hpc.bw Benchmark Report 2022-2024

Hauke Preuß, Guillaume De Nayer, Amartya Das Sharma,
Piet Jarmatz, Willi Leinen, Vahid Jafari, Ruben Horn,
Michael Breuer, Andreas Fink, Philipp Neumann

December 16, 2024

Abstract

In the scope of the dtec.bw project hpc.bw, innovative HPC hardware resources were procured to investigate their performance for HSU-relevant compute-intensive software. Benchmarks for different software packages were conducted, and respective results are reported and documented in the following, considering the Intel Xeon architecture used in the HPC cluster HSUper, AMD EPYC 7763 and ARM FX700.

1 Introduction

High Performance Computing (HPC) is a rapidly evolving field in terms of HPC hardware. Typical life cycles of full HPC clusters lie in the range of 5-10 years. It is crucial to keep track of current hardware developments to ensure optimal efficiency of newly procured clusters in terms of time-to-solution and, also, energy.

In this paper, various software packages that are relevant for research at the Helmut-Schmidt-Universität/Universität der Bundeswehr Hamburg (HSU) are benchmarked on three platforms: the current HPC cluster HSUper (Subsection 1.1), an AMD EPYC 7763 cluster (Subsection 1.2) and an ARM FX700 cluster (Subsection 1.3). All pieces of hardware were procured and investigated under the umbrella of the dtec.bw-funded project hpc.bw, which strives to set up an HPC competence platform at the universities of the federal armed forces. Scalability and performance analysis are provided to suggest optimal settings and provide insight which hardware fits which problem best.

The following software packages were considered:

- Section 2: LESOCC (computational fluid dynamics (CFD) solver)
- Section 3: ls1 mardyn (molecular dynamics (MD) simulation)
- Section 4: waLBerla (Lattice Boltzmann (LB) fluid flow simulation)
- Section 5: MaMiCo (molecular-continuum simulation coupling)
- Section 6: Gurobi (mathematical optimization)

Table 1 gives a hardware and power consumption overview of the three platforms mentioned, Table 2 a benchmark result overview. The following subsections describe the different platforms in more detail.

Figure 1 shows the bandwidth (1b) and latency (1a) results of the OSU MPI benchmarks using the GNU C compiler (GCC) 8.5.0 in table 2 for HSUper, AMD testbench and ARM testbench.

Table 1: Available host machines for benchmarks; hardware overview. The power consumption is measured at the Power Distribution Unit (PDU) where η is the fraction of nodes that are busy. The idle power is the load power for $\eta=0$.[1] For the ARM testbed the standby power increases only negligably to 24.30 ± 7.58 W when connecting all 8 nodes. The idle power consumption increases to 1193.16 ± 12.35 W, which is significantly less than proportional to the number of nodes.

	HSUper	AMD Testbed	ARM Testbed
CPU vendor	Intel	AMD	Fujitsu
CPU model	Xeon P8360Y	EPYC 7763	A64FX
Instruction set	x86_64	x86_64	aarch 64
Base / turbo frequency [GHz]	2.40 / 3.50	2.45 / 3.50	2.00
RAM per node [GB]	256	256	32
RAM bandwidth $[^{GB}/_{s}]$ per node	399.8	204.8	1024
Sockets per node	2	1	1
Cores per socket	36	64	48
Threads per node	144	128	48
L1 per core [KiB]	80	64	128
L2 per core [KiB]	1280	512	682.67
L3 per socket [MiB]	54	256	0
Nodes available	571	4	8
$\mu\pm\sigma$ Standby $\text{Power}_{4\text{ nodes}}$ [W]	57.22 ± 1.24	28.04 ± 1.40	23.33 ± 7.84
μ Stress $\text{Power}_{4\text{ nodes}}(\eta)$ [W]	$253.43\times 4\eta+1459.84$	$270.64\times 4\eta+514.26$	896.55
μ Stream $\text{Power}_{4\text{ nodes}}(\eta)$ [W]	$266.51\times 4\eta+1465.55$	$247.06\times 4\eta+514.88$	$31.67\times 4\eta+897.92$

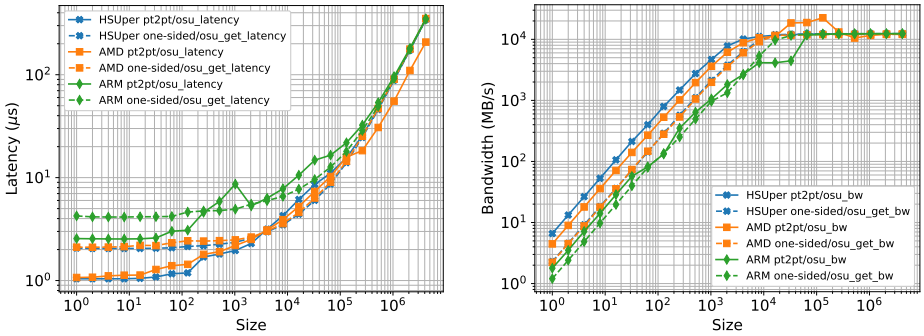
[1] See Appendix A for more information on how the power consumption was measured and a plot of all results in Figure 31.

[2] The optimizations for each platform are described in Appendix B.

[3] The OFED perftest package was used with default settings, namely ib_send_lat, ib_read_lat, ib_write_lat, ib_send_bw, ib_read_bw, and ib_write_bw. The average value of the best run out of three is provided in Table 2.

[4] Information about the compilation of the OSU Micro-Benchmarks package is available in Appendix C.

[5] Appendix D provides information about the configurations of likwid-bench.



(a) Latency results.

(b) Bandwidth results.

Figure 1: OSU MPI benchmark results for GCC 8.5.0 (Avg. over 3 samples per size).

Table 2: Available host machines for benchmarks; benchmark result overview. The memory bandwidths were recorded for each function in the optimized[2] stream micro-benchmark [21] revision 5.10. The OFED perftest package[3] was used for Infiniband tests. OSU MPI tests were performed with the Point-to-Point MPI Benchmarks (P2P) and One-sided MPI Benchmarks (1S) from the OSU Micro-Benchmarks 7.4 package[4] using OpenMPI 4.1.x or Intel MPI 2021.6.0 with GCC 8.5.0. For latency, the range of values (problem size of 1 and 4,194,304) is reported, for bandwidth, results are provided for a problem size of 4,194,304. Maximum FLOP rates were measured using the tool likwid-bench[5].

	HSUper	AMD Testbed	ARM Testbed
Stream Copy $[MB/s]$	320540.9	152564.6	562993.8
Stream Scale $[MB/s]$	327041.0	152372.1	554802.1
Stream Add $[MB/s]$	324034.9	154067.5	633580.7
Stream Triad $[MB/s]$	327494.8	154053.8	630920.1
Infiniband Latency Send / Write / Read $[\mu s]$	1.52 / 1.49 / 2.87	1.19 / 1.24 / 2.24	2.48 / 2.23 / 3.52
Infiniband Bandwidth Send / Write / Read $[MB/s]$	11772.88 / 11795.36 / 11781.01	11270.89 / 11470.71 / 11276.21	11786.69 / 11811.79 / 11796.42
OSU MPI Latency P2P / 1S $[\mu s]$	1.04-344.52 / 2.07-343.79	1.07-207.51 / 2.11-351.25	2.56-350.07 / 4.24-345.53
OSU MPI Bandwidth P2P / 1S $[MB/s]$	12343.7 / 12344.52	12209.45 / 12069.46	12380.4 / 12387.5
likwid-bench (peakflops) $[TFlop/s]$	5.926	2.648	3.049

1.1 HSUper

In December 2022, the new Intel Icelake based HPC system HSUper[6] was inaugurated. It consists of 581 nodes in total: 571 compute nodes with 256 GB RAM per node, 5 nodes with 1 TB RAM per node, and 5 nodes with 2 NVIDIA A100 GPUs per node. Each node contains two sockets equipped with Intel Xeon Platinum 8360Y (36 cores, 2.4GHz, Max Turbo Frequency: 3.5GHz) processors, hence 72 physical cores or 144 virtual CPUs with hyperthreading. The respective caches are structured as follows:

- Level 1 cache size: 36×32 KiB instruction caches, 36×48 KiB data caches per socket
- Level 2 cache size: 36×1.25 MiB caches per socket
- Level 3 cache size: 54 MiB shared cache per socket

Physical cores have their own L1 and L2 caches, and 36 physical cores share an L3 cache. The memory is provided by 16×16 GB or 64 GB DDR4 RDIMM 3200 MHz ECC-registered modules.

Network connectivity is provided by an InfiniBand HDR-based, 100Gb/s-capable, non-blocking fat tree topology.

Energy is provided by 2+1 power supply units with 2100 W per 4 node chassis. The standby power consumption of these 4 nodes is measured to be 57.22 W and the idle power consumption is 1465.55 W. Running on 2 nodes, the optimized stream benchmark[7] consumed 1992,86 W on average and achieved 327494.8 $^{MB}/_{s}$ in the triad function.

Please note: The Infiniband tests were performed while HSUper was under high overall system load.

Figure 2 shows the bandwidth (2b) and latency (2a) of the OSU MPI benchmarks in table 2 with different compilers for HSUper. Notably, when using the same MPI library during execution, compiling with Intel's new oneAPI compilers demonstrates similar performance to traditional Intel compiler builds.

HSUper's TOP500 listing can be accessed online[8]. Linpack performance (Rmax) of 2.12 $^{PFlop}/_{s}$ was achieved. For this report, a single-node Intel-optimized MP Linpack benchmark was executed with a peak performance of 3.95 $^{TFlop}/_{s}$.[9]

1.2 AMD Testbed

The AMD EPYC-based testbed consists of four nodes in total, all equipped with one socket and 256 GB RAM per node, hence 64 physical cores or 128 virtual CPUs. Each node contains one socket equipped with an AMD EPYC 7763 (64 cores, 2.45 GHz, Max. Boost Clock: 3.5GHz) processor with the following cache configuration:

- Level 1 cache size: 64×32 KiB instruction caches, 64×32 KiB data caches

[6] The official HSUper documentation can be found at https://portal.hpc.hsu-hh.de/documentation/hsuper/.

[7] The optimizations for the stream benchmark on HSUper are described in Appendix B.1.

[8] https://top500.org/system/180076/

[9] Details regarding the MP Linpack benchmark on HSUper can be found in Appendix D.1.

- Level 2 cache size: 64×512 KiB caches
- Level 3 cache size: 256 MiB shared cache

Each physical core has its own L1 and L2 cache, and 8 cores share a 32 MiB L3 cache, forming a Core Chiplet Die (CCD). The memory is provided by 8×32 GB DDR4 RDIMM 3200 MHz ECC-registered modules. Network connectivity is provided by an InfiniBand EDR-based, 100Gb/s-capable, non-blocking topology. Energy is provided by 1+1 power supply units with 2000 W per 4 node chassis. The standby power consumption of these 4 nodes is measured to be 28.04 W just under half of the amount consumed by HSUper. When all nodes are idle, the power consumption is 514.26 W, only 35.23 % of HSUper. All 4 nodes running the optimized stream benchmark[10] consumed 1503.12 W. The triad function achieved 154053.8 $^{MB}/_{s}$ or 47.04 % of HSUper.

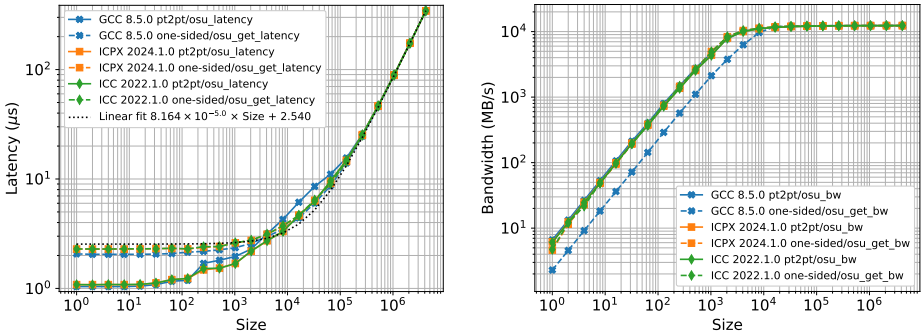
1.3 ARM Testbed

The Fujitsu PRIMEHPC FX700 ARM-based testbed consists of 8 nodes, all equipped with one A64FX CPU and 32 GB RAM per node, hence 48 physical cores and threads are available. The Fujitsu A64FX (48 cores, 2 GHz) processor uses the Arm v8.2 + SVE instruction set architecture, together with the following cache hierarchy:

- Level 1 cache size: 48×64 KiB instruction caches, 48×64 KiB data caches
- Level 2 cache size: 32 MiB (8 MiB×4)

Each core has its own L1 cache, and 12 cores share an 8 MiB L2 cache. The memory is provided by 4×8 GB HBM2 stacks.

[10] The optimizations for the stream benchmark on the AMD testbed are described in Appendix B.2.



(a) Latency results. (The One-sided variant has a higher latency compared to the Point-to-Point one for benchmarks with a size up to 10^3. All results can be characterized by the same linear model.)

(b) Bandwidth results. (Only the One-sided variant built with GCC has a significantly lower bandwidth for benchmarks with a size up to 10^4.)

Figure 2: OSU MPI benchmark for HSUper (Avg. over 3 samples per size).

Network connectivity is provided by an InfiniBand EDR-based, 100Gb/s-capable, non-blocking topology.

Energy is provided by 2+1 power supply units with 2200W per 8 node chassis. The maximum power consumption for the complete system is 2,723 W.[11] With 23.33 W and 24.30 W, the standby power consumption is rather similar with both 4 and 8 nodes in the chassis. The standard deviation, however, is quite high with over 7.5 W compared to HSUper and the AMD testbed for which it is below 1.5 W. With 4 idle nodes of 4 total nodes in the chassis, the power consumption is 897.92 W or 61.51 % of HSUper. With 8 nodes per chassis, this rises to 1193.16 W, which is only 40.87 % of HSUper per node, but still more than the AMD testbed. With 4 out of 4 nodes in the chassis running the optimized stream benchmark[12], the power consumption was 1024,6 W or only 68.16 % of the AMD testbed under the same load. The triad function achieved 630920.1 $^{MB}/_{s}$ or 1.93 times that of HSUper and 4.10 times that of the AMD testbed. While the power consumption is almost constant on the ARM testbed regardless of the load, we observe significant peaks in the power consumption compared to the other two systems (Figure 30). The frequency of these outliers is significantly higher under load compared to idle.

The PRIMEHPC FX700 system exhibits higher latency over Infiniband, as communicated via Fujitsu, primarily due to the presence of a single shared PCI controller on the die. This design induces an average hardware latency of 2 μs – a 1 μs increase from typical x86 systems.

2 LESOCC

2.1 Introduction

$\mathcal{LESOCC}$ ($\mathcal{L}$arge $\mathcal{E}$ddy $\mathcal{S}$imulation $\mathcal{O}$n $\mathcal{C}$urvilinear $\mathcal{C}$o-ordinates) [2, 3, 4, 5] is a 3–D finite–volume code for computational fluid dynamics (CFD) on arbitrary non–orthogonal and non–staggered block–structured grids. It was developed for the prediction of complex turbulent flows by the large-eddy simulation (LES) technique or by direct numerical simulations (DNS). $\mathcal{LESOCC}$ is fully parallelized based on classical domain decomposition with explicit message passing (MPI) between the parallel blocks. For this purpose, the geometric block structure is mapped to a parallel block structure.

2.2 Challenges

The performance tests with $\mathcal{LESOCC}$ are performed on all three aforementioned HPC systems. On HSUper, the CFD code is compiled with the Intel compiler (module: compiler/2022.1.0). Furthermore, the Intel MPI version (module: mpi/2021.6.0) is loaded. On the AMD cluster, the GNU Fortran (GCC) 8.5.0 compiler is used. Additionally, the Open MPI software openmpi-4.1.2rc2 is applied. On the ARM cluster, the FUJITSU compiler `frt` (4.8.0 20220202) is used together with the Open MPI software openmpi-4.1.2a1.

[11] https://www.fujitsu.com/downloads/SUPER/manual/C120-0089-11EN.pdf, p. 41

[12] The optimizations for the stream benchmark on the ARM testbed are described in Appendix B.3.

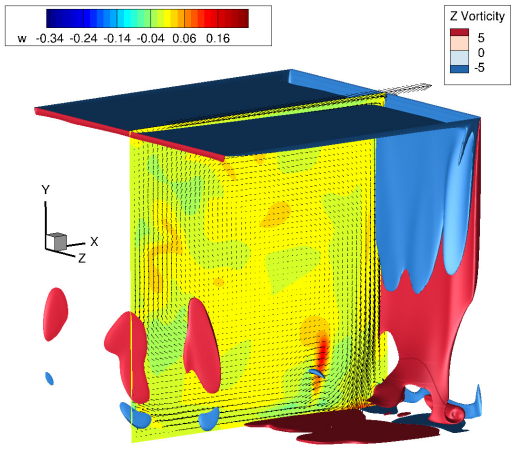


Figure 3: Lid-driven cavity flow at Re = 12,000. Arbitrarily chosen snapshot of the flow showing iso-surfaces of the vorticity component in z-direction and a slice through the center of the cavity depicting the velocity component in z-direction.

Since the grid resolution of the test case is very fine (see Section 2.3), the allocation of very large arrays is required. One issue which has to be taken into account is that in cases in which arrays possess a size larger than 2 GB ($= 2^{31}-1$), the integer values to address the entities of the arrays have to be defined as `Integer*8`, i.e., 64 Bit. For this purpose on HSUper, the compiler flag `-integer-size 64` is used in case of the Intel compiler, while for the application of Intel MPI `mpiifort -i8 -integer-size 64` is required for the compilation and `mpiifort -ilp64 -integer-size 64` for linking. On the AMD cluster, the installed MPI version does not support `Integer*8`, restricting the possible test configurations. Additionally, the size of the machine is restricting available configurations. Similar difficulties arise in the case of the ARM cluster, where the available RAM per node and the number of nodes pose restrictions.

2.3 Benchmark Case

As a benchmark problem, the lid-driven cavity flow scenario is chosen. Here, the simplest case of a cubic cavity with side length ℓ is chosen, i.e. the computational domain is of dimensionless extension of $1\times1\times1$. All walls except the top wall are no-slip walls at rest ($u=0$). The top wall, denoted the *lid*, moves with a constant velocity u_{lid} in positive x-direction, thus driving the fluid flow.

The Reynolds number of this configuration is defined by $\text{Re} = u_{\text{lid}}\ell/\nu$, where ν is the kinematic viscosity of the incompressible, Newtonian fluid. The Reynolds number is set to Re = 12,000, for which a turbulent flow in the cavity is expected [17, 18]. Direct numerical simulations are carried out. Figure 3 shows a snapshot of the three-dimensional, turbulent flow.

The cube is discretized by an equidistant grid with 512 control volumes (CVs)

Table 3: Overview on the different configurations considered on HSUper. Here, '# B' stands for the number of blocks, while the number of nodes and the number of cores per node are abbreviated by '# N' and '# C'.

Case	# B	**Config. 1**		**Config. 2**	
		# N	# C	# N	# C
0	1	1	1		
1	2	1	2		
2	4	1	4		
3	8	1	8		
4	16	1	16		
5	32	2	16	1	32
6	64	4	16	2	32
7	128	8	16	4	32
8	256	16	16	8	32
9	512	32	16	16	32
10	1024	64	16	32	32
11	2048	128	16	64	32
12	4096	256	16	128	32

Table 4: Overview on the different configurations considered on the AMD cluster. The same abbreviations as in Table 3 are used.

Case	# B	**Config. 1**		**Config. 2**		**Config. 3**	
		# N	# C	# N	# C	# N	# C
0	1	1	1				
3	8	1	8				
4	16	1	16				
5	32	2	16	1	32		
6	64	4	16	2	32	1	64
7	128			4	32	2	64
8	256					4	64

in each direction. Thus, $512^3 \approx 134.22$ million CVs are applied. The dimensionless grid spacing is $\Delta x = \Delta y = \Delta z = 1/512 = 1.95 \times 10^{-3}$. The dimensionless time step size is set to 8×10^{-4} leading to a CFL number of about 0.4.

Given the domain decomposition approach of $\mathcal{LESOCC}$ based on MPI, strong scaling tests can be performed by successively splitting up the blocks into smaller ones. Thus, starting from 1 block containing the whole grid, one can end up with $16^3 = 4096$ blocks by splitting the grid into 16 parts in each direction. Note that this requires sufficient main memory and is thus only possible on HSUper and to some extent the AMD cluster. The configurations considered for the strong scaling tests on each of the three systems can be found in Tables 3 to 5.

On HSUper, all 13 cases from Table 3 ranging from one single block up to a domain decomposition of 4096 blocks are taken into account. However, based on

Table 5: Overview on the different configurations considered on the ARM cluster. The same abbreviations as in Table 3 are used.

Case	#Blocks	**Config.**	
		#Nodes	#Cores
3	8	8	1
4	16	8	2
5	32	8	4
6	64	8	8
7	128	8	16
8	256	8	32

previous experiences [25], the memory bandwidth of the nodes yields a decreasing performance, if too many cores per node are being used. To investigate this issue, two scenarios are tested: one with up to 16 cores per node and a second one with 32 cores per node. On the AMD cluster, only a restricted number of cases is considered, cf. Table 4. Again, different core-per-node distributions are investigated. Similarly, due to memory limitations, the possible configurations on the ARM cluster are limited to the ones depicted in Table 5, cf. Section 2.2.

2.4 Results

The results of the scaling tests on HSUper and on the AMD cluster are depicted in Figure 4. Here the average wall-clock time for a certain number of time steps (ranging from about 10^3 to 10^5 depending on the problem size) was measured. Each case is at least repeated three times to improve the statistics. In the double-logarithmic diagram, a nearly linear decrease of the wall-clock time can be observed on HSUper for the case with only 16 cores per node. Using 32 cores per node obviously deteriorates the results by a factor of about 1.5 in the wall-clock times of cases 5 to 12. This increase in run time between the 32- and the 16-cores-per-node configurations is attributed to the memory bandwidth of the nodes, which is already saturated at lower core-per-node counts. This is consistent with a previous study [25]. With an increasing number of cores per node an over-saturation sets in, leading to perceptibly longer run times.

On the AMD cluster, the predictions on a single core need about the same wall-clock time as on HSUper. That changes for 8 and more cores. For the configuration with 16 cores per node, a factor of about 2.5 to 2.7 is found between the measured wall-clock times on the AMD and the HSUper cluster. A slightly lower factor of about 2.2 is observed for the configuration with 32 cores per node. Consequently, the AMD cluster is about this factor slower than HSUper. However, it has to be noted that the code on the AMD cluster was not compiled with the AMD-specific compiler but with the GNU compiler.

A rise in the required wall-clock time between configuration 1 with 16 cores per node and configuration 2 with 32 cores per node is visible in Figure 4 for the AMD cluster. A factor of about 1.3 is found, which is lower than the factor of about 1.5 observed on HSUper. However, the factor between configuration 2 with 32 cores per

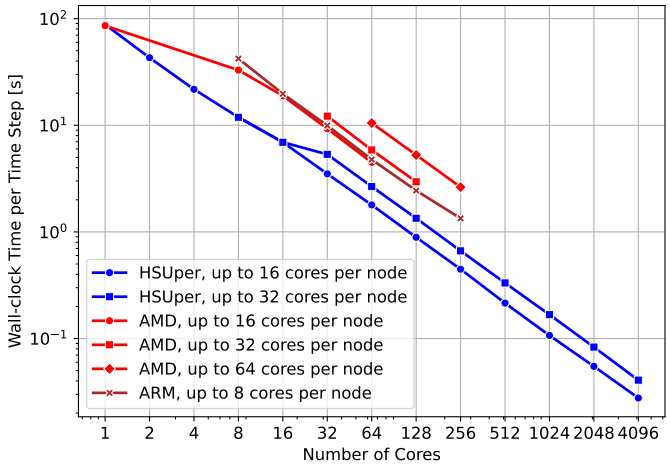


Figure 4: Strong scaling tests for $\mathcal{LESOCC}$ on the three testbeds HSUper, the AMD cluster and the ARM cluster. Wall-clock time per time step for various core-per-node configurations.

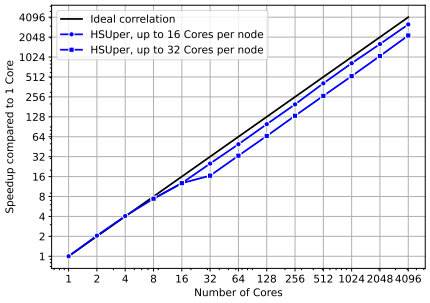
node and configuration 3 with 64 cores per node is perceptibly higher (about 1.8).

About a factor between 2.7 and 3.6 is observed between the wall-clock times on the ARM cluster and HSUper with 16 cores per node. Note, however, that these statements should be considered with caution since in case of the ARM cluster, the number of nodes (8) is fixed, whereas on HSUper the number of cores per node (16) is fixed while the number of nodes is varying. Since on the ARM system no other runs were possible, a comparison under the same circumstances was not feasible.

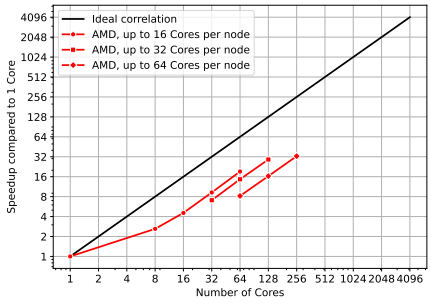
Figure 5 depicts the speedup obtained during the strong scaling tests for $\mathcal{LESOCC}$ on the three testbeds. For HSUper (Figure 5a) and the AMD cluster (Figure 5b), the run times are scaled with the run time of the prediction on a single core (i.e. case 0 in Tables 3 and 4, respectively). For the ARM cluster shown in Figure 5c, the run times are scaled by the run time of case 3 in Table 5, i.e., with 8 cores. In all three cases, the expected ideal correlations are included as a comparison.

In case of HSUper, a nearly ideal scaling is observed in Figure 5a up to 8 cores/blocks. Owing to the saturation of the memory bandwidth an offset of the speedup curve starting at 16 cores is visible for the case with 16 cores per node. Nevertheless, a linear scaling behavior up to 4094 cores is found which is still close to the ideal one. Due to the factor of about 1.5 in the run times between the 16- and the 32 cores per node case, a further offset of the speedup curve is found for the latter case. Nevertheless, the slopes of the speedup curves are the same for both configurations.

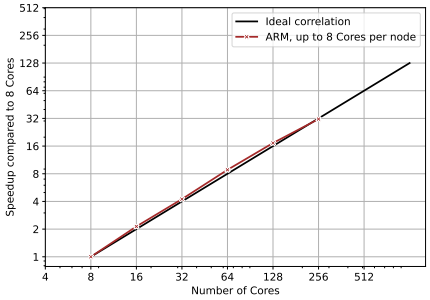
On the AMD cluster (Figure 5b), the configuration with 16 cores per node shows a linear scaling behavior when the number of blocks is 16 or above, while the performance significantly deteriorates between 1 and 8 cores. The other configurations with 32 and 64 cores per node lead to the same slope of the speedup curve, but again an offset is observed which is due to factor between the run times mentioned above.



(a) HSUper.



(b) AMD cluster.



(c) ARM cluster.

Figure 5: Speedup of the strong scaling tests for $\mathcal{LESOCC}$ on the three testbeds HSUper, the AMD cluster and the ARM cluster.

The speedup data for the ARM cluster are depicted in Figure 5c. Note again that due to the reason mentioned above, all run times are scaled with the run time of the case with 8 cores. Surprisingly, for the cases with 16, 32, 64 and 128 cores, the speedup is slightly higher than the ideal values. However, one has to take the special situation of this scaling test into account. Besides, for the 8-core case, 8 nodes are applied. Thus, the entire communication has to use the network between the nodes[13]. For the cases with more cores it means that 2, 4, 8 or 16 cores per node are used. Thus, at least partially the communication between the blocks can take place directly on the node. Obviously, that is faster than between the nodes, which leads to this super-linear speedup. For the last case with 8 nodes and 32 cores per node, a saturation sets in, probably due to the memory bandwidth of the nodes as observed for the other clusters.

2.5 Conclusion

Strong scaling tests were carried out on three different HPC systems for a finite-volume CFD code parallelized by domain decomposition with explicit message passing

[13]Note that a direct comparison on HSUper with either 8 nodes (1 core per node) versus 1 node (8 cores per node) has confirmed that the latter is the faster variant.

based on MPI. A nearly ideal scaling was found for the case when only 16 cores per node (or less) are used on HSUper. In this case a total efficiency of 77 % is reached on 4096 cores. With an increasing number of cores per node, a perceptible deterioration of the speedup is observed which is attributed to the saturation of memory bandwidth of the nodes. Based on this configuration a total efficiency of about 53 % is achieved on 4096 cores. Similar trends are also found on the AMD cluster. However, except for the single-core case, the system is more than a factor of 2 slower than HSUper. Due to the restricted memory of the ARM cluster a completely fair comparison with the other systems is not possible. Nevertheless, it is feasible to say that HSUper is currently the fastest machine available for CFD calculations.

3 ls1 mardyn

3.1 Introduction

ls1 mardyn is a flexible and scalable MD program capable of simulating small rigid molecules within ensembles of fixed sizes at extreme scale [30]. The code is highly modular and can be quickly adapted to any unique use case by implementing plugins. A salient feature of ls1 mardyn is the capability of load balancing across ranks, with a variety of load balancing libraries being supported. Its capabilities were showcased in the once largest MD simulation, which simulated over twenty trillion particles [34].

Over the last years, ls1 mardyn has been extended to use the particle container library AutoPas [10]. AutoPas uses auto-tuning to select combinations of MD algorithms, data structures and shared-memory parallelization schemes at node level at run time to continuously monitor and improve particle simulation performance. This was shown to yield significant speedup in several applications [32]. AutoPas was also extended to utilize SVE vectorization for better performance on ARM hardware, as part of a bachelor's thesis. The details of this work can be found in [7].

In an earlier project, several scenarios were simulated on the supercomputer Fugaku[14]. A subset of these experiments were chosen as a benchmark and replicated on HSUper, and on the AMD and ARM testbeds. Both Fugaku and our ARM testbed feature the A64FX CPU, but Fugaku uses the FX1000 model and the testbed uses the FX700. The FX1000 offers a boost mode with 2.2 GHz clock speed, however this feature was not relevant for the current work.

3.2 Challenges

Since we chose to use the same version of the program as was used to test on the Fugaku system, the challenges from those runs carried over to the present task.

1. The MPI wrappers for compilation had to be specifically mentioned to avoid stalls while compiling the dependencies.

2. Some scenarios were impossible to run to completion because of bugs that were fixed in later commits. In the cases where simulations ran partially, the

[14] https://www.fujitsu.com/global/about/innovation/fugaku/

average time per simulation step was taken, and the time to solution was scaled to correspond with the successful cases. This was done only for the scenario with homogeneous particle distribution, to ensure correctness.

3. The runs were made by checking out git commit 1cc3864. However some bugs were found in the weak scaling runs. Hence, the commit 209229d77 was used for weak scaling tests on AMD, and ca8ec00a was used on ARM due to the presence of a relevant bug fix.

4. No thread pinning instructions were given, since none were given in the Fugaku runs.

All runs were done using both `gcc` and the available native compiler (`aocc` on AMD, `icpx` on Intel and `fcc` on ARM). Optimization level `Ofast` was used for FCC, and `O3` was used in all other cases. There were issues with `srun` on the testbeds, so `mpirun` was used instead. AutoPas had SVE vectorization available on the ARM hardware.

More remarks on compilation process:

- To compile with `aocc` on AMD, the commands `export OMPI_CC=clang && export OMPI_CXX=clang++` were issued, and then compilation was carried out as normal.

- To compile with `icpx` on Intel, passing the flags `"-cc=icx"` as C flag and `"-cxx=icpx"` as C++ flag to cmake was necessary.

- To compile with `fcc` on ARM in clang mode, the cmake flags

  ```
  "-Nclang -std=c17 -stdlib=libc++"
  ```

  as C flags and

  ```
  "-Nclang -std=c++17 -stdlib=libstdc++"
  ```

  as C++ flags were required. The used compilers were `mpifcc` and `mpiFCC`. Afterwards, the `make` command was preceded by a `CC='fcc -Nclang'` and a `CXX='FCC -Nclang'`.

As mentioned earlier, AutoPas has features to leverage SVE vectorization. However the rest of the ls1 codebase was untouched, and hence is invariant across the platforms. The only difference in software was in the AutoPas-specific runs in the ARM system (both the testbed and Fugaku), where the SVE vectorization was explicitly enabled.

3.3 Benchmark Case

For this study, we utilized two benchmark scenarios. A short explanation of these scenarios follow. Both are available on the master branch of the public repository of ls1 mardyn.

Droplet Coalescence Two dense liquid droplets are suspended in vapour, and allowed to interact with each other. The two droplets eventually subsume themselves to form a larger droplet. This scenario provides an inhomogeneous distribution of particles, a phase change boundary, and very high particle count and density. We use a Lennard-Jones truncated-shifted potential with a cutoff radius of 8.479, using Argon atoms (single-site) with mass values and LJ coefficients taken from literature [11].

Spinodal Decomposition The temperature of an equilibrated liquid, which populates a cubic box, is suddenly dropped, causing the liquid to contract and form local inhomogeneities. This scenario gives us a homogeneous starting point with uniformly distributed molecules, evolving to unpredictably shaped regions with varied densities. We use a Lennard-Jones truncated-shifted potential with a cutoff radius of 2.5, using a single-site molecule with mass and LJ coefficients set to unity.

These two scenarios were then used for benchmarking, since data of identical scenarios was available from supercomputer Fugaku. A more in-depth explanation of the scenarios can be found in [32].

1. To evaluate the best layout of ranks $\times$ threads in a MPI $\times$ OpenMP setting, a droplet coalescence scenario with 3 million molecules was run, with all possible permutations of MPI ranks $\times$ OpenMP threads with 4-8 fully saturated nodes (as applicable) on all the testbeds. The simulation was run for 200 steps, and was run twice, once with native linked cells containers provided in ls1 mardyn and once with AutoPas container(s).

2. Using the best layout for each hardware, both weak and strong scaling tests were carried out, using the spinodal decomposition scenario. The strong scaling case comprised of 2 million molecules, and the weak scaling test started at 35000 molecules, doubling with the doubling of node count.

3. The tests were carried out over 10000 simulation time steps on all hardware, except the ARM testbed, where only 2000 steps were carried out. In case the test failed after multiple attempts, the average seconds per time steps over the failed attempts were taken and scaled to match 10000 time steps. All tests were done both with native linked cells of ls1 mardyn and AutoPas. AutoPas had diffusive load balancing enabled.

3.4 Results

In this section we consider the results of the experiments individually, and pick the best performing layouts for each computer for a final comparison. In all diagrams, AP denotes AutoPas, and LC denotes ls mardyn configured to use its native linked cells container to manage the particles.

3.4.1 MPI × OpenMP layout experiments

In Figure 6, we see the results from the simulation experiments. We can see that, in almost all cases, a pure MPI or pure OpenMP approach would be inefficient. Also, the performance does not always match the expectations one could have from the hardware architecture. For example, on HSUper we see the best node-level performance for both linked cells and AutoPas at 2 MPI ranks × 36 OpenMP nodes, and this is expected, since on HSUper every node has 2 sockets with 36 cores per socket. However on AMD, AutoPas achieves the best performance at 4×16, which is not the case for linked cells.

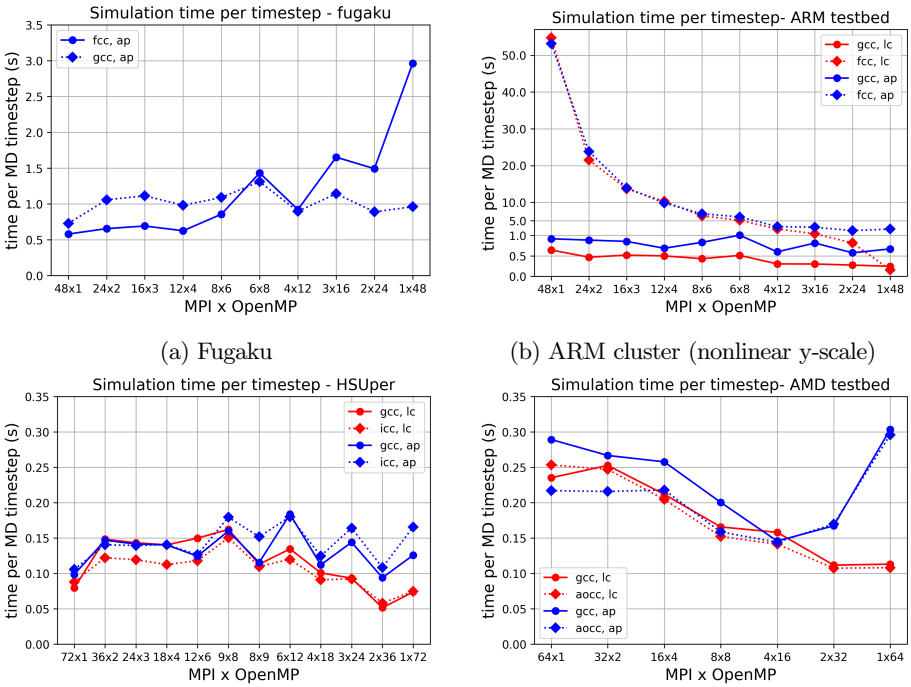
Based on this data, we choose the parameters as follows:

- For Fugaku, the setup with 4 ranks × 12 threads was chosen since 12 cores per node in Fugaku share an L2 cache. The 12×4 setup was also chosen for the strong scaling setup since it provided the best performance.
- For HSUper, the empirically verified fastest setup for both containers (2×36 for both, and 8×9 as the second fastest for AutoPas) were chosen. 8×9 was chosen over 72 × 1 for a more balanced run without large memory requirements.
- For the AMD testbed, similar to HSUper, the fastest setups for linked cells and AutoPas (2×32 and 4×16) were chosen.
- For the ARM cluster, 1×48 and 4×12 were chosen. One would expect the 12×4 layout to perform well since 12 physical cores share an L3 cache, however the experimental results do not confirm this. Hence the fastest layouts were chosen instead.

For the ARM testbed, we see that the performance for the fcc binary is quite bad for the cases with higher MPI ranks. Upon inspecting the output, it was determined that the cause of this slowdown was the excess time taken by the ALL load balancing library in between simulation steps, since the library requires a forced update on the MD container every timestep. This effect was also observed on Fugaku, but it was significantly less pronounced. This occurs on higher MPI rank counts, since this corresponds directly to higher number of MD subdomains, and hence there is a longer time requirement to update all the subdomains. This is a possible area for future code optimization.

3.4.2 Fugaku

The Fugaku supercomputer is not subject of this work, however for comparison's sake we provide the results in Figure 7. We can see that for multi-node cases the native compiler(FCC) performs much better than GCC, and scales better in both strong- and weak- scaling experiments. The empirically confirmed 12×4 layout performs better than the theoretical 4×12 layout.



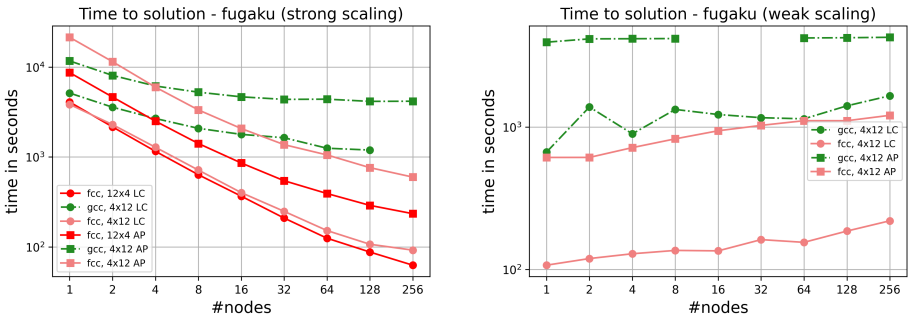
(a) Fugaku

(b) ARM cluster (nonlinear y-scale)

(c) HSUper

(d) AMD cluster

Figure 6: ls1 - Node-level experiments testing the best distribution of MPI ranks and OpenMP threads. Results shown for 8 nodes (4 for AMD). Chosen scenario is droplet coalescence.



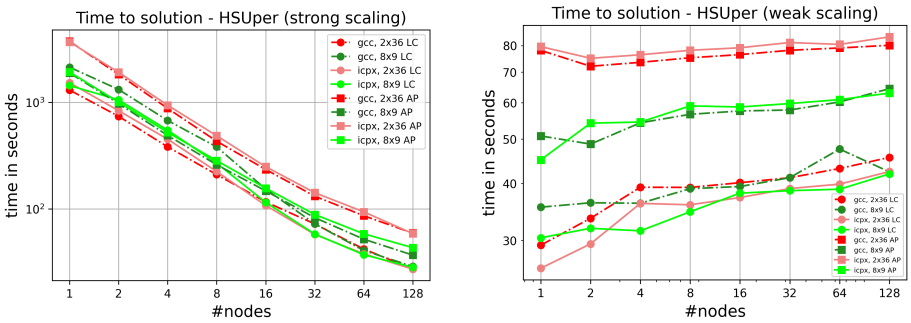
(a) Strong scaling

(b) Weak scaling

Figure 7: ls1 - Results of scaling experiments conducted on the Fugaku supercomputer. The chosen scenario is spinodal decomposition.

3.4.3 HSUper

For HSUper, all layouts seem to scale similarly. Linked cells seem to perform similarly in both MPI×OpenMP layouts, while AutoPas prefers 8×9. The native compiler and GCC provide similar performance in a majority of cases. The results can be seen in Figure 8. Since 36 physical cores on HSUper share an L3 cache, one would expect the 2×36 layout to be the fastest, but AutoPas is clearly faster with the alternate 8×9 layout.



(a) Strong scaling

(b) Weak scaling

Figure 8: ls1 - Results of scaling experiments conducted on HSUper. The chosen scenario is spinodal decomposition.

3.4.4 AMD Testbed

On the AMD testbed (Figure 9) we see good scaling performance, and clear benefits of the 4×16 layout; it performs faster in both experiments, shows better performance with AOCC than GCC, and scales much better in the weak scaling sense. As 8 cores on the ARM testbed share an L3 cache, it would make sense for the case with 4 MPI ranks to perform better than the case with 2 ranks. From our earlier experiment, we had verified that the cases with 4 and 8 MPI ranks perform similarly, except for the GCC+AP case where the 4 rank experiment is faster.

3.4.5 ARM Testbed

The ARM testbed does not perform as well as its peers. The strong scaling is good but the net time to solution is very high, and the weak scaling in the fastest case is quite poor. The native compiler, in many cases, performs worse than GCC. The results are shown in Figure 10.

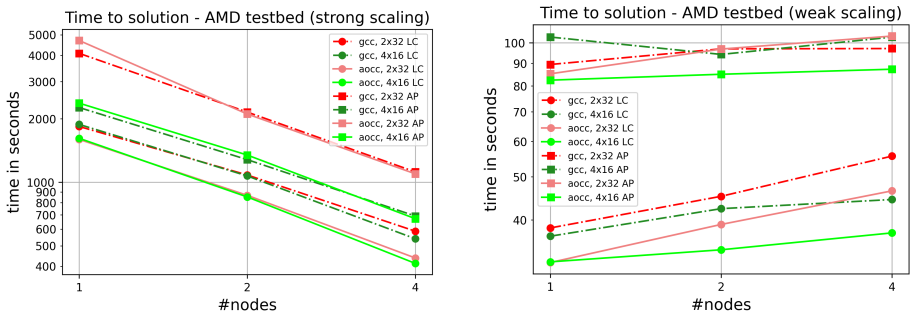
3.4.6 Combined Results: Strong Scaling

In Figure 11 we see combined strong scaling results, taking the fastest performing configurations; Figure 11a shows the GCC performance and Figure 11b shows the

individual native compilers' performances. We can see that the AMD system's performance is admirably close to HSUper's, while the ARM system's performance is closer to Fugaku. A reason for the fall-off in the strong scaling cases at high node counts may be the lower particle density of the chosen scenario, however greater particle densities would make the experiment impossible to complete on the slower hardware.

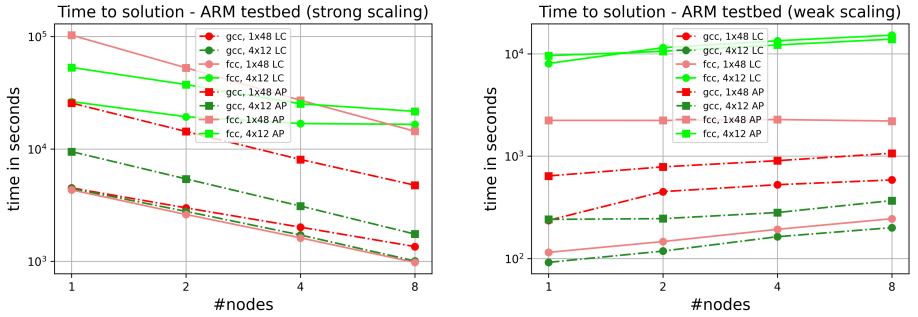
We see a very early flattening for Fugaku when using the GCC compiler. This effect is absent for the native compiler. We presume that this is due to the lack of optimization of the generic GCC compiler on Fugaku, whereas the FCC compiler is optimized for the machine. This behaviour is not observed on the ARM testbed, although it is possible that the effect would manifest at higher node counts.

The AutoPas+FCC+1x48 layout combination for the ARM testbed performs the worst out of the shown options. Looking at the output, the slowdown is observed for all stages of the simulation process, such as the communication, the force computation, the initial domain decomposition etc.



(a) Strong scaling (b) Weak scaling

Figure 9: ls1 - Results of scaling experiments conducted on the AMD testbed. The chosen scenario is spinodal decomposition.



(a) Strong scaling (b) Weak scaling

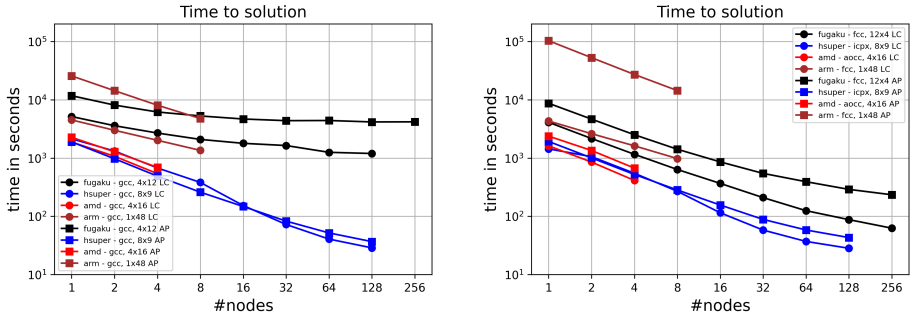
Figure 10: ls1 - Results of scaling experiments conducted on the ARM testbed. The chosen scenario is spinodal decomposition.

		Fugaku	HSUper	AMD	ARM
GCC	LC	1.238	1.877	1.870	1.493
	AP	1.147	1.766	1.808	1.753
Native	LC	1.696	1.788	1.981	1.640
	AP	1.587	1.733	1.883	1.927

Table 6: Average strong scaling factors (ideal is 2)

The same layout with traditional ls1 linked cells performs much better.

Overall, all testbeds show that the AutoPas implementation is slower than vanilla ls1[15], and that the software performs much better on AMD and Intel (AVX) hardware than ARM even though the scaling is acceptable. Hence more work is needed to port the software fully to the ARM system. The average scaling factors (as observed) are given in Table 6.



(a) GCC

(b) Native compiler

Figure 11: ls1 - Combined strong scaling results for the spinodal decomposition scenario.

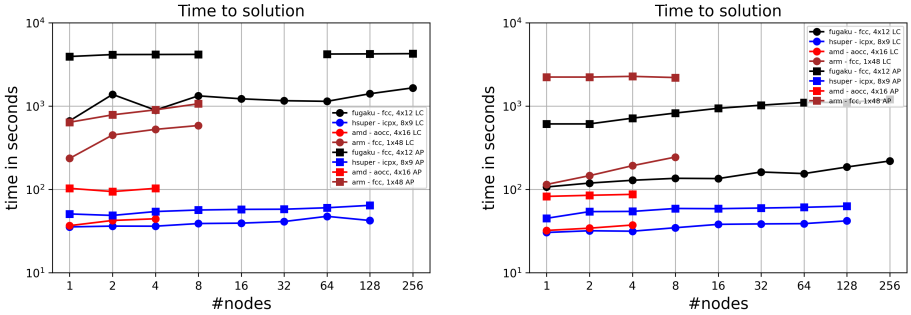
3.4.7 Combined Results: Weak Scaling

We see combined weak scaling performances in Figure 12. Again, we provide the GCC performance in Figure 12a and the native compiler performance in Figure 12b. We see that HSUper provides the best (node-level and scaling) performance while purely looking at time-to-solution, and both AMD and HSUper have reasonable weak scaling. Here too, we see that the AutoPas implementation is slower than vanilla ls1 on all hardware. The average scaling factors (as observed) are given in Table 7.

[15]Recent performance issues have been partially resolved in AutoPas by summer 2024 (after this study), and they yield different performance results.

		Fugaku	HSUper	AMD	ARM
GCC	LC	0.939	0.978	0.910	0.760
	AP	0.986	0.968	1.004	0.844
Native	LC	0.917	0.956	0.928	0.777
	AP	0.920	0.955	0.972	1.005

Table 7: Average weak scaling factors (ideal is 1)



(a) GCC

(b) Native compiler

Figure 12: ls1 - Combined weak scaling results for the spinodal decomposition scenario.

3.5 Conclusion

In this chapter, we discussed the molecular dynamics software ls1 mardyn and AutoPas. We outlined the benchmark scenario, and ran node-level experiments to determine the best MPI ranks × OpenMP thread layout for all the available hardware. The code was compiler with both GCC and the available native compilers, and strong and weak scaling results were presented.

ls1 + AutoPas has been extensively tested and developed on x86 based hardware, and this reflects in the results, where HSUper and the AMD testbed perform similarly to a great extent. More testing and development is needed for ARM based systems to fully leverage the hardware capabilities, and an "out-of-the-box" solution is not enough for comparable performance. As things stand, it is recommended to use x86 based hardware to run ls1, and it is recommended to run a node-level analysis first to verify the best layout for the system, and not to assume that the results will be consistent for all available compilers on the platform.

4 waLBerla

4.1 Introduction

waLBerla [8] is a massively parallel high-performance framework from Erlangen for mesoscopic and CFD flow simulation that can simulate complex geometries, based on the Lattice Boltzmann method (LBM)[16]. It is written in C++ and Python and features hardware specific compute kernels and automatic code generation, based on the Python packages *pystencils* and *lbmpy*, to deliver optimal performance on recent supercomputing CPU and GPU architectures. waLBerla supports adaptive mesh refinement, dynamic load balancing, features a variety of different LBM lattice models (e.g. D2Q9, D3Q19, D3Q27) and collision operators (BGK, TRT, MRT, cumulant model) and compiles with various C++ compilers (such as GCC, Clang, MSVC, Intel compiler). waLBerla supports parallelization via various common technologies, including OpenMP, MPI, CUDA and HIP. There are many extensions of waLBerla available, enabling for instance free surface flows, fluid-structure interaction or phase-field methods.

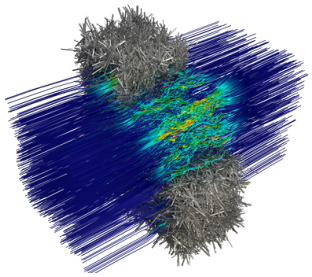


Figure 13: LBM simulation of flow through crystals computed by waLBerla, source https://walberla.net/.

4.2 Challenges

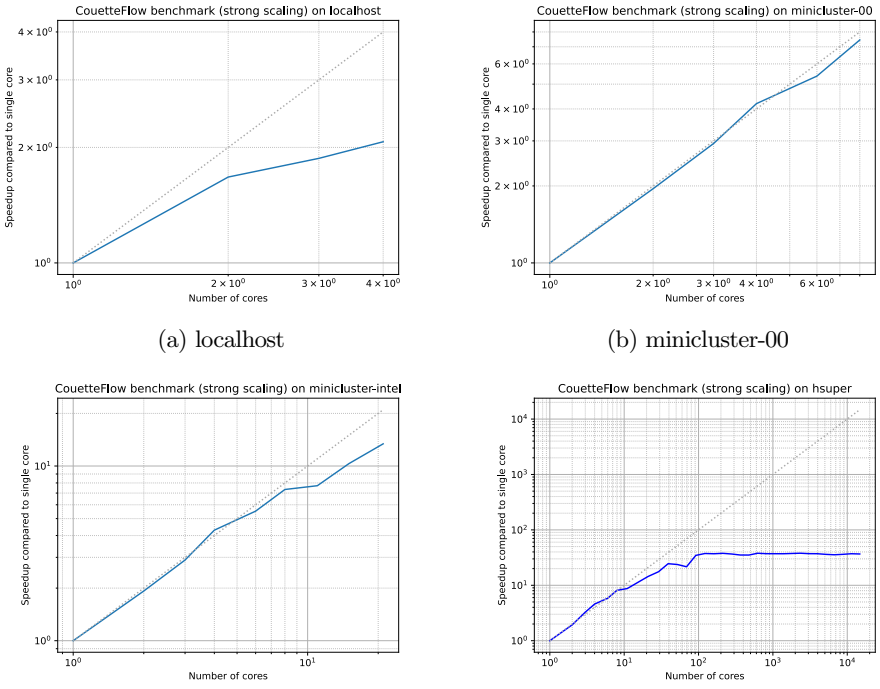
In order to set up and execute the same benchmark scenario with the same waLBerla version on many different host machines, and also to quickly reproduce and verify benchmark results, we have developed an automated benchmark tool, *waLBench*[17]. The tool connects via SSH to the hosts defined in its configuration file. There is an interactive command-line based user interface for the user to control which machines to benchmark and to supervise the status and progress of the benchmarks. The tool is able to connect to all target hosts in parallel and then executes commands on all target machines simultaneously. In order to make the tool more versatile and to design its approaches in a way that is more likely to be applicable to future benchmark platforms without many changes, we have developed and tested it on 6 different host machines - the three main target testbeds of this paper plus three more Intel based machines. More information about the host machines is given in Table 8. Once a connection

[16] https://walberla.net/
[17] https://git.hpc.hsu-hh.de/Piet/walbench (HSU internal)

to a target host is established, waLBench automatically identifies available package managers and installs required software dependencies, without any required previous knowledge about the host or any target machine specific configuration. One of the main challenges for this automated benchmarking process lies is the heterogeneity of the software ecosystems on the target platforms. For instance, in some hosts a suitable version of the GCC compiler is directly available, sometimes the executable is named `gcc`, sometimes `gcc-9`. On some of the hosts, `gcc` can be loaded via a `module` system, sometimes via the `spack` command, sometimes via the `scl` command. On some hosts, the corresponding `scl` package is named `devtoolset-9`, on others the same package is called `gcc-toolset-9`. waLBench tries to abstract and unify these platform specific differences away by automatic detection of the features of the underlying software ecosystem, so that in any case a suitable compiler version is loaded automatically. The versions actually used for the benchmarks are listed in Table 8.

Table 8: Host machines for waLBerla benchmarks, hardware, and software setup

Benchmark machine		minicluster		testbed		
	localh.	00	intel	amd	arm	hsuper
CPU vendor	Intel	Intel	Intel	AMD	Fujitsu	Intel
CPU model	i7-10 510U	Xeon G5222	Xeon G6226	EPYC 7763	A64FX	Xeon P8360Y
Instruction set	x86_64	x86_64	x86_64	x86_64	aarch64	x86_64
Base frequency [GHz]	1.8	3.8	2.7	2.45	2.00	2.40
RAM per node [GB]	16	196	96	256	32	256
RAM bandwith $[GB/s]$ per node	45.8	274.8	274.8	204.8	1024	399.8
Sockets per node	1	2	2	1	1	2
Cores per socket	4	4	12	64	48	36
Threads per node	8	16	24	128	48	144
Max processes/node	4	8	12	64	48	72
L1 per core [KiB]	64	64	64	64	128	80
L2 per core [KiB]	256	1024	1024	512	682.67	1280
L3 per socket [MiB]	8.00	16.5	19.25	256	0	54
Nodes available	1	1	4	4	8	571
Cluster scheduler	None	Slurm	Slurm	Slurm	Slurm	Slurm
Compiler used	GCC 12.3	GCC 11.2	GCC 11.2	GCC 8.5	GCC 8.5 / FCC 4.8.1	GCC 12.1



(a) localhost

(b) minicluster-00

(c) minicluster-intel

(d) HSUper

Figure 14: Speedup of the strong scaling tests with waLBerla CouetteFlow benchmark on the Intel based hosts

Another primary challenge for the benchmarking is the automated installation, i.e. source download and compilation, of waLBerla with a suitable build system configuration on the heterogeneous target platforms. waLBench is designed to avoid unnecessary downloads using `git`, by keeping a local clone of the waLBerla source repository on the benchmark host. In order to guarantee reproducibility of benchmark results, waLBench can checkout a specific waLBerla release version or fixed commit hash rather than the current master branch. If the local repository exists already, for example after a restart due to errors, the waLBerla sources will not be downloaded again. For the `CMake` build system, the following flags are used:

- `BUILD_TESTING=OFF`
- `WALBERLA_BUILD_BENCHMARKS=ON`
- `WALBERLA_OPTIMIZE_FOR_LOCALHOST=ON`
- `CMAKE_BUILD_TYPE=Release`
- `WALBERLA_BUILD_DOC=OFF`

- WALBERLA_BUILD_TUTORIALS=OFF
- WALBERLA_LOGLEVEL=WARNING
- WARNING_PEDANTIC=OFF
- WARNING_DISABLE=ON
- WARNING_ERROR=OF
- WALBERLA_DOUBLE_ACCURACY=ON

With these flags we disable compilation of tests, documentation and tutorials, since we do not need them, but we enable compilation of the predefined benchmarks. Disabling compiler warnings helps to avoid build failures with uncommon compiler versions. The build type `Release` leads to an `-O3` optimization compiler flag and to the `-DNDEBUG` macro definition that disables all debugging related source code. `DOUBLE_ACCURACY` causes the LB method to operate on 64 bit floating point values, which is more common in scientific applications and thus a more realistic benchmark compared to 32 bit precision, besides it matches better to most modern CPU architectures with a 64 bit design. The `OPTIMIZE_FOR_LOCALHOST` setting is especially important for benchmarking, it causes the compiler flags `-march=native` and `-xhost` and `-msve-vector-bits=${SVE_LENGTH}`to be used, if available on the respective target platform and compiler, so that machine specific instruction sets and vectorization with target machine specific vector register length are used for optimization.

A platform specific challenge was given by the ARM testbed, as we had to modify the build system of waLBerla in order to enable compilation with the Fujitsu compiler FCC and thus to leverage the Fujitsu MPI library. Some of our changes have been integrated into waLBerla already[18]. The other changes are primarily related to identification strings of compiler and host system processor. We found that unfortunately for the same compiler binary, the compiler ID changes - depending on the active version of the CMake build system. Thus, either of the two identification strings `FujitsuClang` and `Clang` may be used, so that they have to be considered identical. Also the identification string of the processor architecture is `aarch64` on our test system, while the existing build system was expecting `arm64` instead, making another modification necessary. Besides, we explicitly pass the flags `-mtune=native` and `-mcpu=native` to the FCC compiler on the ARM testbed, since `-march=native` is not available there and would result in compilation failure. Due to these challenges, we have included two separate ARM benchmarking results in this report, (a) for the GCC and OpenMPI software stack that is more easy to configure, and (b) for the FCC and Fujitsu MPI software stack that requires more build system tweaks but might offer enhanced performance.

4.3 Benchmark Case

We directly utilize the predefined `CouetteFlow` benchmark with the unmodified scenario configuration file `CouetteFlow/Test2.dat` from the original waLBerla

[18] https://i10git.cs.fau.de/walberla/walberla/-/merge_requests/615

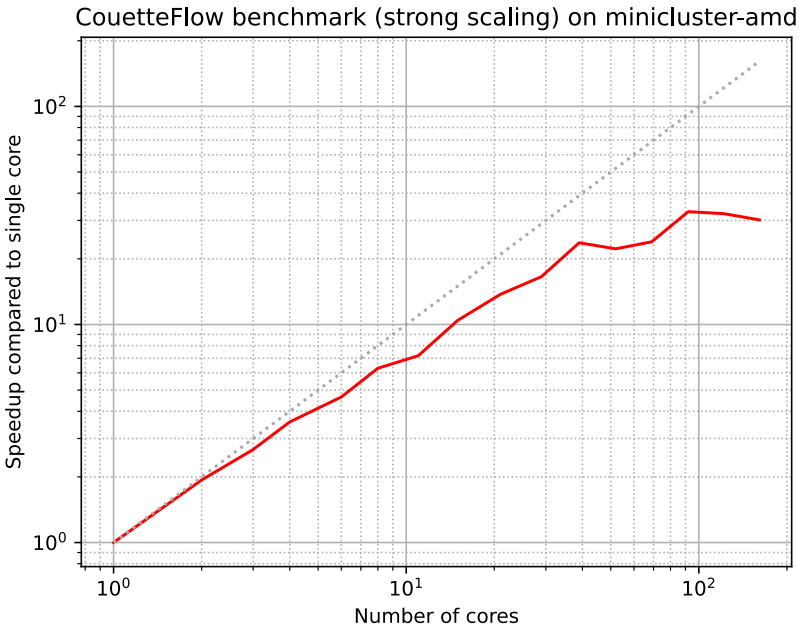
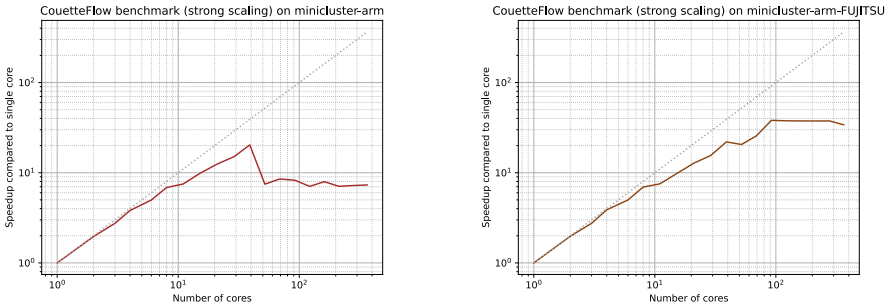


Figure 15: Speedup of the strong scaling test with waLBerla CouetteFlow benchmark on AMD

repository, this ensures the same parameters, the same scenario setup on all clusters. The scenario executes a classic Couette flow test case, i.e. a three-dimensional flow between two parallel moving walls, with periodic boundary conditions. A D3Q19 stencil is used with an SRT collision operator, implemented in a plain C++ kernel, not using the automatic code generation feature. This kernel does not support HIP / CUDA, thus it does not provide GPU support, so this is a CPU-only benchmark. The scenario is using adaptive mesh refinement, 3000 cells on the coarsest mesh level, 3 mesh levels, 73000 cells in total on all grid levels. To make the benchmarks as comparable as possible, the number of cells is fixed and does not change with the number of available CPU cores, i.e. this is a strong scaling case. Dynamic load balancing is used to distribute blocks of fluid cells to the available MPI ranks. OpenMP is not used (threads per process = 1). We run 10000 simulation timesteps, then we determine the average performance during this execution, measured in term of **MLUPS** (million lattice cell updates per second). We compare this MLUPS-performance with the number of MPI processes (ranks) that were launched. If this number of ranks is smaller than or equal to the configured number of processes per node that are used for the benchmark (see Table 8), then they run on a single node. Otherwise, multiple nodes are used.



(a) Compiled with GCC, using OpenMPI (b) Compiled wit FCC, using Fujitsu MPI

Figure 16: Speedup of the strong scaling tests with waLBerla CouetteFlow benchmark on the ARM based host

4.4 Results

Figure 14 compares the benchmark results for the four different Intel based test machines by plotting the speedup over the number of MPI ranks / CPU cores used. The speedup is the relative performance compared to the sequential execution performance, we discuss it first. The absolute performance is discussed later below (see Figure 17). It can be observed in Figure 14a that the scaling on `localhost` is much worse than on the other systems, this can be explained by the different CPU architecture: `localhost` has an `i7-10510U` mobile processor with only two memory channels and a high turbo frequency (4.90 GHz) but a low base frequency (1.80 GHz). This means that the full memory bandwidth can be saturated by only two cores, and that the CPU frequency is reduced when more cores are used. Thus, this CPU benefits much less from parallelization and shows worse scaling behaviour. In contrast to that, the other CPUs are server processors that are designed for HPC applications and scalability, they do not show this behaviour. `minicluster-00` shows nearly optimal, linear scaling in Figure 14b - however this is not difficult as there is only a single node available and no network communication is involved. `minicluster-intel` features very good scaling as well (Figure 14c). Note that the data points for more than 10 MPI ranks show slightly reduced performance, because those are the multi-node cases where communication via Infiniband comes into play and additional network latency has an effect. On the cluster HSUper, waLBerla also scales very well for a small number of ranks, as shown in Figure 14d. However, since HSUper has a large number of nodes, the strong scaling limit of the relatively small test scenario is quickly reached. Note that there are only 73 cell blocks in total on all mesh levels, and these blocks are distributed via a load balancing mechanism. Thus, this benchmark scenario test case can not benefit much from more than 73 MPI ranks. However we can observe that even for a much larger number of nodes involved, the MLUPS performance stays constant and does not decrease.

Figure 15 shows the speedup (compared to sequential execution) over the number of MPI ranks used on the AMD testbed. As mentioned earlier, we expect to reach the strong scaling limit of the test scenario at around O(100) ranks, and indeed this

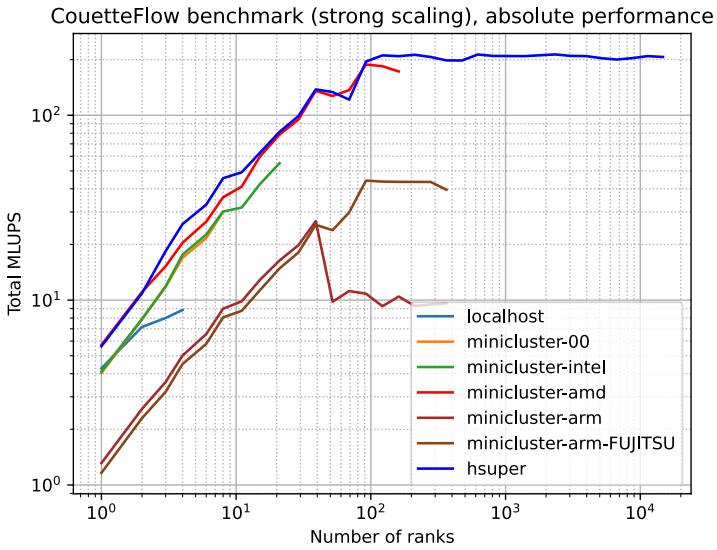


Figure 17: Direct comparison of absolute waLBerla performance on all benchmark machines

can be seen in the data plot. However, for a smaller number of ranks, we obtain nearly optimal scaling.

In a similar way to Figures 14 and 15, the speedup for the ARM based testbed is depicted in Figure 16. Here we distinguish between the GCC and OpenMPI software stack with results shown in Figure 16a and the FCC and Fujitsu MPI software stack shown in Figure 16b. In Figure 16a, there is a significant performance decrease visible once the number of MPI ranks exceeds the number of cores per node (48) so that several nodes are used. This can be explained by GCC and OpenMPI not using the hardware in an optimal way and introducing a high latency for using the Infiniband network connection. In contrast to that, as shown in Figure 16b, the FCC and Fujitsu MPI software stack solves this problem, most probably since the vendor-specific MPI implementation allows to use the Fujitsu networking hardware with much lower latency. Thus, we obtain significantly better scaling behaviour in Figure 16b.

Finally, the absolute MLUPS performance results are shown in Figure 17 and directly compared for all the benchmark host systems. The large performance difference to the other systems indicates that, amongst others, the compilers fail to employ vectorization effectively for this C++ Lattice-Boltzmann kernel code and thus waste the hardware capabilities in this case.
The three older (2019) Intel based architectures (`localhost`, `minicluster-00` and `minicluster-intel`) each reach a performance of 4.1 ± 0.15 MLUPS per core, which is not surprising as the CPU architectures are very similar.
The two more recent (2021) Intel and AMD architectures both show a single process

performance of 5,67±0,05 MLUPS, and the highest absolute performance at about 100 ranks: The AMD testbed reaches 188.15 MLUPS for 92 MPI ranks, HSUper reaches 210.84 MLUPS for 122 MPI ranks.

4.5 Conclusion

The results demonstrate that the most recent Intel and AMD CPU architectures are best for this code and that both perform similarly well for this scenario.

5 MaMiCo

5.1 Introduction

In the present section, the performance of the macro-micro-coupling tool (MaMiCo), that is developed at the chair for high performance computing at HSU, is tested on the three processor architectures, cf. Subsections 1.1 to 1.3. MaMiCo provides different features and functionality to control the coupling of molecular and fluid dynamics simulations. In particular, it allows to run several quasi-identical MD simulations and couple them to a single CFD simulation.

5.2 Challenges

MaMiCo is compiled on each platform with gcc and one more, rather platform-specific compiler. On HSUper, MaMiCo is compiled with the gcc compiler as well as the Intel/oneAPI compiler. Furthermore, up to 64 nodes of 72 cores are used for testing. On the AMD cluster the gcc compiler as well as the AMD/AOCC compiler are used and up to 4 nodes of 64 cores are used for testing. On the ARM cluster beside the gcc compiler the FUJITSU compiler and up to 8 nodes of 48 cores are used for testing purposes. In each case, the impact of different optimization levels (with the compiler flags `-O2`, `-O3`) is considered.

5.3 Benchmark Case

In this work, we consider a molecular-continuum coupling simulation of so-called Couette flow. The set up is shown in Figure 18. The computational domain is decomposed into overlapping CFD and MD regions. Coupling is achieved by transferring mass and momentum in the overlap region. The green- and red-colored cells are used to impose mass and momentum values from the CFD solver onto the MD solver, whereas flow velocities are sampled in the orange-colored cells and used (in the outer orange-colored cells) as Dirichlet boundary conditions by the CFD solver. One CFD time step corresponds to 50 MD time steps. We refer to the CFD steps also as coupling cycles.

We consider Couette flow as our show case, which possesses an analytical solution for validation purposes. The 3D computational domain consists of a periodic channel between a moving wall at the top and a lower wall at the bottom (Figure 18). The fluid, which is initially at rest, is accelerated by the moving wall and converges

towards a linear velocity profile along the channel. It is modelled on MD side by a short-range single-site Lennard-Jones model.

All parameters in MaMiCo are chosen dimensionless corresponding to liquid argon at the density-temperature-defined state point $(\rho,T)=(0.81,1.1)$. Also, length scales are dimensionless. In this work we consider only a so-called MD30 domain, which consists of a $50\times50\times50$ domain and an embedded $30\times30\times30$ MD domain. We used the MD solver SimpleMD implemented in Mamico as our MD solver to perform different test setups as listed in Table 9:

Name	#MD sim.	#Cores/Nodes
SSS[a] 1 ARM/AMD/HSUper	576 MD30	Up to 48/64/72 cores
SSS 2 ARM/AMD/HSUper	4608 MD30	Up to 4/8/64 nodes
WSS[b] ARM/AMD/HSUper	Up to 4608 MD30	Up to 4/8/64 nodes

Table 9: MaMiCo test cases. The second column denotes the number of MD simulations in multi-instance sampling, cf. [24]; *a* Strong scaling setup; *b* Weak scaling setup.

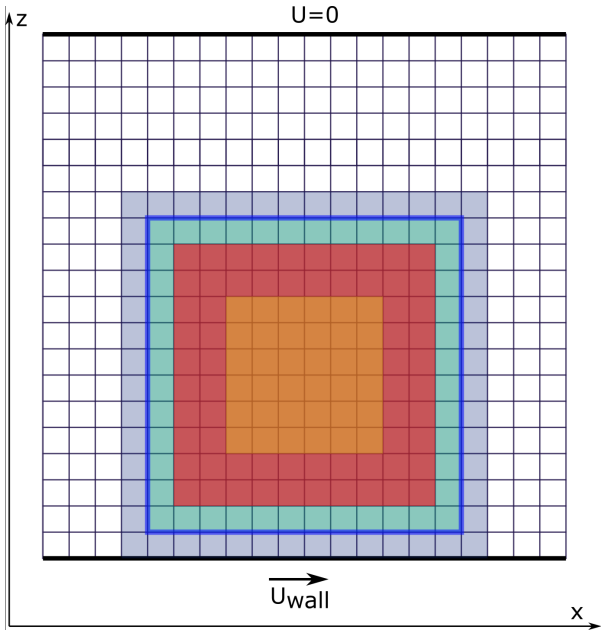


Figure 18: Cut through computational domain (x-z plane) with overlapping domain decomposition applied in MaMiCo, illustrating the CFD mesh as well as MD, CFD and overlapping domain. White/gray/green/red region: Macroscopic (CFD) domain, green/red region: Microscopic (MD) domain, gray region: ghost layer. Momentum/Mass transfer from CFD to MD solver and vice versa occurs in green/red cells. $z=0$ indicates the moving wall.

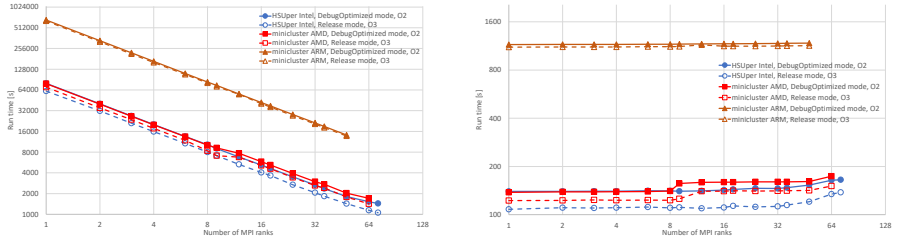
1. Strong scaling setup 1: 576 MD30 instances are launched on 1 to 48, 64 and 72 MPI ranks of ARM/ AMD/ HSUper testbeds respectively, such that one entire node of each platform is used. Figures 19 (a), 20 (a), 21 (a) and 22 (a) show the results according to this setup.

2. Strong scaling setup 2: 4608 MD30 instances are launched on 1 to 4, 8 and 32 nodes of ARM/ AMD/ HSUper testbeds respectively. Figure 23 (a) shows the results according to this setup.

3. Weak scaling setup: The number of MD30 instances is equal to the number of MPI ranks used in each simulation such that each core handles one single MD instance. Figures 19 (b), 20 (b), 21 (b), 22 (b) and 23 (b) show the results according to this setup.

5.4 Results

First of all, the MaMiCo performance is studied using gcc compilers on all three testbeds. Figures 19 (a) and (b) demonstrate the strong and weak scaling results on one single node of each platform. For the strong scaling test, 576 number of MD instances are chosen, so that they can be distributed homogeneously on all three platforms. The performance of HSUper and AMD using optimization level 2 (flag `-O2`) is very comparable, with the AMD results to be slightly better. Using optimization level 3 (flag `-O3`) has the biggest impact on HSUper and smallest impact on ARM performance, respectively. In all cases, the ARM testbed is almost 10 times slower than the others, which is not a completely fair comparison, since, for example, neither MaMiCo nor SimpleMD have been optimized for ARM architectures so far. In terms of the scalability, AMD shows the best, i.e. the most linear, scalability results. These trends can also be seen for the cases of more than one node in Figure 23 (a) and (b).

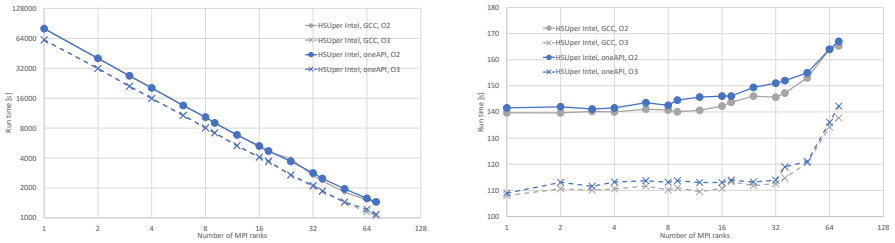


(a) (b)

Figure 19: Node level scalability in terms of time-to-solution (TTS) on different CPU architectures using the GCC compiler. Different optimization levels are compared. (a) strong scaling, (b) weak scaling.

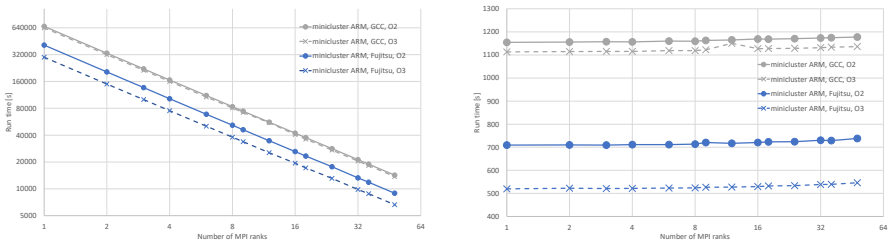
In the next step, we investigate the effect of different compilers on the different platforms. The results on HSUper are very similar, with slightly better results by using the Intel compiler instead of gcc (see Figure 20 (a) and (b)). Using the Fujitsu compiler instead of gcc has a huge impact on ARM results, such that the time-to-solution is cut in

half by the optimization level 2 (comparing the gray and blue solid lines in Figure 21 (a) and (b)). Even the optimization flag has a big impact of around 28% if the Fujitsu compiler is used (comparing the gray solid and dashed lines in Figure 21 (a) and (b)). Compiling MaMiCo with the AOCC compiler instead of gcc reduces the runtime to approx. 89% and 75%, using flags `-O2` and `-O3` respectively (see Figure 22 (a) and (b)).



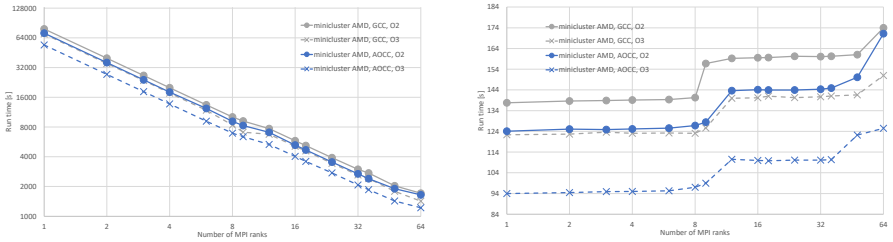
(a) (b)

Figure 20: Scalability results in terms of time-to-solution (TTS) on HSUper using the GCC and Intel/oneAPI compilers on one node with different optimization levels (`-O2` and `-O3` flags). (a) strong scaling, (b) weak scaling.



(a) (b)

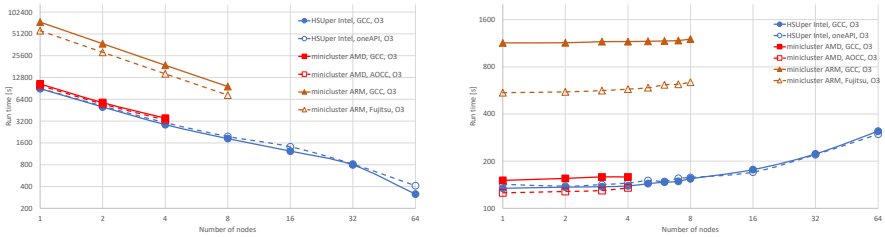
Figure 21: Scalability results in terms of time-to-solution (TTS) on ARM platform using the GCC and Fujitsu compilers on one node with different optimization levels (`-O2` and `-O3` flags). (a) strong scaling, (b) weak scaling.



(a) (b)

Figure 22: Scalability results in terms of time-to-solution (TTS) on AMD platform using the GCC and AMD/AOCC compilers on one node with different optimization levels (`-O2` and `-O3` flags). (a) strong scaling, (b) weak scaling.

Figures 23 (a) and (b) summarize the performance results of all three testbeds with different compilers using multiple nodes and flag `-O3`.



(a) (b)

Figure 23: Scalability results in terms of time-to-solution (TTS) on different CPU architectures using different compilers on multiple nodes. The optimization level is set to O3 + (`-Ofast` flag). (a) strong scaling, (b) weak scaling.

5.5 Conclusion

In this study, first the MaMiCo performance is studied using gcc compilers. The performance of HSUper and AMD using optimization level 2 (flag -O2) show to be very comparable, with the AMD results to be slightly better. The performances using the gcc compiler and optimization level 2 (flag -O2) is considered as references in this study. Table 10 summarizes all speed-up results using different compilers and optimization levels compared to these reference values.

Platform	HSUper (Intel)	AMD Testbed	ARM Testbed
gcc, -O3	$\approx$ 1.3	$\approx$ 1.15	$\approx$ 1.03
oneAPI, -O2	$\approx$ 0.99-1.01	-	-
oneAPI, -O3	$\approx$ 1.3	-	-
AOCC, -O2	-	$\approx$ 1.07-1.1	-
AOCC, -O3	-	$\approx$ 1.43	-
Fujitsu, -O2	-	-	$\approx$ 1.6
Fujitsu, -O3	-	-	$\approx$ 2.17-2.2

Table 10: Speed-up results using different compilers and optimization levels compared to gcc compiler and optimization level 2 (flag -O2) results as references

It can be seen that using Fujitsu compiler instead of gcc on the ARM platform has the biggest impact on the performance, especially when optimization level 3 (-O3) is activated. On the other hand, the difference between gcc and oneAPI compilers are minor. HSUper is shown to be the best platform to run MaMiCo with the smallest runtime, followed by AMD, whose results are very close to the ones of HSUper. It is also advisable to use the compilers Intel/AOCC/Fujitsu instead of gcc on HSUper/AMD/ARM platforms, respectively. Especially, it has a very big effect on ARM. When the flag -O3 optimization is used, the influence gets even bigger.

6 Gurobi

6.1 Introduction

Gurobi[19] is a closed-source commercial solver which can be applied to different types of optimization problems such as mixed-integer linear programming (MILP)[20]. Parallel shared-memory as well as distributed optimization, i.e., multithreading and MPI processes[21], are supported on various hardware architectures, e.g., x86 and ARM, as well as different operating systems such as Windows, Mac, Linux, AIX. However, we focus on the performance of Gurobi on a single cluster node and do not test the multi-node version[22]. Gurobi provides performance benchmarks[23] and general guidelines about good hardware setups[24]. Practical guidelines for solving MILPs are discussed in [13]. Parallel MILP solvers and their progress in recent years are studied in [14, 16, 31].

[19] https://www.gurobi.com

[20] https://support.gurobi.com/hc/en-us/articles/360013156432-What-types-of-models-can-Gurobi-solve

[21] https://www.gurobi.com/events/parallel-and-distributed-optimization-with-gurobi/

[22] https://www.gurobi.com/solutions/gurobi-compute-server/

[23] https://cdn.gurobi.com/wp-content/uploads/Whats-New-in-Gurobi-11.0_Webinar-slides.pdf?x34984, https://gurobi.com/pdfs/benchmarks.pdf

[24] https://support.gurobi.com/hc/en-us/sections/360010047491-Hardware

6.2 Challenges

A major challenge in evaluating the performance of a MILP solver is performance variability, since the performance is intrinsically not deterministic (i.e., seemingly performance-neutral changes such as a changed order of the constraints, or using a different random seed or different hardware can lead to a significant change in the performance) [6, 15, 19]. Benchmarking MILP software is further discussed in [1, 20, 22].

The Gurobi solver is deterministic as long as no time-dependent behavior such as concurrent optimization is included[25]. However, the actual amount of work[26] during the solution of a MILP may be different if the hardware or a solver parameter such as the number of threads used is changed[27]. Hence, we stick mainly to the Gurobi default parameters[28] and do not focus on parameter tuning[29]. Another challenge is that Gurobi is a closed-source solver and reverse assembling, reverse compiling, translating, or reverse engineering are not allowed due to the end-user license agreement. Hence, Gurobi is a blackbox solver and the performance evaluation is restricted to the content of the Gurobi log file and the study of performance counters via tools such as LIKWID [36]. LIKWID is the main data source for the continuous benchmarking tool xbat[30] which is utilized in [35] to study the hardware usage of the Gurobi solver on HSUper.

6.3 Benchmark Case

We use mainly the time-dependent traveling salesman problem ta064 and the corresponding 3PQ.mod and ta064.opl.dat files from [9] to generate an mps file[31] via the command

```
oplrun -e ta064.mps 3PQ.mod ta064.opl.dat
```

for IBM ILOG CPLEX OPTIMIZATION STUDIO[32], VERSION 22.1.1 for Linux x86 with 64 bits. Moreover, we show results for further time-dependent traveling salesman problems from [9], which have the same problem size but usually need a larger runtime than ta064 [29]. The command

```
gurobi_cl Threads=1 Seed=0 ta064.mps
```

starts the Gurobi solver (Version 11.0.1, build v11.0.1rc0 for (ARM-)Linux with 64 bit is used) with 1 thread and seed 0 on 1 compute node. HSUper (Subsection 1.1) and AMD (Subsection 1.2) are used with 1, 2, 4, 8, 64 and 128 threads and ARM (Subsection 1.3) is used with up to 32 threads (i.e., the number of threads is doubled until it exceeds the available number of virtual CPUs). The setups with 128 threads

[25] https://support.gurobi.com/hc/en-us/articles/360031636051-Is-Gurobi-deterministic-

[26] https://www.gurobi.com/documentation/11.0/refman/work.html

[27] https://support.gurobi.com/hc/en-us/articles/360045849232-Why-does-Gurobi-perform-differently-on-different-machines-

[28] https://www.gurobi.com/documentation/11.0/refman/parameters.html

[29] https://www.gurobi.com/documentation/11.0/refman/parameter_tuning_tool.html

[30] https://www.megware.com/en/products/xbat

[31] https://www.gurobi.com/documentation/11.0/refman/mps_format.html

[32] https://www.ibm.com/products/ilog-cplex-optimization-studio

use simultaneous multithreading, i.e., the number of threads used is larger than the number of physical cores.

6.4 Results

Figure 24 shows the influence of the seed value on the runtime in seconds, the work, the number of explored nodes and the number of simplex iterations w.r.t. the number of threads for ta064 on HSUper. Each plot consists of 30 markers since 10 different seed values are used and each setup is computed three times. However, only the runtime can vary if a setup is computed again. This figure illustrates that the seed value has a huge influence on the number of explored nodes and the number of simplex iterations, and with that on the runtime and the amount of performed work.

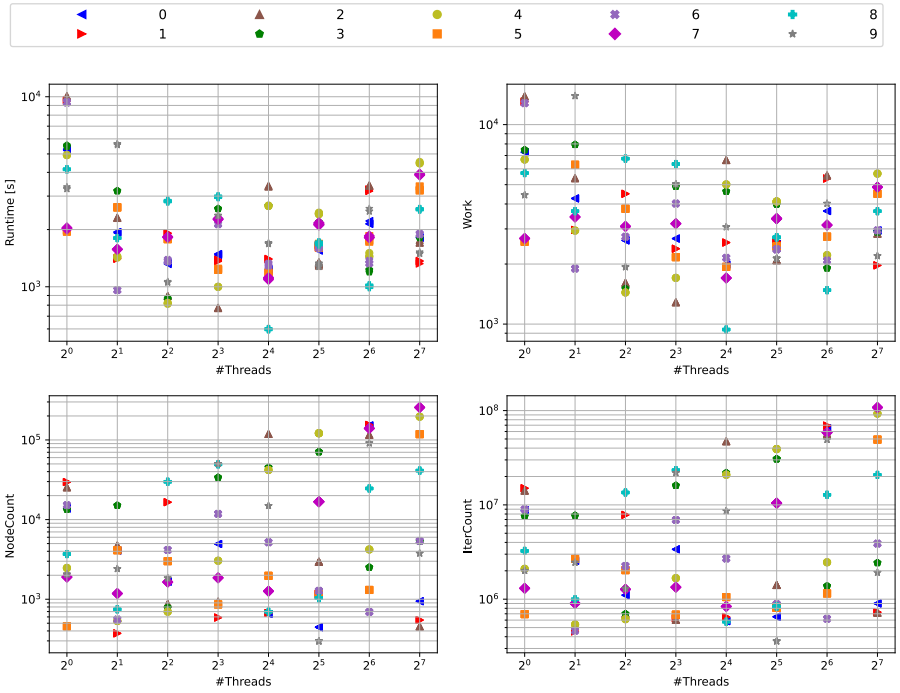


Figure 24: Runtime, work, number of explored nodes and number of iterations w.r.t. the number of threads used for ta064 with various seed values on HSUper.

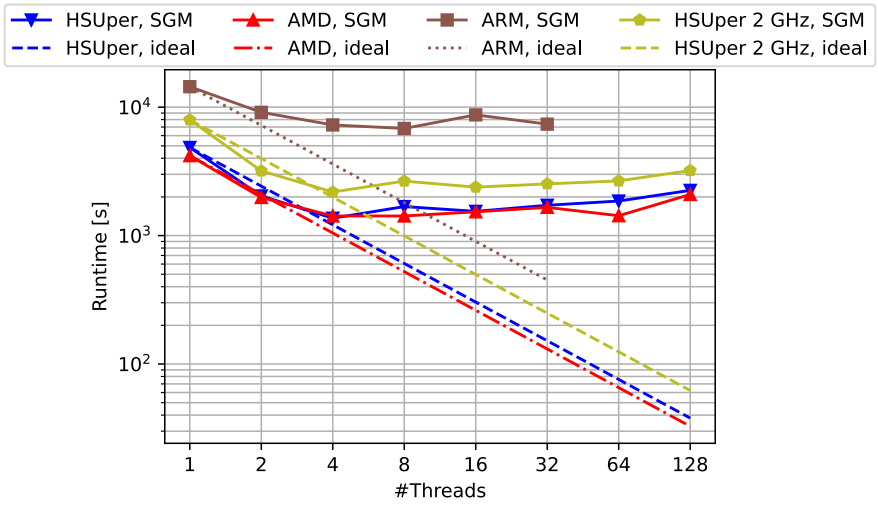


Figure 25: Strong scaling tests for ta064 on the three testbeds HSUper, AMD and ARM. Wall-clock time in seconds w.r.t. the number of threads used.

Figure 25 shows strong scaling results corresponding to the computations in Figure 24. The plot contains for each hardware the shifted geometric mean (SGM)[33] (with a shift value of 100) of all 30 runs and its ideal strong scaling behavior. In addition, there are results for HSUper via

```
srun --cpu-freq=2000000-2000000:performance
```

with a fixed clock speed of 2 GHz which is equal to the clock speed of the ARM system. This demonstrates that it is hard to estimate a priori the optimal number of threads since increasing the number of threads can increase as well as decrease the runtime. Moreover, the optimal thread number depends on whether the minimal, the maximal or the mean runtime should be minimized [27]. Nevertheless, ARM leads for a fixed number of threads to the highest runtime whereas HSUper and AMD typically lead to significantly smaller runtimes. The runtimes of HSUper are increased by fixing the clock speed to 2 GHz but they are still significantly below the runtimes of ARM. The speedup (in terms of the shifted geometric mean) for using the AMD or HSUper system instead of the ARM system is in our tests between 2.9 and 5.8 (see Table 11). Close-to-ideal strong scaling behavior is obtained on ARM up to 2 threads and on HSUper and AMD up to 4 threads. Using a significantly larger number of threads is usually not advisable which corresponds to Gurobi's choice to impose a soft limit of 32 threads in the Gurobi solver.

[33] https://community.fico.com/s/blog-post/a5Q2E000000Dt0JUAS/fico1421

Threads	speedup AMD vs ARM	speedup HSUper vs ARM
1	≈ 3.4	≈ 3
2	≈ 4.6	≈ 4.5
4	≈ 5.1	≈ 5.3
8	≈ 4.8	≈ 4.1
16	≈ 5.7	≈ 5.6
32	≈ 4.4	≈ 4.3

Table 11: Speedup w.r.t. the results in Figure 25 for using the AMD or HSUper system instead of the ARM system.

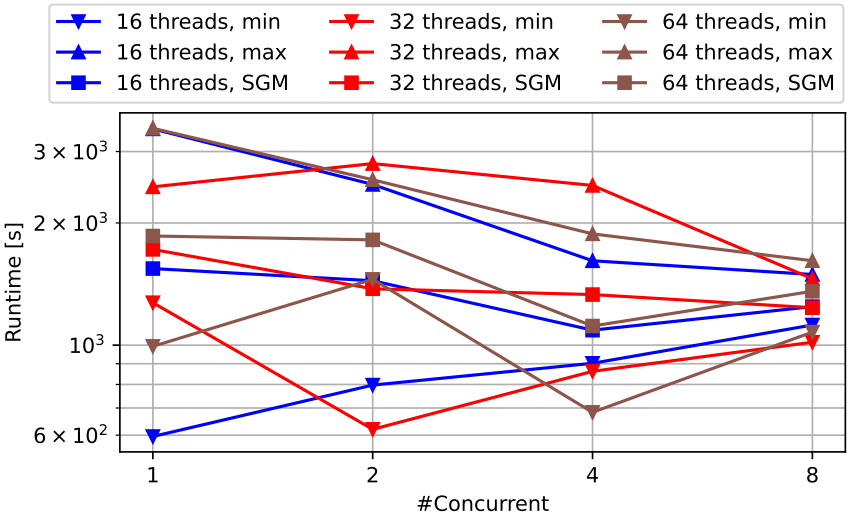


Figure 26: Concurrent optimization tests for ta064 with various thread numbers on HSUper. Wall-clock time in seconds w.r.t. the number of concurrent instances.

Figure 26 shows results for concurrent optimization[34] with 16, 32 and 64 threads. The idea of this parallelization approach is to split the threads into groups such that each group starts an independent MIP solve with different parameter settings and to stop the solution when the first MIP solve terminates. Each setup is solved three times for 10 different seeds. This plot indicates that concurrent optimization can reduce the runtime. In particular, the maximal runtime and the SGM of the runtimes are often reduced if the number of concurrent instances is increased.

[34] https://www.gurobi.com/documentation/11.0/refman/concurrentmip.html

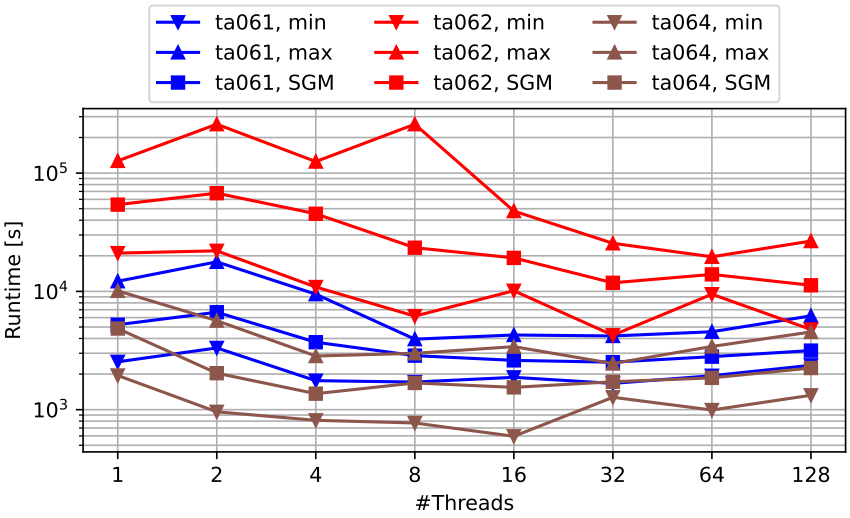


Figure 27: Strong scaling tests for ta061, ta062, ta064 on HSUper. Wall-clock time in seconds w.r.t. the number of threads used.

Figure 27 is similar to the Figure in [27] except that the hardware is fixed to HSUper and various test problems are solved. The runs for ta062 lead for 2 threads and 8 threads for one seed value to a timeout w.r.t. our time limit of 259140s = 71h 59min, i.e., the corresponding max values show this time limit. Furthermore, it can be observed that different test problems lead to different optimal thread numbers. However, a moderate number of threads (e.g., ≤ 32) can usually reach a relatively small runtime.

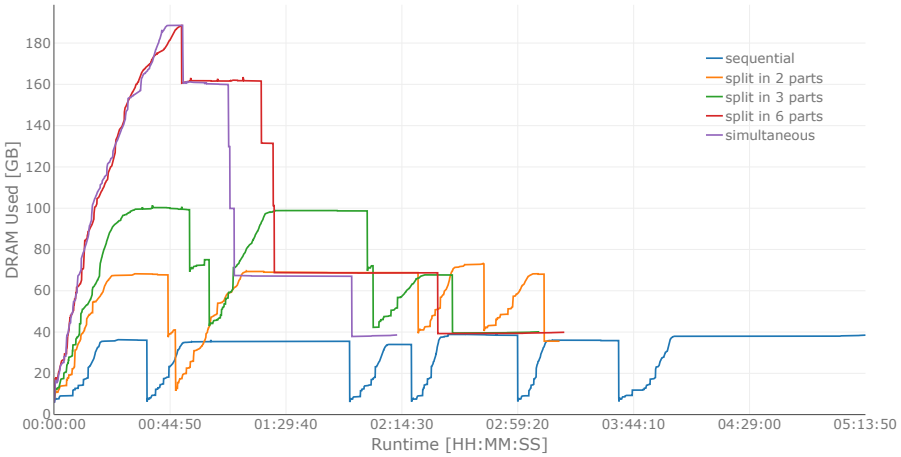


Figure 28: RAM usage in GB as a function of the runtime for various options for solving ta061, ta063, ta064, ta065, ta067 and ta068 on HSUper.

Figure 28 shows the RAM usage in GB as a function of the runtime for various options for solving the test problems ta061, ta063, ta064, ta065, ta067 and ta068 on HSUper with Gurobi default parameters, i.e., each Gurobi run uses 32 threads. The difference between these options is the scheduling of the various Gurobi runs. The highest runtime corresponds to "sequential" and means that one problem is solved after the other. The smallest runtime corresponds to "simultaneous" and means that all Gurobi runs are performed simultaneously, i.e., the $32 \cdot 6 = 192$ Gurobi threads share the 144 virtual CPUs of one HSUper node. The "split" options mean that one node is split via SLURM and individual srun commands into 2, 3 or 6 equal parts and each part runs one problem at a time. It can be observed that each Gurobi run needs on HSUper up to approximately 30 to 40 GB RAM and their simultaneous run can lead to a RAM usage of more than 180 GB.

6.5 Conclusion

The smallest runtimes are often reached for setups with around 4 to 64 threads, i.e., it is often not worthwhile to use more than a few cores. However, the ARM system leads usually to the highest runtimes whereas the HSUper and AMD systems are often significantly faster and show comparable performance (e.g., for the ta064 benchmark). These observations are in accordance with Greg Glockner (Gurobi)[35] that Gurobi is not yet highly tuned for ARM processors and large core counts since most of their customers have hardware with 4 to 16 cores. Gurobi states that "For most of the algorithms in Gurobi, the performance of the machine is ultimately limited by the rate at which the cores can pull data from memory. With the machines we're used to, it only takes a few cores to saturate the memory system. These new AMD machines provide more memory bandwidth, which leads to better parallel scaling. There are limits to how much parallelism you can exploit, but you hit those limits much later on these new systems."[36]. Hence, running a single Gurobi model on one exclusive HSUper node would benefit from a change of the configuration[37], i.e., from a reduction of the number of available cores in exchange for an increased clock speed [26].

The results in [28] showed that limiting the number of cores available to the Gurobi solver leads on the HSUper system only to a moderate increase in the runtime. A reason for this observation is that the Gurobi solver frequently does not utilize all available threads on hardware with many cores [35]. Hence, the most efficient use of the Gurobi solver on hardware with many cores seems to be to run several models simultaneously as in Figure 28 (optionally, with a manually set small thread parameter to avoid oversubscribed hardware)[38]. However, running several models simultaneously requires more RAM which could lead to problems on the ARM system due to its relatively small RAM of 32 GB. Furthermore, our results showed

[35] https://www.nvidia.com/en-us/on-demand/session/gtc24-s62555/

[36] https://www.gurobi.com/resources/better-hardware-for-solving-mathematical-optimization-problems-the-future-is-looking-bright/

[37] https://www.intel.com/content/www/us/en/products/sku/212459/intel-xeon-platinum-8360y-processor-54m-cache-2-40-ghz/specifications.html

[38] https://support.gurobi.com/hc/en-us/articles/25414264687121-Solving-batches-of-models-with-short-model-runtimes-Efficient-API-usage-parameter-tuning-and-deployment-considerations

that a decrease in the clock speed leads to an increase in the runtime which also favors the HSUper and AMD systems over the ARM system.

Acknowledgment

We would like to express our gratitude for the opportunity to utilize the computational resources of the HPC cluster HSUper and the testbeds provided by the projects hpc.bw and MaST. hpc.bw and MaST are funded by dtec.bw – Digitalization and Technology Research Center of the Bundeswehr. dtec.bw is funded by the European Union – NextGenerationEU.

Appendix

A Power Measurement

Power and energy measurements were taken on all testbeds with a metered PDU. We have created a simple Python3 utility[39] to execute benchmark workloads on the testbed over SSH while periodically logging measurements from the PDU. During

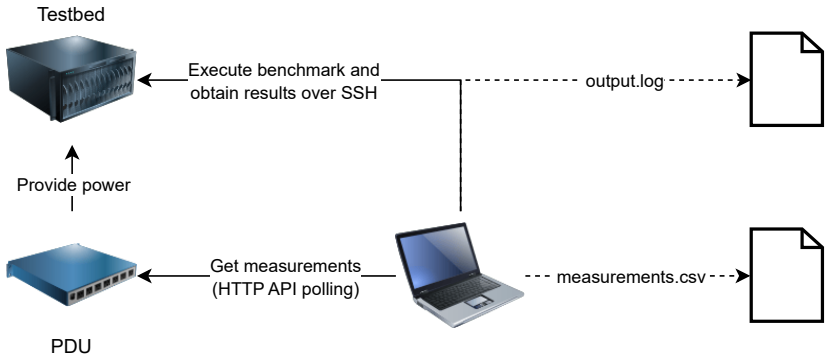


Figure 29: A visualization of the power measurement setup for benchmarking the testbeds.

the execution of the benchmarks, the testbed is powered through the PDU, which is connected to the benchmark execution system using Ethernet (Figure 29). For the PDU, we used the netio PowerPDU 4KS, which provides power and energy measurements for all four outputs over an HTTP interface. According to the manufacturer, the provided values are accurate up to an error of less than 1% [23]. Since the measurements are updated at a rate of 1 Hz, the API is polled at a rate of 2.1 Hz according to the Nyquist–Shannon sampling theorem [33]. The energy consumption is measured in whole watt-hours (Wh). Therefore, we instead derive the total energy consumption by integrating the instantaneous power consumption, measured in whole watts (W) over the time between consecutive measurements. In order to not capture the *warm-up* and *cool-down* phase of the benchmark, 60 seconds of the recorded data are cut from the start and end before processing. On all systems, the power consumption was measured for up to 4/8 nodes, an excerpt of which is visualized in Figure 30, using the following scenarios:

- **Standby:** All nodes are shut down, and only the standby power consumption of the power supplies is measured for at least 60 seconds.
- **Idle:** All nodes are powered up, and the idle power consumption is measured for 5 minutes.

[39] `https://git.hpc.hsu-hh.de/rubenhorn/energybar` (HSU internal)

- **Stress(η):**All nodes are powered up, a fraction of them (η) perform floating point matrix multiplications at full CPU saturation using stress-ng [12] (`stress-ng --matrix 0 --cpu-load 100 --cpu-ops` n with $n=10^8$ *bogus CPU operations*) while the rest remain idle. The duration of this benchmark scenario depends on the testbed.

- **Stream(η):** All nodes are powered up, a fraction of them (η) perform the stream micro-benchmark [21] revision 5.10 with an array size of 10^7 and 2.5×10^5 runs per kernel while the rest remain idle. The duration of this benchmark scenario depends on the testbed.

The stress-ng[40] and stream binaries were build from source with the Red Hat GCC version 8.5.0-10 on HSUper and 8.5.0-22 on the AMD and ARM testbeds. Stream was built with OpenMP support but without any optimizations for a common baseline. Since the ARM testbed provides a total of 8 nodes in one chassis, experiments were performed for both 4 and 8 nodes powered by the PDU for comparability to the other testbeds and complete assessment of the ARM testbed. Where necessary, the corresponding resources were reserved through the slurm workload manager. Jobs running (`sleep`) were manually started to prevent idle nodes from shutting down after 15 minutes in the stress and stream scenarios on HSUper. The results of the power measurement using the benchmark scenarios described above are visualized in Figure 31. For these experiments, non-optimized binaries were build using GCC instead of the platform optimized compiler in order to obtain a general power consumption under load.

[40] https://github.com/ColinIanKing/stress-ng/tree/caba0aec67a59f5d510f4d967504cbf9e6881a75

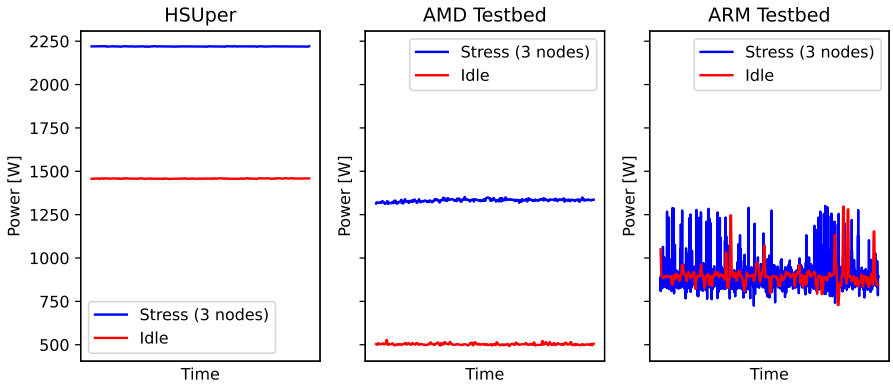


Figure 30: Power consumption of the three testbeds during the idle and stress benchmarks. The power consumption of the ARM system has much more extreme peaks than either of the other two testbeds.

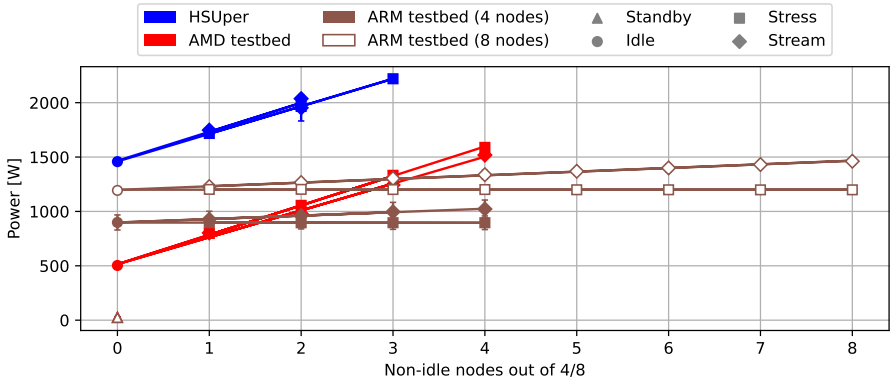


Figure 31: Visualization of the power ($\mu \pm \sigma$) measured across the testbeds with the corresponding linear fit. The range of utilized nodes differs between testbeds due to the power output being limited to 2300 W on the PDU. The power consumption for the stress-ng benchmark is constant on the ARM testbeds regardless of node utilization.

B Stream Benchmark

In addition to the *standard* stream benchmark build used for power consumption measurements described in Section A, an optimized binary is compiled for each testbed. The compile time and run time optimizations are described below for each platform. On all platforms, static scheduling of OpenMP threads at runtime is ensured using the following environment variables:

```
export OMP_SCHEDULE=static
export OMP_DYNAMIC=false
```

On the AMD and ARM testbeds, the threads are pinned to the cores using `export OMP_PROC_BIND=TRUE`.

B.1 HSUper (Intel)

On HSUper, following the optimizations described by the Intel documentation[41], the Intel C/C++ compiler (ICC) is used to build the application with the following flags:

- `-xCORE-AVX512`
- `-qopt-zmm-usage=high`
- `-DNTIMES=100`
- `-DOFFSET=0`

[41] https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-memory-bandwidth-on-stream-triad.html

- `-DSTREAM_TYPE=double`
- `-DSTREAM_ARRAY_SIZE=268435456`
- `-O3`
- `-mcmodel=medium`
- `-qopenmp`
- `-shared-intel`
- `-qopt-streaming-stores always`

At runtime, the number of threads and their placement was using the following environment values:

```
export OMP_NUM_THREADS=72
export KMP_AFFINITY=granularity=fine,compact,1,0
```

B.2 AMD Testbed

On AMD, the optimized binary, built with the AMD Optimizing C/C++ Compiler (AOCC), can be obtained through the spack package manager with the command:

```
spack install stream +openmp %aocc \
    stream_array_size=430080000 \
    ntimes=100 \
    stream_type=double
```

Following the recommendations by AMD[42], the operating system memory settings are updated before running the benchmark:

```
echo always > /sys/kernel/mm/transparent_hugepage/enabled
echo always > /sys/kernel/mm/transparent_hugepage/defrag
echo 3 > /proc/sys/vm/drop_caches
echo 1 > /proc/sys/vm/compact_memory
```

The placement of threads to the 64 cores is controlled using the following environment variables:

```
export OMP_PLACES=0:64:2
export OMP_NUM_THREADS=64
```

[42] https://www.amd.com/de/developer/zen-software-studio/applications/spack/stream-benchmark.html

B.3 ARM Testbed

On ARM, the application is built using the Fujitsu GCC (FCC) with the following flags:

- `-DNTIMES=100`
- `-DOFFSET=0`
- `-DSTREAM_TYPE=double`
- `-DSTREAM_ARRAY_SIZE=100000000`
- `-O3`
- `-fopenmp`

To enable large memory pages and set the number of threads to match the hardware, the following environment variables are exported:

```
export XOSMMM_L_PAGING_POLICY=demand:demand:demand
export OMP_NUM_THREADS=48
```

C OSU Micro-Benchmarks

The OSU Micro-Benchmarks 7.4 package was compiled using different compilers and MPI implementations.

C.1 HSUper (Intel)

An OpenMPI/gcc build was created using mpicc/mpicxx from GCC 12.1.0 and Open MPI 4.1.3.

Two builds of the OSU Micro-Benchmarks 7.4 package were compiled using Intel compilers. The Intel build was created using mpiicx from

- Intel(R) oneAPI DPC++/C++ Compiler 2024.1.0 (2024.1.0.20240308)
- Intel(R) MPI Library for Linux* OS, Version 2021.12 Build 20240410 (id: 49f2f80)

For execution, the Intel(R) MPI Library for Linux* OS, Version 2021.6 Build 20220227, was employed to maintain comparability with the Intel classic build.

The Intel classic build was compiled using mpiicc and mpiicpc from:

- Intel(R) ICC 2021.6.0 20220226
- Intel(R) MPI Library for Linux* OS, Version 2021.6 Build 20220227

Every build was executed via a Slurm job script without any additional environment variables. The srun command was used to launch the execution on two nodes, each with one task and one CPU per task.

C.2 AMD and ARM Testbeds

For the AMD and ARM testbeds, an OpenMPI/gcc build was compiled separately on each platform using mpicc/mpicxx from GCC 8.5.0 and Open MPI 4.1.2.

Each build was executed via a Slurm job script without any additional environment variables set. The mpirun command was used to launch the executions on two nodes, with one process per node.

D likwid-bench Measurements

likwid-bench, version 5.4.0, was used in all runs. Configuration was leveraging AVX (AMD), AXX512 (Intel/HSUper) and SVE (ARM) support.

D.1 HSUper (Intel)

On HSUper, the benchmark was executed according to:
`likwid-bench -t peakflops_avx512_fma -w S0:512kB -w S1:512kB -i 160000000`
Although the theoretical peak is at ca. 5.5 TFLOPS, a higher clock speed of up to 2.6 GHz (instead of 2.4 GHz) was observed during this benchmark.

D.2 AMD Testbed

On AMD, the benchmark was executed according to:
`./likwid-bench -t peakflops_avx_fma -w N:512kB -i 80000000`

D.3 ARM Testbed

On ARM, the benchmark was executed according to:
`./likwid-bench -t peakflops_sve512_fma -w S0:256kB -w S1:256kB -w S2:256kB -w S3:256kB -i 8000000`

References

[1] T. Achterberg and R. Wunderling. "Mixed integer programming: Analyzing 12 years of progress". In: *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*. Springer, 2013, pp. 449–481.

[2] M. Breuer. "A challenging test case for large-eddy simulation: High Reynolds number circular cylinder flow". In: *Int. J. of Heat and Fluid Flow* 21.5 (2000), pp. 648–654.

[3] M. Breuer. *Direkte Numerische Simulation und Large-Eddy Simulation turbulenter Strömungen auf Hochleistungsrechnern*. Habilitationsschrift, Universität Erlangen–Nürnberg, Berichte aus der Strömungstechnik. Sha-ker, Aachen, Germany, 2002. ISBN: 3–8265–9958–6.

[4] M. Breuer. "Effect of Inflow Turbulence on an Airfoil Flow with Laminar Separation Bubble: An LES Study". In: *Flow, Turbul. and Combust.* 101.2 (2018), pp. 433–456.

[5] M. Breuer. "Large-eddy simulation of the sub-critical flow past a circular cylinder: Numerical and modeling aspects". In: *Int. J. for Numerical Methods in Fluids* 28.9 (1998), pp. 1281–1302.

[6] E. Danna. "Performance variability in mixed integer programming". In: *Workshop on Mixed Integer Programming, Columbia University, New York*. Vol. 20. 2008.

[7] T. Eke. "AutoPas on A64FX: Evaluation of Arm SVE Vectorization for Optimizing Molecular Dynamics Simulations". MA thesis. Technical University of Munich, 2022.

[8] C. Feichtinger et al. "WaLBerla: HPC software design for computational engineering simulations". In: *J. of Comput. Sci.* 2.2 (2011), pp. 105–112.

[9] A. Fink and S. Voß. "Solving the continuous flow-shop scheduling problem by metaheuristics". In: *European J. Oper. Res.* 151.2 (2003), pp. 400–414.

[10] F. A. Gratl et al. "AutoPas: Auto-Tuning for Particle Simulations". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Rio de Janeiro: IEEE, 2019.

[11] M. Griebel, G. Zumbusch, and S. Knapek. *Numerical Simulation in Molecular Dynamics*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2007, p. 96. DOI: 10.1007/978-3-540-68095-6.

[12] C. I. King. *stress-ng – stress test a computer system*. https://wiki.ubuntu.com/Kernel/Reference/stress-ng. Retrieved July 24, 2024. 2020.

[13] E. Klotz and A. M. Newman. "Practical guidelines for solving difficult mixed integer linear programs". In: *Surv. Oper. Res. Manag. Sci.* 18.1-2 (2013), pp. 18–32.

[14] T. Koch, T. Ralphs, and Y. Shinano. "Could we use a million cores to solve an integer program?" In: *Math. Methods Oper. Res.* 76 (2012), pp. 67–93.

[15] T. Koch et al. "MIPLIB 2010: mixed integer programming library version 5". In: *Math. Program. Comput.* 3 (2011), pp. 103–163.

[16] T. Koch et al. "Progress in mathematical programming solvers from 2001 to 2020". In: *EURO J. Comput. Optim.* 10, 100031 (2022).

[17] H. C. Kuhlmann and F. Romanò. "The Lid-Driven Cavity". In: *Computational Modelling of Bifurcations and Instabilities in Fluid Dynamics*. Cham: Springer International Publishing, 2019, pp. 233–309. DOI: 10.1007/978-3-319-91494-7_8.

[18] E. Leriche and S. Gavrilakis. "Direct numerical simulation of the flow in a lid-driven cubical cavity". In: *Phys. Fluids* 12.6 (2000), pp. 1363–1376. DOI: 10.1063/1.870387.

[19] A. Lodi and A. Tramontani. "Performance variability in mixed-integer programming". In: *Theory Driven by Influential Applications*. INFORMS, 2013, pp. 1–12.

[20] S. J. Maher, T. K. Ralphs, and Y. Shinano. *Assessing the Effectiveness of (Parallel) Branch-and-bound Algorithms*. 2021. arXiv: 2104.10025.

[21] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. https://www.cs.virginia.edu/stream/. Retrieved July 24, 2024. 2013.

[22] H. D. Mittelmann. "Benchmarking Optimization Software - a (Hi)Story". In: *SN Oper. Res. Forum*. Vol. 1. 2020, 2.

[23] NETIO products a.s. *PowerPDU 4KS*. https://www.netio-products.com/en/device/powerpdu-4ks. Retrieved July 24, 2024.

[24] P. Neumann and X. Bian. "MaMiCo: Transient Multi-Instance Molecular-Continuum Flow Simulation on Supercomputers". In: *Comput. Phys. Commun.* 220 (2017), pp. 390–402.

[25] P. Neumann et al. "hpc.bw: A Supercomputer with Competence Platform for the Universities of the Federal Armed Forces". In: *dtec-Beiträge der Helmut-Schmidt-Universität / Universität der Bundeswehr Hamburg: Forschungsaktivitäten im Zentrum für Digitalisierungs- und Technologieforschung der Bundeswehr dtec.bw*. Vol. 1. 2022, pp. 305–310. DOI: 10.24405/14569.

[26] P. Neumann and M. Rathmann. *Newsletter hpc. bw 03/2023*.

[27] P. Neumann and M. Rathmann. *Newsletter hpc.bw 01/2024*.

[28] P. Neumann and M. Rathmann. *Newsletter hpc.bw 02/2024*.

[29] P. Neumann and M. Rathmann. *Newsletter hpc.bw 04/2023*.

[30] C. Niethammer et al. "ls1 mardyn: The massively parallel molecular dynamics code for large systems". In: *Journal of Chem. Theory and Comput.* 10.10 (2014), pp. 4455–4464.

[31] T. Ralphs et al. "Parallel solvers for mixed integer linear optimization". In: *Handbook of Parallel Constraint Reasoning* (2018), pp. 283–336.

[32] S. Seckler et al. "AutoPas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning". In: *J. of Computat. Sci.* 50, 101296 (2021).

[33] C. Shannon. "Communication in the Presence of Noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21. DOI: 10.1109/jrproc.1949.232969.

[34] N. Tchipev et al. "TweTriS: Twenty trillion-atom simulation". In: *The Int. J. of High Perf. Comput. Appl.* 33.5 (2019), pp. 838–854. DOI: 10.1177/1094342018819741.

[35] N. Tippmann et al. *xbat: A Continuous Benchmarking Tool for HPC Software*. Submitted, 2024.

[36] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environment". In: *2010 39th international conference on parallel processing workshops*. IEEE. 2010, pp. 207–216.