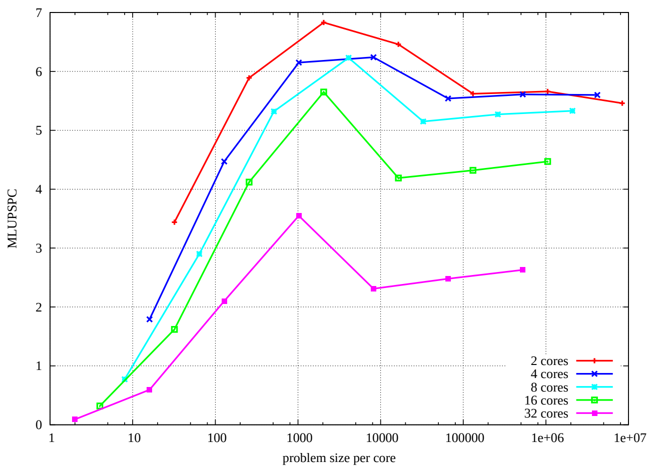

Figure 11. Intranode performance map for the LBM-BGK model

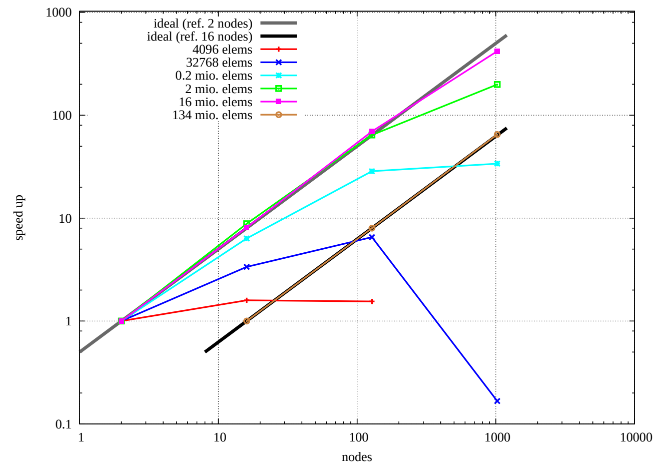

Figure 10. Speed up plot of the FVM implementation compared to the running time on 2 nodes.

roughly the relation of available main memory per core to the available cache.

The corresponding speed up plot is shown in Figure 10. Please note, that we included two different ideal speed up references due to the fact that we used a different normalization for the 134 million cells problem as it does not fit onto 2 nodes.

3) LBM Performance and Scaling: Similar to the performance study for the FVM implementation, we use a fully periodic, cubic simulation domain with powers of eight as problem sizes. As usual in the LBM context, we refer to lattice updates instead of cell updates here and change MCUPS and MCUPSPN to MLUPS (i.e. million lattice updates per second) and MLUPSPN (i.e. million lattice updates per second per node).

BGK model performance analysis: Again, we start with an analysis of the intranode performance analysis. Figure 11 is showing the measured data for the LBM-BGK implementation. We observe the same three regions in our performance graph. Though we see a decreasing performance per core if more cores are used within the node, deploying all 32 cores still provides the highest performance per node. Therefore, we again consider full node computations in the following internode performance analysis.

The internode performance map of the LBM-BGK model is shown in Figure 12. We consider testcases with 8^3 to 8^{11} cells. Like for the FVM analysis we use core counts of powers of eight. We observe the large influence of the communication for small problems per node. This results in steep gradient of the performance graph in Figure 12 for problem sizes with less than 10^5 lattices per node on more than two nodes. Benefits from the caches are completely diminished by the increased communication in these cases. Due to the highly optimized kernel with only around 160 operations per lattice update and therefore low computing time we still see an influence of the problem size on the performance for larger problems. Thus strong scaling will be worse than in the FVM application, while we still see a nearly ideal weak scaling.

The weak scaling of our LBM-BGK implementation is shown in Figure 13. We consider testcases between $1.3 \cdot 10^5$ and $8.4 \cdot 10^6$ cells per node. The points of all data lines of Figure 13 correspond from left to right to $2, 16, 128$ and 1024 compute nodes of the Hermit system. For the smallest testcase of $1.3 \cdot 10^5$ cells per node we observe the following situation: The two node execution is still in a region where the cache is speeding up the calculation. For larger numbers of nodes the increasing amount of communication overhead is decreasing the performance per node at this problem size per node. In the performance graph (Figure 12) this is

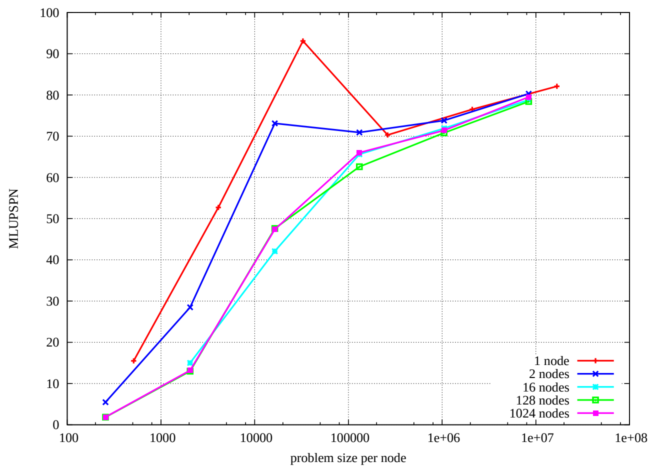

Figure 12. Internode performance map for the LBM-BGK model

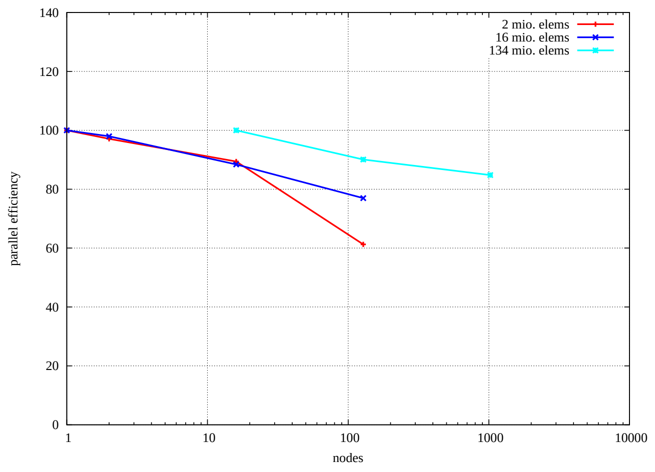

Figure 14. Parallel efficiency for strong scaling of the LBM-BGK implementation on the Hermit system for different problem sizes

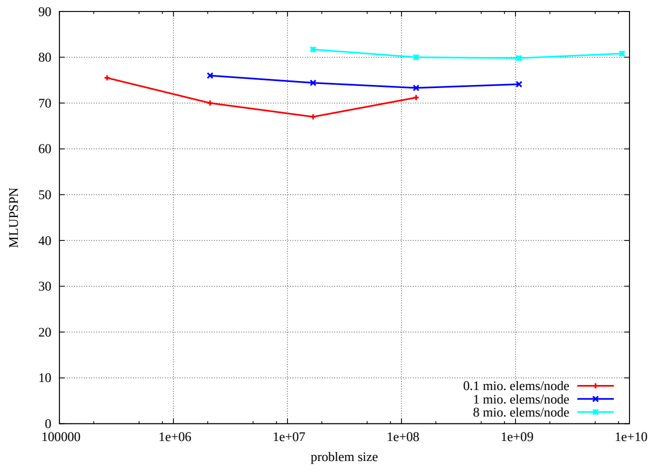

Figure 13. Weak scaling of the LBM-BGK implementation on the Hermit system for different problem sizes per node.

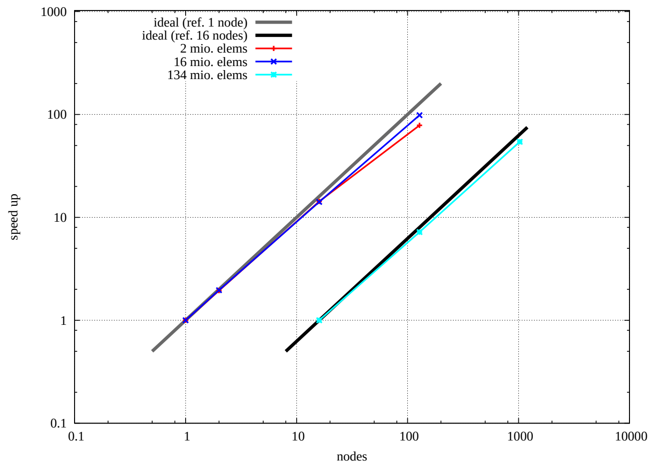

Figure 15. Speed up plot of the LBM-BGK on Hermit

represented by the fact that the points for $1.3 \cdot 10^5$ cells per node are distributed vertically. The larger the problem size per node, the less significant the influence of communication and therefore the better the weak scaling. For more than one million cells per node points at fixed number of cells per node nearly coincide. In our dedicated Figure 13 we can find this observation, too. For sufficiently large problem sizes per node the weak scaling is nearly perfect.

Figure 14 shows the strong scaling for our LBM-BGK implementation. We remind the reader that the strong scaling can be obtained by connecting *diagonal* points in the performance graph in Figure 12. In a region of up to 10^5 cells per node the communication time is dominating the computation time, resulting in a steep performance gradient on more than 2 nodes. Figure 14 shows this effect on the strong scaling for the two million cell test case. Compared to the FVM, we find a higher number of cells per processor is necessary to sustain the computations. For a total problem size of approximately 130 million cells we can scale up to 1024 nodes with more than 80 percent parallel efficiency. This leads to approximately $1.3 \cdot 10^5$ cells per node, beyond this process count a steeper drop in the performance is expected. All in all it is possible to scale a problem that fits into the memory of a single node roughly by a factor of 600 without a significant loss of parallel efficiency for the LBM-BGK. The corresponding speed up graph is shown in Figure 15.

MRT model performance analysis: In the following we discuss the performance of the LBM-MRT implementation. We already mentioned that the structure of the algorithm is the same, but the number of floating point operations

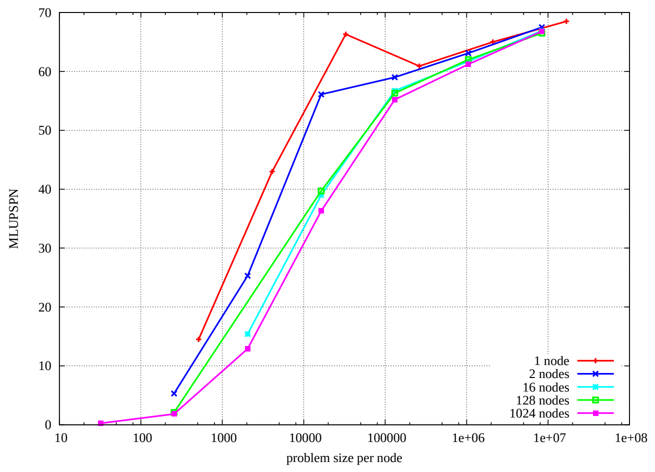

Figure 16. Performance graph for the LBM-MRT. Parameters are the same as in Figure 12.

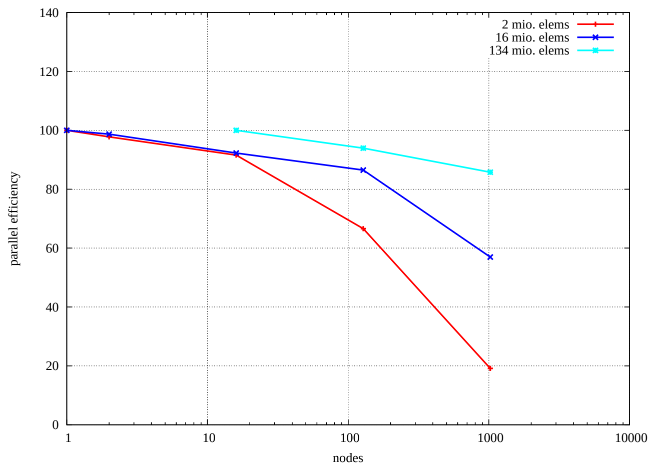

Figure 17. Strong scaling of the LBM-MRT implementation. We use the same test sizes as in Figure 14.

per byte is increased. In our implementation the number of floating point operations is approximately three times higher for the LBM-MRT than for the LBM-BGK. However, the amount of communicated data remains the same. Figure 16 is showing the performance graph of the MRT model. Since the computational part of the solver is larger than in the BGK model, the influence of the communication is vanishing earlier.

We omit details of the weak scaling graph of the LBM-MRT, as it resembles that of the LBM-BGK. For smaller problem sizes per node, the weak scaling of the LBM-MRT is slightly better, due to the fact that the number of floating point operations per lattice update is much higher.

The strong scaling of the MRT model is shown in Figure 17. We observe an improvement for larger number of cores for a fixed total problem size compared to the BGK model. For a total problem size of 16 million cells and 128 compute nodes we reach more than 80% parallel efficiency (i.e. 86.5% parallel efficiency). For the LBM-BGK we reached for the same setup only 76% of parallel efficiency.

The measured performance for the MRT Lattice Boltzmann model corresponds to a sustained performance of around 9% of peak performance.

V. Conclusion and Outlook

We have shown that the implementations of the Finite Volume and Lattice Boltzmann method is performing well on a Cray XE6 system. Both methods show a nice weak and strong scaling behavior. In general we are able to scale the largest problem size that fits on a single node by a strong scaling by a factor of 1000 for the Finite Volume method and by a factor of 600 for the Lattice Boltzmann method without a significant loss of parallel efficiency. We pointed out that the scaling properties of the MRT model are slightly better than for the BGK model. Especially the fully distributed approach to the simulation enables runs for very large problems, we managed to run testcases with up to 68 billion elements on 3072 nodes. This shows, that the scaling in memory does not hinder full usage of the distributed memory system.

The memory consumption per cell is higher in case of the Finite Volume method, leading to smaller maximum problem sizes per node compared to the LBM BGK or MRT model. It is worth mentioning that our Finite Volume implementation is not optimized with respect to serial performance yet, but this implementation issue should nicely fit into the chosen approach, as the LBM kernels already show.

All in all, we have shown that our implementations are able to use the Cray XE6 Hermit system efficiently and we are able to scale down to cache size without a significant loss of efficiency. Furthermore, we outlined the need for implicit knowledge as provided by the octree is essential for distributed computations.

In the future we are going to investigate our octree based framework in several directions. One of the open issues, is the study of local refinement and load imbalances. We have already shown that the octree framework is still able to execute all its tasks in parallel with a minimum amount of synchronization. We will consider load balancing for octree meshes from a broad perspective and investigate it with respect to memory balancing, workload redistribution and a well defined mixture of both.

We have also seen that a strong scaling is usually limited by a high ratio of communication overhead to compute time for small problem sizes per node. In these regions the performance per node drops down significantly. Usage of OpenMP or communication hiding by overlapping the computation, might help here. In the context of Finite

Volume methods, the influence of higher order schemes on computational efficiency could also improve the scalability and go hand in hand with a hybrid parallelism.

VI. Acknowledgment

This work has been funded by Bundesministerium für Bildung und Forschung (Federal ministry for education and research, BMBF) in the framework of the HPC software initiative in the project HISEEM. We thank the Gauss Alliance of German Supercomputing Centers for the provided computing time. We are grateful for the ongoing kind support by HLRS and Cray in Stuttgart. For insightful discussions and performance hints for the Lattice Boltzmann solver we thank Thomas Zeiser, RRZE Erlangen.

References

[1] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Adaptive local refinement with octree load balancing for the parallel solution of Three-Dimensional conservation laws," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 139–152, Dec. 1997.

[2] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, "Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees," Dec. 2009.

[3] M. Mehl, T. Neckel, and P. Neumann, "NavierStokes and LatticeBoltzmann on octreelike grids in the peano framework," *International Journal for Numerical Methods in Fluids*, vol. 65, no. 13, pp. 67–86, Jan. 2011.

[4] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K. Ma, and D. R. O'Hallaron, "From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[5] H. Klimach and S. Roller, "Distributed coupling for multiscale simulations," in *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, P. Ivanyi and B. Topping, Eds. Civil-Comp Ltd., 2011.

[6] Morton, "A computer oriented geodetic data base and a new technique in file sequencing," IBM Ltd., Tech. Rep., 1966.

[7] D. F. Harlacher, M. Hasert, H. Klimach, S. Zimny, and S. Roller, "Tree based voxelization of stl data," in *High Performance Computing on Vector Systems 2011*, M. Resch, X. Wang, W. Bez, E. Focht, H. Kobayashi, and S. Roller, Eds. Springer Berlin Heidelberg, 2012, pp. 81–92.

[8] R. J. Leveque, *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, Aug. 2002.

[9] B. van Leer, "Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method," *Journal of Computational Physics*, vol. 32, no. 1, pp. 101–136, 1979.

[10] E. F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, Apr. 2009.

[11] X. He and L.-S. Luo, "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation," *Physical Review E*, vol. 56, no. 6, pp. 6811–6817, 1997.