Exploiting Hybrid Parallelism in the LBM Implementation Musubi on Hawk

Harald Klimach, Kannan Masilamani and Sabine Roller

Abstract In this contribution we look into the efficiency and scalability of our Lattice Boltzmann implementation *Musubi* when using *OpenMP* threads within an *MPI* parallel computation on *Hawk*. The Lattice Boltzmann method enables explicit computation of incompressible flows and the mesh discretization can be automatically generated, even for complex geometries. The basic Lattice Boltzmann kernel is fairly simple and involves only few floating point operations for each lattice node. A simple loop over all lattice nodes in each partition of the *MPI* parallel setup lends to a straight forward loop parallelization with *OpenMP*. With increased core counts per compute node, the use of threads on the shared memory nodes is gaining importance, as it avoids overly small partitions with many outbound communications to neighboring partitions. We briefly discuss the hybrid parallelization of *Musubi* and investigate how the usage of *OpenMP* threads affects the performance when running simulations on the *Hawk* supercomputer at *HLRS*.

1 The Lattice Boltzmann method

The Lattice Boltzmann method (*LBM*)[9] offers an efficient explicit method to compute incompressible or weakly compressible flows by modelling the gas with a discrete velocity space for the Boltzmann equation in a mesoscopic scale. For the discretization a regular mesh is used, usually with cubic cells, and the connections to the neighbors offer the discrete velocity directions to be considered in the numerical method. Hence, *LBM* can be considered as a stencil method, with similar properties in communication and parallelization as other mesh-based methods. An advantage of *LBM* can be observed in the treatment of boundaries. Due to the use of discrete

Harald Klimach, Kannan Masilamani and Sabine Roller
DLR e.V., Institut für Softwaremethoden zur Produkt-Virtualisierung, Zwickauer Str. 45, 01069 Dresden,
e-mail: harald.klimach@dlr.de, kannan.masilamani@dlr.de, sabine.roller@dlr.de

M. M. Resch et al. (eds.), *Sustained Simulation Performance 2021*,
https://doi.org/10.1007/978-3-031-18046-0_4

velocities, boundaries only have to be considered along those discrete directions and accurate boundaries can be obtained by intersecting the one dimensional lines in cubical boundary cells with the surfaces describing geometrical boundaries. Such line intersections with surfaces can be computed robustly and are, thus, well suited for automated mesh generations.

The most commonly used stencil in three dimensions are *D3Q19* and *D3Q27*. The *D3Q19* makes use of 18 neighbors together with the state at rest resulting in 19 values of the probability density function to describe the fluid state. In this stencil, 18 neighbors are all immediately connected cells except for those at the corners of the cube (6 sides and 12 edges of the cube). Other stencil *D3Q27* is required in some *LBM* approaches which makes use of all 26 immediate neighbors.

The lattice Boltzmann equation with classical collision operator from Bhatnagar, Gross and Krook *BGK*[1] is given by

$$f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) - f_i(\mathbf{x}, t) = \Omega_i(\mathbf{x}, t) \tag{1}$$

where $f_i(\mathbf{x}, t)$ is the probability density function at the position vector $\mathbf{x}$ and at time step t along the discrete direction i; Δt is the discrete time step; $\mathbf{c}_i$ is the discrete velocities and Ω_i is the collision operator. There is a multitude of collision operations available. However, here we will only consider the classical operation described by Bhatnagar, Gross and Krook (*BGK*) as

$$\Omega_i = -\frac{1}{\tau}(f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)) \tag{2}$$

where $\mathbf{f_i^{eq}}$ is the Maxwell-Boltzmann distribution function. For weakly-compressible flows, it is given by

$$f_i^{eq}(\rho, \mathbf{u}) = \omega_i \rho \left(1 + \frac{(\mathbf{c}_i \cdot \mathbf{u})}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{(\mathbf{u} \cdot \mathbf{u})}{2c_s^2}\right), \tag{3}$$

where ω_i are the lattice weights and $c_s = c/\sqrt{3}$ is the speed of sound in lattice. $c = \Delta x/\Delta t$ is the lattice velocity where Δx is the discretization size. ρ and $\mathbf{u}$ are macroscopic density and velocities which are computed from probability density function by

$$\rho = \sum_{i=1}^{Q} f_i \tag{4}$$

and

$$\rho \mathbf{u} = \sum_{i=1}^{Q} \mathbf{c}_i f_i. \tag{5}$$

The pressure P is calculated from the density ρ using the equation of state relation as $P = c_s^2 \rho$. The relaxation time τ in Eq. 2 is related to kinematic viscosity as

$$\nu = c_s^2 \Delta t \, (\tau - 0.5) \, . \tag{6}$$

In LBM, Eq. 1 is solved in two steps: streaming and collision. Streaming is exchanging the probability density functions of the particles along their respective directions with neighboring cells and collision is computing a new state in each cell according to collision operator. In general, the collision requires only few floating point operations per cell.

2 The Musubi implementation

Musubi[3] is our open source implementation of the Lattice Boltzmann method, mostly written in Fortran 2003 and primarily parallelized with via the Message Passing Interface (*MPI*) [6]. It makes use of an octree mesh discretization with cubical cells. Cells in the mesh are sorted according to the Morton or Z curve [5], which provides some maintaining of the multi dimensional locality in the one dimensional sorted list of cells. The solver works on the cells according to that ordering and the mesh partitioning is achieved by splitting the ordered list of cells into equally sized chunks. A double buffer is used to hold the state and allow the access to the previous iteration in the streaming step. Some additional arrays are used to hold further auxiliary values for all cells. All in all we find a computational intensity of around $1/3$ floating point operations per Byte.

Meshes can be created with the mesh generator *Seeder* [2], which provides the mesh in this form of an ordered list of cells or for simple meshes. This allows for a distributed reading of the mesh information as each process can identify the part of the file it needs to read with little information on the mesh. Alternatively, simple meshes like the ones we will consider here, can also be generated by *Musubi* itself. The list of cells may be sparse and thus, explicit neighborhood information is needed to address the stencil cells. Hence, the stencil implementation here behaves as an unstructured mesh with indirect addressing of the stencil cells. However, the known topology of the octree and the ordering according to the space-filling curve enables the identification of neighbor cells across partitions in the distributed memory parallel computation. Therefore, nearly arbitrary stencils can be employed and *Musubi* makes use of that in the implementation of the various *LBM* schemes. Meshes might have cells on different levels of the octree refinement, but on each level the kernel just acts as if working on uniform mesh. This is achieved by ghost cells that provide interpolated values from other refinement levels. Due to this behavior it is possible to perform some assessment of fundamental properties of this kind of kernel in a single level uniform mesh, which we look at in this contribution.

2.1 OpenMP in Musubi

The *OpenMP* parallelization[8] in Musubi is incomplete and various features do not yet benefit from it. But the parallelization of the fundamental kernel is straight forward, as it essentially is a single loop over all cells to update the lattice nodes. An *OpenMP* parallel region is put around these loops to realize the shared memory parallelism within *MPI* processes. The *MPI* communication itself is not put into a parallel region and does not profit from shared memory parallelization. With a static schedule the loop parallelism this way results in a partitioning similar to the *MPI* partitioning as the cells are sorted according to the space-filling curve. Accordingly the expectation is that the degree of parallelism can be shifted interchangeably between the one and the other.

3 Hybrid parallelization

In supercomputing systems a hierarchy of parallelism and data access can be observed. The most obvious decomposition can be observed in the construction of large clusters from individual nodes. Where individual nodes provide shared memory access between all processing units within it. The number of processing units within such a node mostly depends on the number of cores we find in the employed processors. And accordingly, we observe a growing degree of shared memory parallelism within those nodes as the number of cores in modern processors increases. Using a distributed memory parallelization concept uniformly for all processing units is possible, but results in small partitions that end up with many individual neighbor partitions that may be located on other nodes. This results in a larger number of smaller network communications between nodes. A strategy to minimize this effect and obtain larger *MPI* partitions with fewer, but larger network communications, is to resemble this two-level hierarchy of the hardware in the application. With *OpenMP* in each *MPI* partition processing units can be dedicated to parallel work in a reduced number of distributed memory partitions. Such a strategy than results in a less fragmented communication pattern across the network of the cluster.

Unfortunately, there are also some downsides involved in the hybrid parallelization. The management of threads results in some overheads and we increase the risk encounter resource conflicts in commonly used resources in the node, like shared caches or memory interfaces. As long as there are parts of the code that do not benefit from *OpenMP* parallelization, we face the problem that some parts do not benefit from a larger number of threads, but would benefit from more *MPI* processes. In the end the optimal choice depends at least on the system, the application is run on. It also depends on the specific setup to be run, but here we want to look at properties of the fundamental kernel, which may also be instructive in a wider context.

4 Performance assessment on Hawk

To evaluate the effect of shifting parallelism between *MPI* and *OpenMP* on *Hawk*, we run *Musubi* for various problem sizes on a range of node counts. These runs are performed with different numbers of threads per process, such that always all cores are participating in the computation.

4.1 The Hawk computing system

The *Hawk* computing system installed at the High Performance Computing Center Stuttgart (*HLRS*) is based on the *AMD EPYC 7742* processor[7], which has 64 cores operating at 2.25 GHz and *AVX2* vector instructions, yielding a theoretical peak performance of 2.3 TFLOPS. Each node has two of these processors and, thus, has a total of 128 physical cores. There are groups of 4 cores that share their L3 cache and build a so called *CoreCompleX*. Two of those *CoreCompleXs* are paired together and share one of the 8 memory channels in the processor. Finally two of those memory channels are put into a NUMA node per socket. Accordingly we see 16 physical cores in each NUMA node and with the two sockets in each computing node a total of 8 NUMA nodes. We will consider an *OpenMP* parallelism of up to 16 threads per *MPI* process, which corresponds to one *MPI* process per NUMA node. A further increase in the degree of shared memory parallelism is expected to incur degrading performance in comparison to a distributed memory strategy, due to the strong hierarchical structure of the memory access paths. Because of the shared L3 cache, a natural choice for the degree of shared memory parallelism in this system is 4 threads per process, putting all cores in a *CoreCompleX* to work on a shared memory region. Each memory channel provides a bandwidth of around 24 GB/s resulting in a total of 192 GB/s per socket or 384 GB/s per node to access the capacity of 256 GB.

4.2 The Musubi setup

For this contribution we use *Musubi* in version *9ccba4387413* [4]. It is using the environment offered by the modules *gcc/9.2.0* and *mpt/2.23* and with *OpenMP* support. The simulation setup is a small initial pressure pulse in a cubic domain of edge length 10 that is periodic in all directions. This simple mesh can be generated by *Musubi* during the simulation and can be easily scaled up in factors of 8. The initial spherical pulse is located in the center of the domain, has an amplitude of 1.2 over a background value of 1 and a halfwidth of 1. As collision operator we use *BGK* and we look at the *D3Q19* and *D3Q27* stencils. Each simulation is executed for at least

5 minutes of running time. The executable is run with the following command, with *nprocs* representing the number of *MPI* processes and *tpp* the number of *OpenMP* threads per process:

```
mpirun -np $nprocs omplace -tm pthreads -nt $tpp
```

4.3 Results

Performance for Lattice Boltzmann methods is usually measured in million lattice updates per second *MLUPS* and we consider this measure here per node. This measure is independent of the actual running time and allows for a comparison between different runs and simulations.

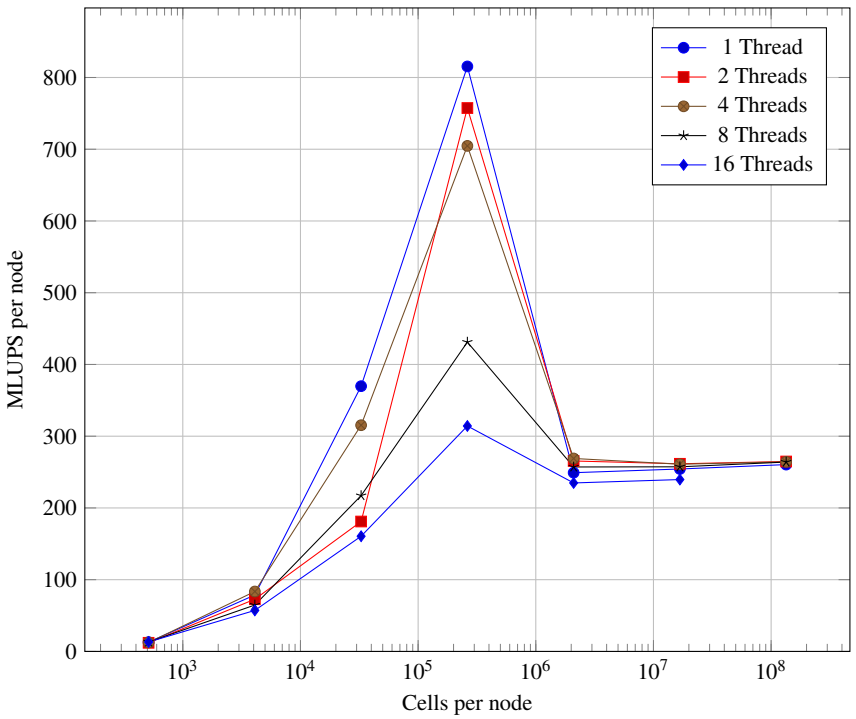

Fig. 1: Performance for *D3Q19* on a single node, utilizing all 128 cores.

As described above we perform runs with varying problem sizes on a node and record the resulting performance measure in *MLUPS*. Figure 1 shows the resulting graph for the *D3Q19* stencil on a single node. This form of representation nicely shows the variation of the performance with the problem size. Overheads, like

communication, dominate for very small problem sizes at the left end of the graph. Then a peak can be observed where the overall problem is still sufficiently small to completely fit into the caches, avoiding slower memory accesses. Finally, a relatively flat performance plateau is reached for larger problem sizes, until it does not fit into the memory of the node anymore.

What we can observe in this single node analysis is that in the region with memory access with more than a million cells, there is basically no performance difference between 1, 2, 4 and 8 *OpenMP* threads per *MPI* process. With 16 threads the performance is clearly reduced, which seems to indicate that it is important for *Musubi* that an *MPI* process is not spread across multiple memory channels. In the cache region we also see a clear diminishing of the performance for 8 *OpenMP* threads, where the shared memory of a process spans across two *CoreCompleXs* and accordingly two shared L3 caches.

For the *D3Q27* stencil a similar behavior can be seen in figure 2. With 27 discrete velocity directions to represent the state, more memory is required to represent the state in each cell and less cells fit into the memory of the single node than with 19 directions only. The largest domain that still fit into memory for the *D3Q19* stencil (134, 217, 728 cells), therefore, does not fit here anymore and the graphs, and the largest mesh we compute is 16, 777, 216 cells large. Note, that this is a lot smaller than what would fit into the memory, which would be more than 90 million cells.

This single node analysis shows the principal behavior of the *LBM* implementation in *Musubi* on each node of *Hawk* and illustrates the performance impact of the different parallelization strategies on the hierarchical memory layout of the system. We can note that the use of 4 *OpenMP* threads on as many cores yields roughly the same performance as a *MPI*-only parallelization and in the region with memory access up to 8 threads can be used interchangeably to *MPI* parallelism.

The analysis on a single node, however, does not show how the use of *OpenMP* threads influences the network communication between nodes. As stated above a motivation to make use of *OpenMP* parallelism is to reduce the number of individual communication partners with whom comparably small messages need to be exchanged. To assess this, we repeat these runs on larger node counts with 8, 64 and 512 nodes. Incrementing by a factor of 8 yields here the same problem sizes per node again, and allows for a direct comparison of the individual data points.

For brevity we only depict the corresponding graphs for 512 nodes. In this case we have 65, 536 cores working in parallel. This analysis is shown in figure 3 for *D3Q19* and in figure 4 for *D3Q27*. As can be seen in these figures, the behavior on 512 nodes is quite similar to the one on a single node. However, we also observe some differences. Most importantly we now see that for small problem sizes per node a higher performance is achieved with 2 and 4 threads and not with the pure *MPI* parallel computation.

In confirmation to the observation for a single node it appears reasonable to make use of a single *MPI* process for each *CoreCompleX* with the cores that share their L3 cache also sharing their memory address space. Shared memory parallelization beyond that diminishes the performance in the cache region with small cell counts per node, but for larger problems with the need to access the memory, also larger

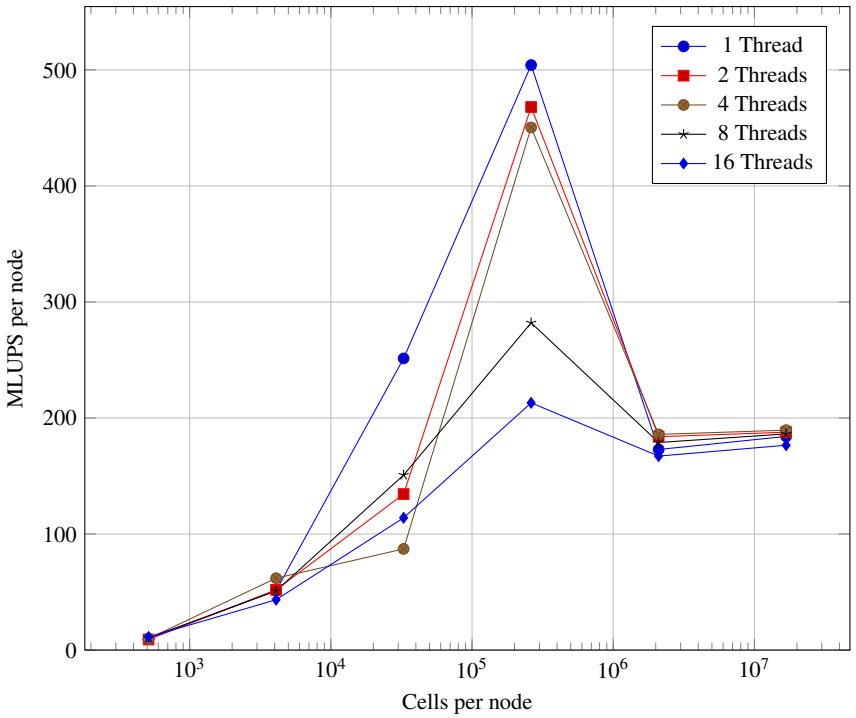

Fig. 2: Performance for *D3Q27* on a single node, utilizing all 128 cores.

shared memory processes with up to 8 cores (sharing one memory channel) can be utilized. Though for smaller problems, where all elements would fit into the caches, a larger performance decrease can be observed for those shared memory partitions spanning more than a single *CoreCompleX*.

Using 16 cores, spanning two memory channels in a NUMA node, as a shared memory parallel group within an *MPI* incurs too many drawbacks in the memory access of the hierarchical processor design to be used efficiently by *Musubi* also on 512 nodes.

For the scaling we look at the *D3Q27* stencil as the more memory and communication intensive scheme and stick to the parallelization with one *MPI* process per *CoreCompleX* and 4 *OpenMP* threads per process to allow concurrent computation on the 4 physical cores. As we have seen in the above measurements this configuration nicely fits the physical properties of the processor and provides good performance across problem sizes.

We also include the computation on 2048 nodes here, though these do not result in exactly the same number of cells per node as the other runs. This is the maximal number of nodes available to users in the regular queue on *Hawk* and provides 262, 144 physical cores for parallel execution. The resulting performance per node is illustrated in figure 5.

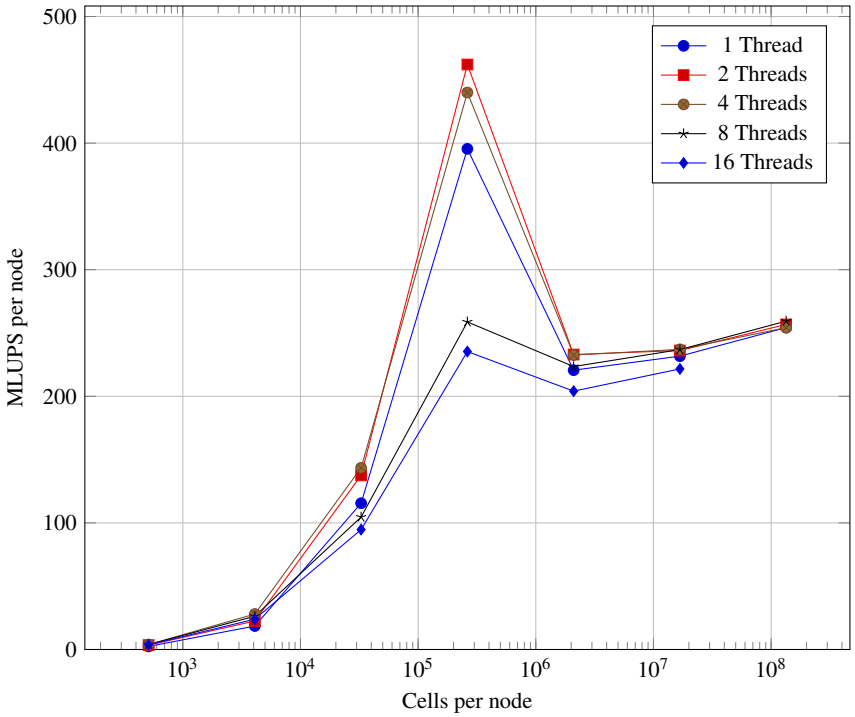

Fig. 3: Performance per node for *D3Q19* on 512 nodes, utilizing all 65, 536 cores.

This illustration shows that there is a significant performance degradation per node in the region of small problem size that fit into the cache from 450 MLUPS on a single node to 293 MLUPS per node on 2048 nodes. Due to the fast computation without accessing memory outside the caches, the necessary communication on the larger node counts increasingly dominates the execution time in this region. Nevertheless, there is still a higher performance observed in this cache region than when accessing the memory for larger problems per node. Without *OpenMP* parallelism the performance drops further down to 255 MLUPS per node. In the region with memory access, however, the performance degradation is relatively small dropping from 190 MLUPS on a single node to 166 MLUPS per node on 512 nodes for 16, 777, 216 cells per node.

As observed above, the *OpenMP* parallelism does not have much of an influence for problem sizes per node and for other numbers of threads a similar behavior is observed. And the performance for 16, 777, 216 cells per node on 512 nodes does not vary much with the number of threads per process. This is summarized in table 1.

Of a little more interest in this respect is the strong scaling, where the problem size per node decreases with growing numbers of nodes. Figure 7 shows the strong scaling efficiency for the various number of threads per process. Aside from the rapid

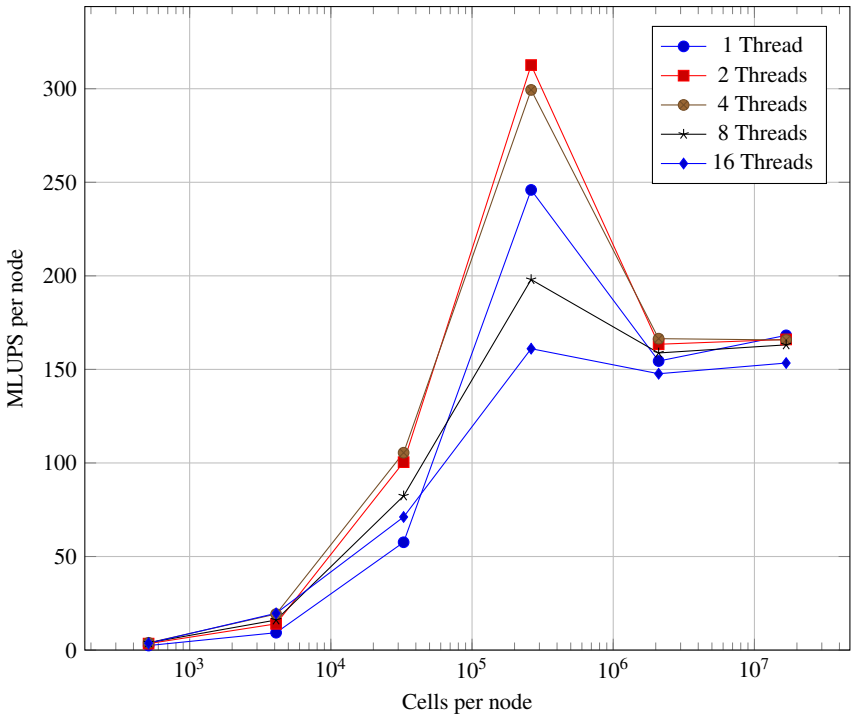

Fig. 4: Performance per node for *D3Q27* on 512 nodes, utilizing all 65, 536 cores.

Table 1: Performance per node on 512 nodes for 16, 777, 216 cells per node

Threads per process	MLUPS per node
1	168
2	166
4	166
8	163
16	153

decline in the parallel efficiency beyond the peak in the cache region we see that the use of *OpenMP* threads here allows for a better scaling to small problems per node, with 4 threads per process, matching the *CoreCompleX* yielding the highest parallel efficiency on 512 nodes. Note, that this graph is somewhat truncated due to the few cells fitted on a single node, though more than 6 times as many could fit into the memory.

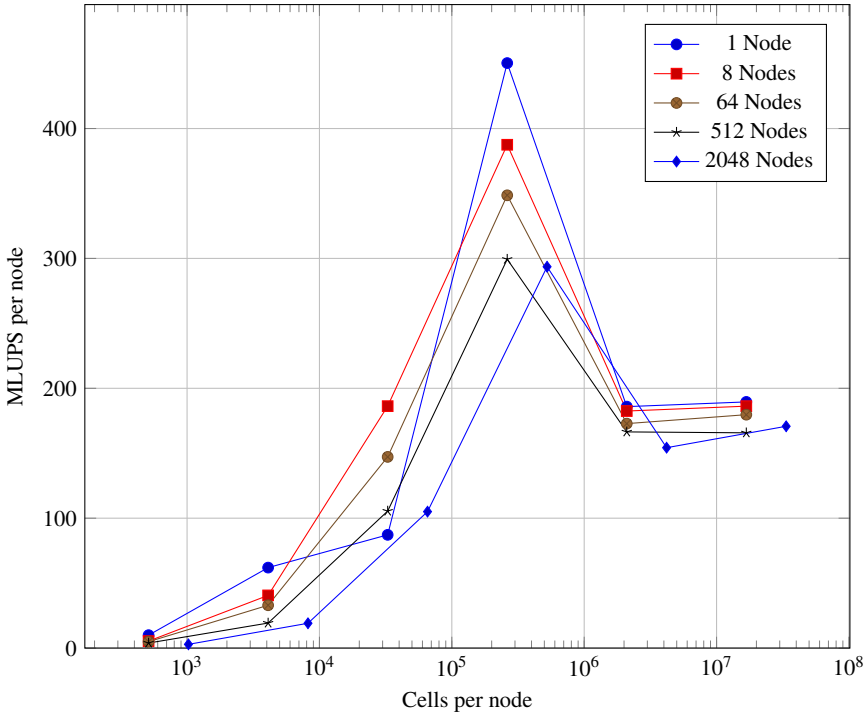

Fig. 5: Performance per node for *D3Q27* with 4 *OpenMP* threads per *MPI* process.

5 Conclusion

We have presented a basic analysis of the performance behavior of *Musubi* on the *HLRS* computing system *Hawk*. It reveals that up to 4 *OpenMP* threads per process can be used interchangeably with *MPI* parallelism and can slightly improve the performance in strong scaling for very small problems per node. This number of threads corresponds to the *CoreCompleX* of the *AMD EPYC 7742* processors, which groups 4 physical cores that share a L3 cache together. The largest problem computed in this analysis contained $68,719,476,736$ cells and was computed on $262,144$ cores.

Acknowledgements We thank the High-Performance Computing Center Stuttgart (*HLRS*) for the computing time on *Hawk* to perform the presented analysis.

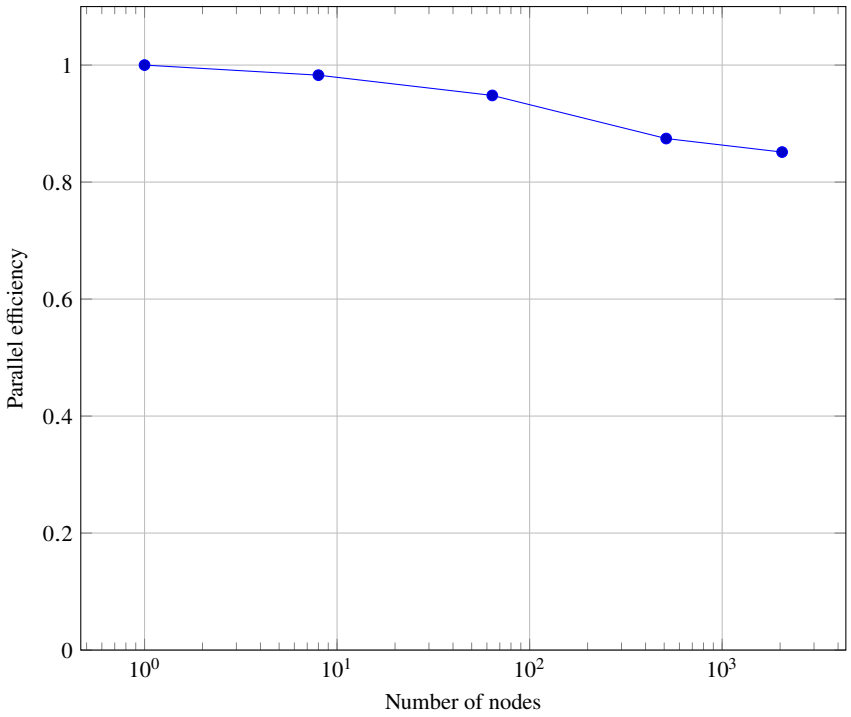

Fig. 6: Weak scaling parallel efficiency for *D3Q27* with 4 *OpenMP* threads per *MPI* process and 16, 777, 216 cells per node (linear interpolated for 2048 nodes.

References

1. P.L. Bhatnagar, E.P. Gross and M. Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.* **94**(3), 511–525, (1954).
2. D. 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, pp. 81–92, Springer Berlin (2012).
3. 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. *J. Comp. Sci.* **5**(5), 784–794, (2014).
4. H. Klimach. Musubi Mercurial Repository. https://osdn.net/projects/apes/scm/hg/musubi/. Accessed 2022-03-25.
5. G.M. Morton. A computer oriented geodetic data base and a new technique in flie sequencing. Technical report, IBM Ltd. (1966).
6. MPI: A Message Passing Interface 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. Accessed 2022-03-25.
7. NASA: HECC Knowlegebase: AMD Rome Processors. https://www.nas.nasa.gov/hecc/support/kb/amd-rome-processors_658.html. Accessed 2022-03-25.

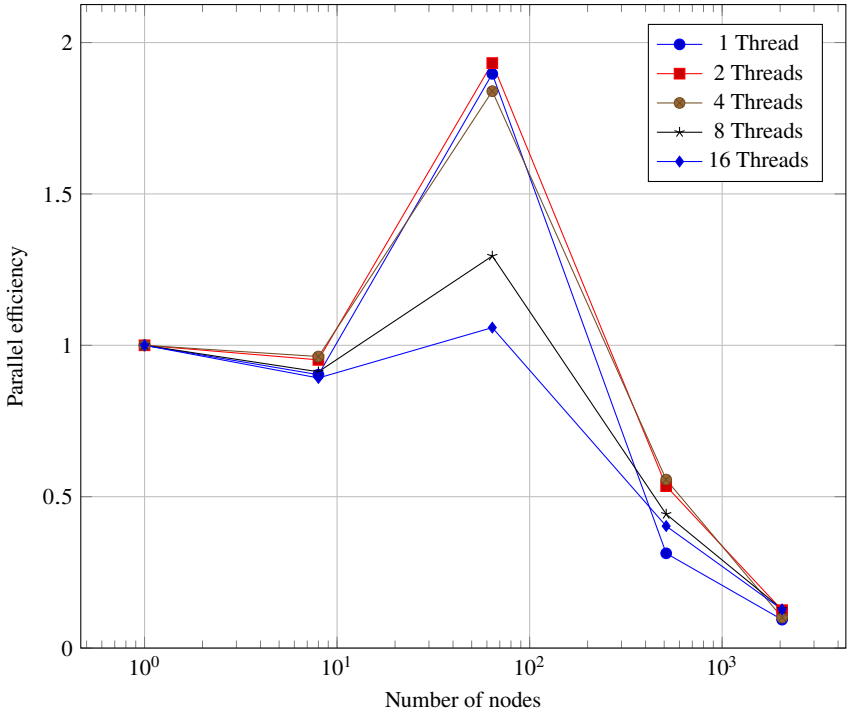

Fig. 7: Strong scaling parallel efficiency for *D3Q27* and 16, 777, 216 cells in total.

8. OpenMP Application Programming Interface 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Accessed 2022-03-25.
9. S. Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Oxford Univ. Press (2001).