EUROfusion

EUROFUSION WPISA-REP(18) 20587

R Hatzky et al.

HLST Core Team Report 2017

REPORT

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

HLST Core Team Report 2017

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014–2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission

Contents

1. Executive Summary

1.1. *Progress made by each core team member on allocated projects*

In agreement with the HLST PMU responsible officer, Richard Kamendje, the individual core team members have been/are working on the projects listed in Table 1.

Project acronym	Core team member	Status
BIT2-3	Kab Seok Kang	running
CINCOMP2	Serhiy Mochalskyy Nils Moschüring	finished
GOKE	Tamás Fehér	finished
GRIMG2	Kab Seok Kang	finished
JORSTAR2	Serhiy Mochalskyy	finished
REFMUL3	Tiago Ribeiro	finished
SOLPSOPT	Tamás Fehér	finished
VIRIATO2	Tiago Ribeiro	finished

Table 1 Projects distributed to the HLST core team members.

Roman Hatzky has been involved in the support of the European users on the CINECA computer, MARCONI-Fusion. Furthermore, he was occupied in management and dissemination tasks due to his position as core team leader. In addition, he contributed to the projects of the core team.

Tamás Fehér worked on the GOKE and SOLPSOPT projects.

The Gysela code is a nonlinear global full-f gyrokinetic code that can be used to model turbulence and heat transport in tokamaks close to reactor conditions. These resource-intensive calculations can utilize up to 64k cores in modern supercomputers. The aim of the GOKE project is to optimize kernels of the Gysela code for the Knights Landing (KNL) architecture.
First, a set of benchmarks was performed on both the KNL and Broadwell partitions on Marconi, in order to compare their performance, and to understand how to make best use of the new hardware. The execution time of the Gysela code was measured on the KNL nodes and the results were compared to a Broadwell node on Marconi. The key to decrease the KNL execution time is to improve single core performance, which translates to improving vectorization, and avoiding divisions which can be very costly on KNL. There are two tridiagonal solvers used in Gysela. A vectorized version was implemented for both of them. In one case, it offers a performance improvement of up to 9x on KNL. For the other solver it was even possible to avoid divisions altogether, which further improved the performance. Factoring out divisions improved the execution time of the heat source computation and decreased the total computation time by 8% on KNL. Five subroutines were identified, where the compiler was not able to optimize the code in OpenMP mode. Improving the vectorization solved the problem for one of the subroutines and significantly decreased the execution time of the OpenMP code. To aid the compiler in vectorization, the `contiguous` attribute was added to more than a thousand pointers, which gave a performance improvement of around 6%.
These optimizations decreased the execution time not only on the KNL architecture, but also on the Broadwell and Skylake architectures. There is a 23–25% improvement in the single node computation time on all three architectures.

The SOLPSOPT projects optimized the parallelization of the SOLPS package, which is a collection of several codes. The two main components are the Eirene and the B2

codes. The B2 code is a plasma fluid code to simulate edge plasmas and the Eirene code is a kinetic Monte-Carlo code for describing neutral particles.
The OpenMP parallelization of the SOLPS 5.0 version of the B2 code was merged into the SOLPS-ITER version of the B2 code. Up to 90% parallel fraction was reached for a standalone B2 simulation of the target test case, which is a 98 species ITER scenario. This results in a factor of 6.7 speedup for the standalone B2 code.
The Eirene code in SOLPS-ITER already had an MPI parallelization, and now it is possible to run the coupled B2-Eirene code with hybrid MPI-OpenMP parallelization. The coupled B2-Eirene simulation was tested with different test cases and it achieves a factor of 4-5x speedup.

Kab Seok Kang worked on the GRIMG2 and BIT2-3 projects.

The contribution of the HLST to the GRIMG2 project was the implementation of the matrix generation routine for Neumann boundary conditions using a zero-order approximation. Resulting from this work, GRILLIX can now handle Dirichlet and Neumann boundary conditions on a generically shaped domain. We implemented a matrix generation routine and a matrix-vector multiplication routine employing the ELLPACK format which allows an easy implementation of the Gauss-Seidel smoother. We modified the restriction and prolongation operators as well. In addition, we implemented Jacobi and Gauss-Seidel smoothers. The multigrid solver itself was rewritten with GMRES as a coarsest level solver.
We obtained, for the multigrid algorithm, typical numerical results with the modified code. We found that the required number of smoothing steps for convergence is at least two for both smoothers. Specifically, it is three for the Jacobi smoother and two for the Gauss-Seidel smoother. Numerical results show that the former five smoothing steps are too costly in terms of performance. We also report on the scaling properties of the solver.
We have sent the code to the project coordinator in order for it to be merged with the GRILLIX code. The project coordinator plans to modify the numbering system of the ghost nodes which would result in the sole use of positive numbers, removing negative numbers from the numbering.

A parallel 3D multigrid solver for the Poisson problem for the BIT3 code, using either the finite difference (FDM) or the finite element method (FEM), has been implemented. We have completed the parallel multigrid method with a merging step using a hybrid MPI/OpenMP parallelization scheme. The numerical study of the current implementation shows that the 3D multigrid solver needs, as expected, a fixed number of iterations.
We found that the required number of iterations is identical for two domains with different shapes, one domain was a cube domain and the other one was an elongated rectangular parallelepiped domain. Cube cell elements were used in both cases. We also found that the required number of iterations is the same for problems with different types of boundary conditions, i.e., a problem with Dirichlet boundary conditions everywhere, and a problem with Neumann boundary conditions along one direction, and Dirichlet boundary conditions along all other directions.
On a single Broadwell node, the hybrid MPI/OpenMP parallelization shows no benefit in comparison with the pure MPI parallelization. However, we see a significant performance improvement on more than 1000 cores. On a single KNL node, the pure MPI parallelization has outstanding performance in comparison to the hybrid MPI/OpenMP parallelization. This phenomenon continues up to 8000 cores where we still don't measure any benefit from the hybrid MPI/OpenMP parallelization on a KNL system. The solution times using cache mode are identical to the ones using flat mode on a single KNL node, but on more than two nodes (more than 128 cores) cache mode is slightly faster than flat mode. We showed that in a node to node comparison, the performance on Broadwell and on KNL is similar and we get good scaling properties up to 8000 cores on both systems. In addition, we tested the multigrid solver on the Marconi A3 partition (SKL) and achieved a performance

improvement of about 33% for the same number of nodes compared to the Broadwell (A1) and the KNL (A2) partitions.

Serhiy Mochalskyy worked on the CINCOMP2 (Part 1) and JORSTAR projects.

For the CINCOMP2 project, we analyzed the performance of the Marconi supercomputer which went into operation in July 2016. The first partition of the machine had the code name A1 and was based on the Intel Xeon processor E5-2600 v4 product family (*Broadwell*). The second partition was added at the beginning of 2017 and is named A2. It is equipped with the next-generation of the Intel Xeon Phi product family (*Knights Landing*). In the framework of this project, both the hardware and the software of the A1 and A2 partitions were tested using a variety of benchmarks. As expected for a new supercomputer, many issues were found and subsequently reported to the Marconi support team via the ticket system.
The Stream benchmark showed a very high memory bandwidth for the MCDRAM memory of a KNL node in comparison to a Intel *Broadwell* node. However, the memory bandwidth for DDR4 was slightly lower on KNL in comparison to Intel *Broadwell*. The Intel MPI PingPong test for the intra node benchmarks on the A2 partition has revealed a drastic bandwidth drop for message sizes larger than 3 kB.
For the JORSTAR2 project, different libraries (e.g. ScaLAPACK, HDF5 and MPI) were analyzed in order to find the best possible solution for parallel I/O. Consequently, MPI I/O was implemented in both the STARWALL output and the JOREK input subroutines as it can automatically translate the format of the output submatrices from the block-cycling distribution to the ordinary layout during the writing procedure. The execution time of the complete reading and writing procedure for a production run has been reduced to a few minutes when using 16 and 64 computing nodes.

All sequential subroutines in JOREK, that use distributed matrices from the STARWALL output file, were parallelized and tested. Identical results in comparison to the original code version were achieved.
During the development, a bug was found in the Intel MPI library that significantly limits the size of the read/written data per operation. It was reported to the Intel support team and should be corrected in the next version of the library. In order to be able to work with large matrices and to circumvent this bug, several subroutines in JOREK had to be modified.
The new version of the code was merged with two other code versions: the most recent version from the *develop* branch and another specially tuned one. The final code version provides results which are much faster than the develop version and can work with the very large matrices from the STARWALL output. Thus, it is now possible to run JOREK with a realistic wall resolution for ITER-like cases.

Nils Moschüring worked on the CINCOMP2 (Part 2) project.

One main objective of the CINCOMP2 project is monitoring the performance and reliability of all computing resources available to the EUROfusion community. The monitoring capability was expanded by adding usage tracking of all computing nodes as well as monitoring of the uptime of the file system. After the addition of the KNL (Intel Xeon Phi) resources at the start of 2017, another big upgrade of the available resources at the Marconi supercomputer came in the shape of the Skylake (Intel Xeon 8160) partition near the end of the same year. The previously established benchmarks and techniques were expanded to include the SKL resources as well. A host of problems was detected, reported and subsequently solved. In order to guarantee efficient usage of the KNL partition we reached out to all its users and provided individual optimization advice for several common performance hurdles.
The new SKL partition suffered from an issue with the timing facilities. Investigations revealed that the optimal configuration should use the Time Stamp Counter (TSC) timing facility, a switch from the initially configured High Performance Event Timer (HPET). Continuing from this improved understanding of time measurements a

library was written which aims at properly measuring time differences in massively parallel systems. This library enables the elimination of a systematic measurement anomaly called process skew.
Application of this new library enabled proper investigations into the performance of the I/O system of Marconi as well as an in-depth analysis of the effects of hardware interrupts on the performance. Using a custom benchmarking suite, the I/O system was found to work in an appropriate manner. The hardware interrupt investigation revealed some key insights into the CPU and operating system structure. Apart from enabling the HLST to give users well founded usage advice, these investigations will also help identify problems ahead of time.

Tiago Ribeiro worked on the REFMUL3 project.

Simulation of reflectometry using a Finite-Difference Time-Domain (FDTD) code is one the most popular numerical techniques used. This method requires however a fine spatial grid discretization, which implies a high-resolution time discretization to comply with CFL stability condition. Consequently, three-dimensional simulations necessarily imply the development of a domain decomposed parallel code. The REFMUL3 project aimed precisely at this goal, namely, obtaining a parallel scalable three-dimensional FDTD simulation code with the same name.
The single-core code profiling and optimisation were the first steps taken. From these an overall 1.8x speedup factor resulted for the reference test-case given by the project coordinator. OpenMP threads were then implemented to exploit the parallelism at the node-level, with a 18.6x speedup obtained with 36 threads on a Marconi Broadwell compute node.
The implementation of a three-dimensional domain decomposition and corresponding MPI communication in REFMUL3 proceeded as the major task of the project. After a time consuming code development phase, followed by a debugging phase, the parallel REFMUL3 code was finally ready. It shows very good scaling properties and, compared with the original serial code, runs the reference test-case three orders of magnitude faster, if enough resources are used. Additionally, within the available project time budget, an effort was made to implement a working parallel I/O technique, with minimal changes to the original serial counterpart. For the bigger domains that are now computationally available, especially if they don't fit within the memory of a single compute-node, no straightforward serial I/O technique based on data gathering can work. The devised parallel solution makes this possible by having one HDF5 file written per MPI task.

VIRIATO uses a unique framework to solve a reduced (4D, instead of the usual 5D) version of gyrokinetics applicable to strongly magnetized plasmas. It is pseudo-spectral perpendicular to the magnetic field and uses a spectral representation (Hermite polynomials) to handle the velocity space dependency. The parallel scalability of the resulting algorithm, which uses extensively 2D Fourier transforms, becomes a non-trivial problem. This is the main topic of this proposal, which further extends the work that was carried out in 2015.
The strategy devised exploits the node-level NUMA topology of current HPC machines to reduce the transpose communication complexity, by removing explicit MPI communication inside groups of resources able to share memory. Such pure MPI hybridization was implemented using shared memory windows in the framework of the MPI-3 standard. A completely new transpose method compatible with VIRIATO was devised from scratch. This represented a large portion of the project time budget, but the task was nevertheless successfully finished.
The results obtained show that, even if on paper the pure MPI hybridization is a promising concept well fitted to VIRIATO's parallel FFT algorithm, in reality obtaining good scaling performance is rather dependent on both the hardware and software stack (MPI library), as well as the problem size under consideration. In practice, no advantages could be found for typical VIRIATO problem sizes compared to the original flat-MPI algorithm. For better suited (larger) problem sizes, a sound benefit

was yielded on the Skylake partition of MARCONI, but not on the Intel Knights Landing partition. The latter should however be considered as preliminary findings since not much time was invested in the subject of Many Integrated Cores.
Finally, a potential limitation regarding network bandwidth across compute-nodes was found when larger shared memory windows are used. A simple algorithmic workaround, which could improve its scalability, has been proposed, but remains as a suggestion for future work due to lack to time in the current project.

1.2. *Further tasks and activities of the core team*

1.2.1. **Dissemination**

Tamás Fehér: Experience with the Knights Landing nodes on Marconi, *NMPP Seminar*, 30th May 2017, IPP, Garching, Germany.

Mochalskyy, S. and Hatzky, R.: The experience of the High Level Support Team (HLST), *5th IFERC-CSC Review Meeting*, 22nd March 2017, Rokkasho, Japan.

Moschüring, N. and Hatzky, R.: HLST project CINCOMP3, *Accelerated Computing for Fusion, 28th–29th November 2017*, Maison de la Simulation, Gif-sur-Yvette, France.

1.2.2. **Training**

Tiago Ribeiro has visited:
The project coordinator Filipe da Silva to work for the REFMUL3 project, 30th January–8th February 2017, IPFN-IST, Lisbon, Portugal.

1.2.3. **Internal training**

Tamás Fehér has attended:

- Intel AI Day Munich - The Future of Artificial Intelligence, 1st February 2017, ICM - International Congress Center Munich, Munich, Germany.
- MATLAB EXPO 2017, 27th June 2017, Munich, Germany.

Roman Hatzky has attended:

- Numerical Methods for the Kinetic Equations of Plasma Physics (NumKin2017), 23rd–27th October 2017, IPP, Garching, Germany.

Kab Seok Kang has attended:

- PRACE PATC Course: *Introduction to hybrid programming in HPC*, 12th January 2017, LRZ, Garching, Germany.
- Intel AI Day Munich - The Future of Artificial Intelligence, 1st February 2017, ICM - International Congress Center Munich, Munich, Germany.
- Intel AI Software Workshop for Developers and Data Scientists, 2nd February 2017, ICM - International Congress Center Munich, Munich, Germany.
- Workshop on Numerical Methods for Optimal Control and Inverse problems, 5th–7th April 2017, LRZ, Garching, Germany.
- PRACE PATC Course: *Intel MIC Programming Workshop*, 26th–28th June 2017, LRZ, Garching, Germany.
- Scientific Workshop: HPC for natural hazard assessment and disaster mitigation, 28th–30th June 2017, 2017, LRZ, Garching, Germany.
- Power Aware Computing PACO 2017, 5th–7th July 2017, Schloß Ringberg, Germany.
- Numerical Methods for the Kinetic Equations of Plasma Physics (NumKin2017), 23rd–27th October 2017, IPP, Garching, Germany.
- IPP Theory Meeting, 20th–24th November 2017, Harnack-Haus, Berlin, Germany.
- PRACE PATC Course: *Node-Level Performance Engineering*, 30th November –1st December 2017, LRZ, Garching, Germany.

Serhiy Mochalskyy has attended:

- Lenovo: KNL Workshop, 22nd–23rd March 2017, CINECA, Bolognia, Italy.
- PRACE PATC Course: *Node-Level Performance Engineering*, 30th November –1st December 2017, LRZ, Garching, Germany.

Nils Moschüring has attended:

- PRACE PATC Course: *HPC code optimization workshop*, 4th May 2017, LRZ, Garching, Germany.
- Lenovo: KNL Workshop, 22nd–23rd March 2017, CINECA, Bolognia, Italy.
- PRACE PATC Course: *Intel MIC Programming Workshop*, 26th–28th June 2017, LRZ, Garching, Germany.
- Advanced Fortran Topics, 11th–15th September 2017, LRZ, Garching, Germany.
- PRACE PATC Course: *Node-Level Performance Engineering*, 30th November –1st December 2017, LRZ, Garching, Germany.

Tiago Ribeiro has attended:

- PRACE PATC Course: *Introduction to hybrid programming in HPC*, 12th January 2017, LRZ, Garching, Germany.

1.2.4. **Workshops & conferences**

Da Silva, F., Heuraux, S., and Ribeiro, T.: Introducing REFMULF, a 2D full polarization code and REFMUL3, a 3D parallel full wave Maxwell code, *13th International Reflectometry Workshop for Fusion Plasma Diagnostics (IRW13)*, 10th–12th May 2017, Daejeon, South Korea.

Mochalskyy, S.: The experience of the High Level Support Team (HLST) on the Marconi supercomputer, *Lenovo: KNL Workshop*, 22nd–23rd March 2017, CINECA, Bolognia, Italy.

Moschüring, N., Fehér, T., Mochalskyy, S., and Hatzky, R.: The experience of the HLST on Europe's biggest KNL cluster, *PRACE PATC Course: Intel MIC Programming Workshop*, 26th–28th June 2017, LRZ, Garching, Germany.

Vicente, J., Da Silva, F., Silva, C., Heuraux, S., Ribeiro, T., and Conway, G.D.: Application of 2D full-wave simulations to study plasma turbulence with conventional reflectometry, *13th International Reflectometry Workshop for Fusion Plasma Diagnostics (IRW13)*, 10th–12th May 2017, Daejeon, South Korea.

1.2.5. **Meetings**

Roman Hatzky attended on a regular basis:

- IFERC HPC follow-up working group
- HPC Operation Committee meeting
- EUROfusion MARCONI Ticket Meeting

2. **Final report on HLST project GOKE**

2.1. *Introduction*

The Gysela code is a nonlinear global full-f gyrokinetic code that can be used to model turbulence and heat transport in tokamaks close to reactor conditions. Such calculations are very resource-intensive; therefore efforts are made to utilize accelerators like the Intel MIC architecture or NVIDIA GPGPUs.

Recently a new generation of the Intel MIC architecture, code named Knights Landing (KNL), has been launched, and it is being deployed in HPC clusters. The Marconi supercomputer has also installed a large KNL partition. Therefore the primary optimization target of this project is the KNL architecture.

The aim of the GOKE project is to evaluate and optimize kernels of the Gysela code on different accelerators. After the initial encouraging tests on the KNL architecture, the project coordinator recommended to switch from working with isolated kernels, to working with the actual Gysela code. This increased the complexity of the project, but ensured that the outcome is directly beneficial to the users of the Gysela code.

In this report, the KNL architecture is introduced and the performance of the KNL nodes of Marconi are compared to the KNL nodes of the Frioul cluster and also to the Broadwell nodes of Marconi. After that, we measure the performance of the Gysela code, and optimize those kernels where the KNL execution time is much longer than the Broadwell execution time. The optimized code is tested not only on the KNL and Broadwell partitions, but also on the recently opened Skylake partition of Marconi.

2.2. *The Knights Landing architecture*

The new Knights Landing processors are available as self-bootable processors. Every node of the A2 partition of Marconi has just one KNL processor. The project coordinator has also arranged an account at the Occigen supercomputer at CINES, France, where they have a KNL test cluster called Frioul. Both Marconi and Frioul have the same Xeon Phi 7250 CPUs.

Table 2 shows the basic parameters of the Knights Landing node of Marconi (these parameters are the same for Frioul) and the current Broadwell node of Marconi. Note that both processors have a separate AVX base frequency (used for high AVX workload), which is lower than the default base frequency. On the KNL architecture every core has two Vector Processing Units (VPUs). A single thread can issue a vector instruction for both VPUs in every cycle. Similarly to the KNL architecture, one thread on the Broadwell architecture can use two execution ports and issue two Fused Multiply Add (FMA) instructions per cycle. In principle we can reach peak performance on both architectures by using one thread per core. Multiplying rows 1, 3, 4, and 5 in Table 2 gives the number of arithmetic instructions that can be executed in a second. The peak performance in row 6 is calculated that way, assuming that only FMA instructions are executed (two arithmetic operations per instruction).

2.2.1. **Clustering modes**

The KNL processor has a large number of cores and these cores are grouped into tiles. Every tile contains two cores and their shared L2 cache (see Fig. 1). The L2 caches are kept coherent across all the tiles using a system of distributed “tag directories” (TDs) that store information about the state and location of each cache line. After an L2 cache miss, a request is sent to the tag directory, to retrieve the cache line. The latency of such a request depends on how the memory addresses are mapped to the distributed TD.

There are different operation modes of the KNL processor, for which the mapping between the TD and the memory is different. These so-called clustering modes are

set by the system administrators. Both Marconi and Frioul have chosen the quadrant mode.

		Marconi A2 Node Xeon Phi 7250 (KNL)	Marconi A1 node Xeon E5-2697v4 (Broadwell)
1	Cores	68	2 x 18
2	Frequency	1.4 GHz	2.3 GHz
3	AVX Frequency	1.2 GHz	2.0 GHz
4	Vector register	8 doubles	4 doubles
5	FMA execution units	2 Vector Processing Units / core	2 ports / core
6	Peak TFlop/s	2.61	1.15
7	Power	230 W	2x145 W
8	Memory	96 GB + 16 GB on chip MCDRAM	128 GB
9	Bandwidth	90 GB/s, 490 GB/s (MCDRAM)	119 GB/s

Table 2 Comparison of KNL and Broadwell nodes of Marconi

In quadrant mode, the tiles are divided into four virtual quadrants. The memory addresses belonging to the memory controllers of a specific quadrant are mapped into TDs of the same quadrant. This ensures low latency in case the TD lookup has to be followed by a memory lookup.

There are two other topologies for the clustering mode: hemisphere, where the tiles are divided into two groups; and all-to-all where the memory addresses are uniformly mapped to all TDs.

The so called sub-NUMA clustering (SNC) modes are variations of the quadrant and hemisphere modes. In SNC2 mode the hemispheres are also represented as two NUMA nodes, while in SNC4 mode there are four NUMA nodes defined for the four quadrants. Except for the all-to-all mode (which should not be used for production), all clustering modes show similar performance, with a slight difference in the overall latency.

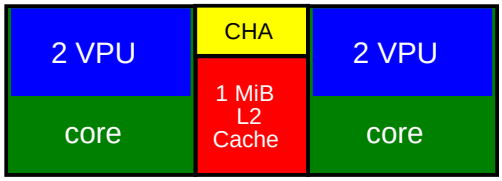


Fig. 1 A KNL tile contains two cores and 1 MiB L2 cache. Each core has two vector processing units.

2.3. *STREAM benchmark*

A KNL node at Marconi has 96 GB of DDR4 system memory and 16 GB of high bandwidth memory integrated on the KNL processor which is called Multi-Channel DRAM (MCDRAM). There are different operating modes for the MCDRAM. The most important modes are: flat mode, where the MCDRAM is available to the user as a separate NUMA node; and cache mode, where the MCDRAM acts as a third level cache.

In flat mode, if an application needs less than 16 GB of memory, it is possible to simply use the `numactl` command to bind the application to the MCDRAM. This would lead to a run-time error if the code tries to use more than 16 GB of memory. One can also use a specific allocator, to control individually where the arrays are allocated, but this requires changes in the source code.

The cache mode is an attractive alternative for applications with high memory usage, because no change is needed in the source code.

The STREAM benchmark (McCalpin 1995) can be used to measure the memory bandwidth. It measures the execution time of the following four kernels:

1. Copy: $c[1:N] = a[1:N]$,
2. Scale: $b[1:N] = scalar \times c[1:N]$,
3. Add: $c[1:N] = a[1:N] + b[1:N]$,
4. Triad: $a[1:N] = b[1:N] + scalar \times c[1:N]$,

where N is a large number. OpenMP is used to parallelize the loops that process the arrays. Knowing the array size and the execution time, the memory bandwidth is calculated for each kernel.

The bandwidth was measured on the KNL nodes of Marconi and Frioul. Both of these clusters use the quadrant clustering mode, and we can choose between flat and cache mode for the high bandwidth memory. In the following sections we show how the bandwidth depends on the number of threads, the array size, and the memory model.

2.3.1. **DDR4 Bandwidth in flat mode**

The STREAM benchmark was executed using different numbers of threads (without hyper-threading). Fig. 2 shows the bandwidth in Flat mode using the regular DDR4 system memory. Already with 12 cores we can reach 95% of the total bandwidth. Adding more cores slowly increases the bandwidth until it becomes saturated. Marconi has a slightly higher peak bandwidth than Frioul.

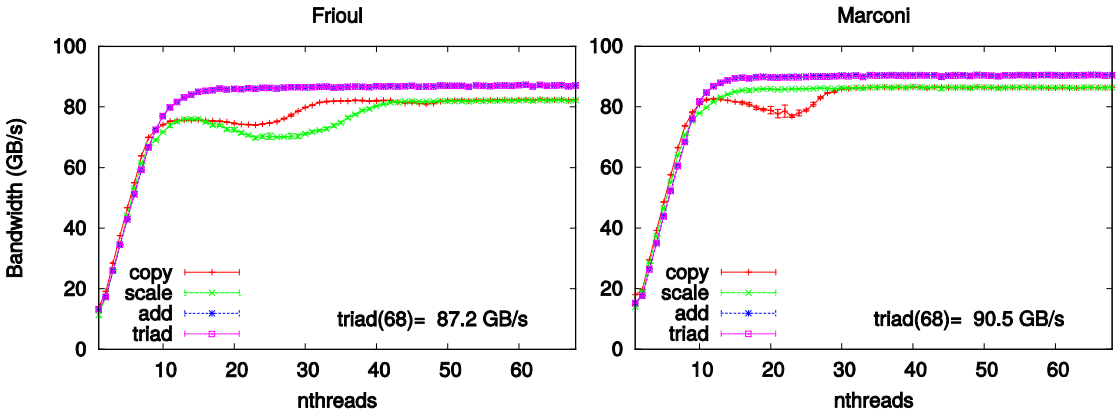


Fig. 2 Flat DDR4: bandwidth vs. number of threads on the KNL nodes of Marconi and Frioul (without hyper-threading).

2.3.2. **MCDRAM Bandwidth in flat mode**

The test was repeated to measure the bandwidth of the MCDRAM in flat mode. The `numactl` command was used to bind the test program to the high bandwidth memory. Surprisingly, the results of the STREAM benchmark were sensitive to the size of the arrays that were streamed. We repeated the test using 68 threads by slightly changing the array size. The array size was kept around 22.5 million elements. It was varied in the interval 22.5 M +/- 2000 elements. The size of the individual arrays was around 171 MiB, which satisfies the rule of thumb that the array size should be four times the size of the last level cache (1 MiB L2 cache per tile gives an aggregate 34 MiB L2 cache for the whole chip). Fig. 3(a) shows that the bandwidth changes significantly as a function of the array size.

2.3.3. **Alignment for MCDRAM**

By changing the size of the arrays, their relative alignment also changes, and it is actually the alignment which was responsible for the fluctuating performance. An article from Intel (Raman 2016) recommends aligning the arrays for the STREAM benchmark on 2 MiB boundaries. We repeated the same test of changing the array size with the aligned version of the STREAM benchmark. The results in Fig. 3(b) show that the bandwidth does not depend on the array size anymore if we align the arrays.

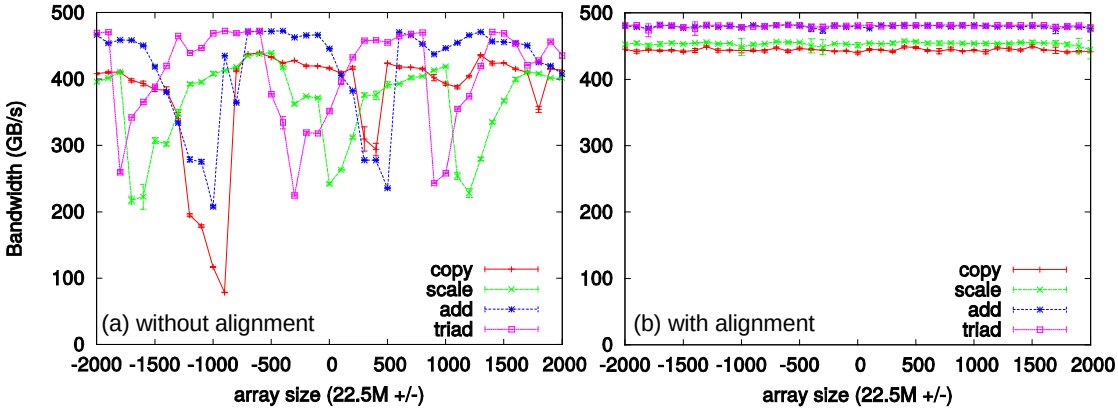


Fig. 3 Flat MCDRAM: bandwidth as a function of array size. The number of array elements was varied in the interval 22.5 million +/- 2000. (a) Results without aligned arrays. (B) Results with aligned arrays.

It is surprising that the alignment has such a drastic effect. We are streaming very large arrays, and an initial misalignment is not expected to have such an impact. Even if we use arrays of 500 million elements (in which case every core has to process 56 MiB array slices), the results without alignment are very sensitive to small changes of the array size. We have contacted Intel to clarify this behavior, but did not receive any explanation.

There are different methods to specify array alignment. Let us consider array `A` (with `N` elements) that we want to align on `M` bytes. When programming in the C language, we can use the following options to specify the alignment:

- static array: `double A[N]__attribute__((aligned(M)));`
- dynamic array: `double *A = aligned_alloc(M, N * sizeof(*A));`

The `aligned` attribute for static arrays is a GNU extension to the C language, which is also supported by Intel and IBM compilers. The `aligned_alloc` function is part of the C standard library (since C11).

The Fortran standard does not specify aligned allocators. In Intel Fortran, one can use the following directive before the array declaration:

- ```
  !DIR$ attributes align : M :: A
  double precision :: A(N)
  ```

This directive works both for static and allocatable arrays. The maximum value of `M` depends on the operating system and the compiler version. On Linux, using Intel Fortran (version 2018), the maximum value of `M` for allocatable arrays is 2097152 (equals to 2 MiB), for static arrays it is limited to 64k. The `-align arrayMbyte` compiler option for Intel Fortran can also be used to specify alignment, but the maximum value of `M` is only 64 bytes in this case.

For other compilers that do not support alignment directives (like GFortran), one can declare an interface to the `aligned_alloc` C function, and use the `C_F_POINTER` intrinsic to convert it to a pointer to a Fortran array.

Fig. 4 shows that at least 64k alignment is necessary to guarantee maximum STREAM bandwidth for all array sizes.
  ```

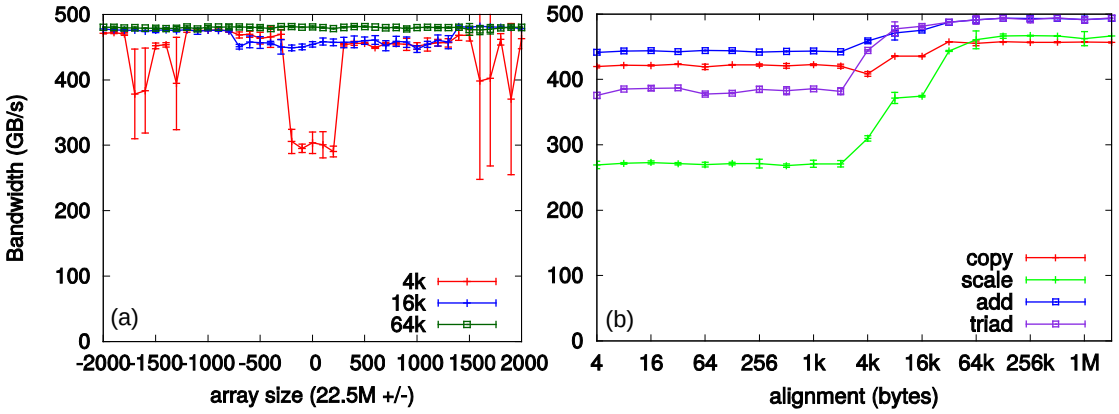


Fig. 4 STREAM benchmark with different alignments: (a) changing array size for the add kernel and using three different alignments, (b) fixed array size and different alignments.

After fixing the alignment, we measured the bandwidth as a function of the number of threads. Fig. 5 shows that both Frioul and Marconi have a similar performance. We should note that the MCDRAM bandwidth per core is similar to the system memory bandwidth, but the MCDRAM is able to serve a larger number of cores before it becomes saturated. This makes the aggregated bandwidth significantly larger than the bandwidth of the system memory. In fact, there is more than five fold increase in bandwidth compared to the system memory (a factor of four compared to the bandwidth of a Broadwell node).

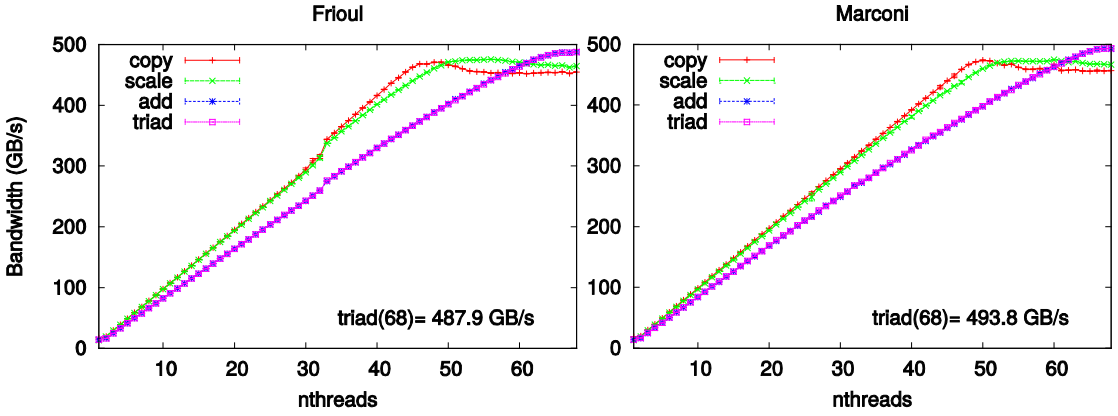


Fig. 5 Flat MCDRAM: bandwidth vs. number of threads, using aligned arrays.

There is a discrepancy in the measured bandwidth between the C and the Fortran version of the STREAM benchmark. The Fortran version of the test delivers around 460 GiB/s in flat MCDRAM mode for the add and triad kernels. This is 30 GiB/s lower than what the C version of the benchmarks achieves. The bandwidth measured using the other memory models (or other kernels in flat MCDRAM) do not differ between the C and Fortran implementations of the benchmark.

2.3.4. **Bandwidth in cache mode**

In cache mode, the finite size of the MCDRAM introduces a dependency of the bandwidth on the array size. In this section we will measure this dependency. While in the previous section we did only a slight change in the array size, here we will change in the interval between 8 MiB and 92 GiB.

In cache mode, if the array size is small enough to fit into the MCDRAM cache, then the array alignment has the same effect as shown in the previous section. We use 2 MiB alignment for the tests in this section to avoid bandwidth degradation due to unfavorable alignment.

The time to stream an array of 32 MB at 500 GB/sec is 64 µs. We have to consider that the resolution of the timer that we use is 1 µs. Using the original STREAM

benchmark would lead to results that are highly fluctuating for small array sizes. Additionally, we should note that creating the OpenMP regions adds an overhead, which is around 6 µs for 68 threads on KNL. The source code was modified to repeat the measurements *M* times, where *M* was chosen in a way that the execution time for each measurement is around 0.5 sec. Additionally, we also measured and subtracted the OpenMP overhead from the execution time of the kernels.

Using the modified benchmark, we measured how the bandwidth of the triad kernel changes as a function of the array size. The number of threads is kept constant (one thread per core). In Fig. 6 we compare the performance of cache mode to flat mode. The *x* axis corresponds to the total size of all the arrays used in the triad kernel. In cache mode we can reach around 2/3 of the bandwidth of the flat mode. Above 16 GiB, the arrays do not fit into the MCDRAM cache, and therefore the bandwidth drops. It is not a sharp drop: there is a wide transition phase as the bandwidth decreases from above 330 GB/s to 59 GB/s.

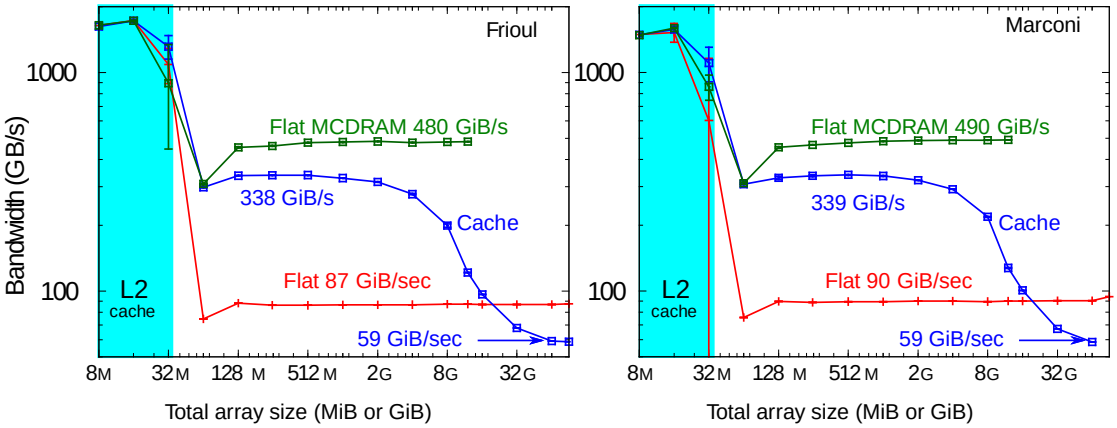


Fig. 6 Triad kernel bandwidth as a function of array size (total size of all arrays) using a modified version of the STREAM benchmark, 68 threads were used for these tests.

2.3.5. **Memory bandwidth using hyper-threading**

In the previous section we have seen that using a single-thread per core, the KNL nodes on Marconi and Frioul have very similar performance, with Marconi showing a slightly higher bandwidth. In this section we present STREAM benchmark results using hyper-threading.

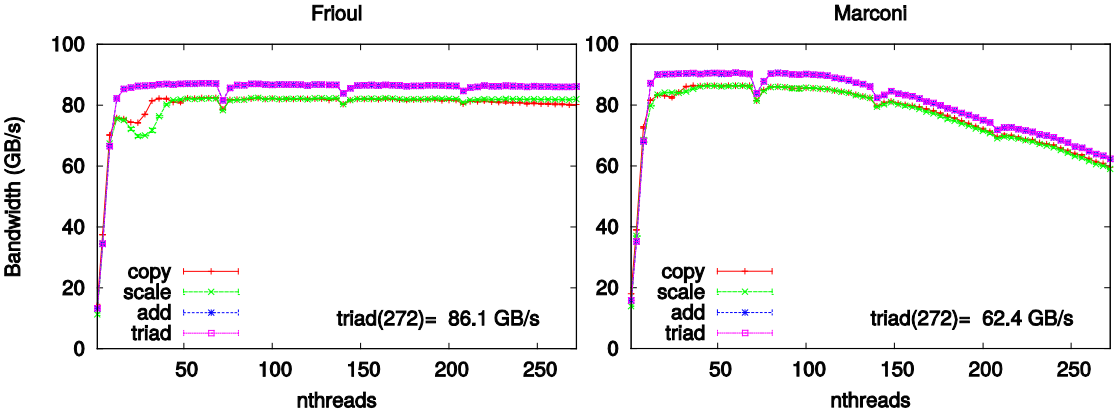


Fig. 7 Flat DDR4 bandwidth vs number of threads using hyper threading and scatter affinity.

Fig. 7. shows the bandwidth in flat mode using the regular DDR4 system memory. An anomaly was detected earlier this year: above 68 threads, where we have more than one hyper-thread per core, the performance on Marconi dropped. The bandwidth on Marconi became 31% lower when all cores hosted four hyper-threads.

The support team of Marconi was informed about this problem, and they fixed it during the maintenance in the beginning of April. After the maintenance the bandwidth remained high in hyper-threading mode (Fig. 8).

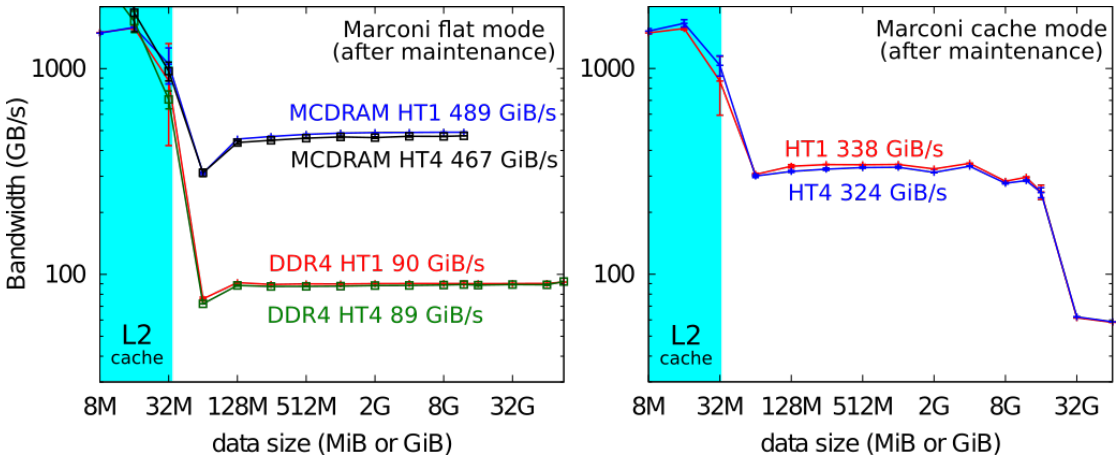


Fig. 8 Bandwidth as a function of array size after maintenance in April. Using one hyper-thread per core (HT1) or four hyper-threads per core (HT4) has only a minimal effect on the achievable bandwidth.

2.3.6. **Latency**

The memory access latency was measured by a pointer chasing benchmark. An array was initialized with pointers that randomly point to different elements in the same array. Following the pointers, we read every element in the array exactly once. The latency can be derived from the execution time of traversing the array. Changing the array size allows us to measure the memory access latency on different levels of the cache hierarchy.

In Fig. 9 we compare the latency between a Broadwell and a KNL node. We should note that the latency of the MCDRAM is comparable to the regular system memory, which is significantly higher than the latency of an L3 cache in the Broadwell architecture. The different memory models on KNL show similar results. We expect that latency bound kernels will not perform well on KNL. Certain subroutines in Gysela have non-trivial memory access pattern and might be affected by this.

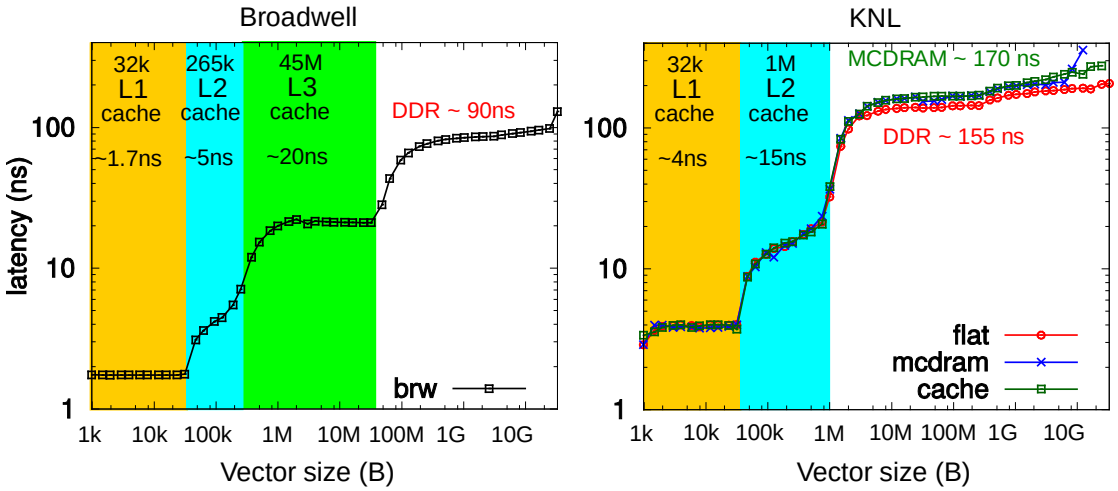


Fig. 9 Memory latency of a Broadwell and a KNL processor as a function of data size.

Others have reported similar trends for the latency, with lower absolute values (McCalpin 2016). Our aim was only a qualitative comparison, and there is still some room for optimization in our latency measurement program. This could explain the differences from measurements published by others.

2.4. *Roofline model*

A real code has several computational kernels. Each kernel can be characterized by its arithmetic intensity, which represents the number of Flops performed for each byte that is transferred between the CPU and the memory. We can use the roofline model (Williams, Waterman and Patterson 2009) to depict how well different kernels would perform on different computer architectures. The model is designed to visualize how the performance of the application is limited by the memory bandwidth and the computing capacity of the CPU cores.

Using the memory bandwidth measured by the STREAM benchmark and the maximum theoretical floating-point performance from Table 2, the roofline model for a KNL node and a Broadwell node (at Marconi) is compared in Fig. 10.

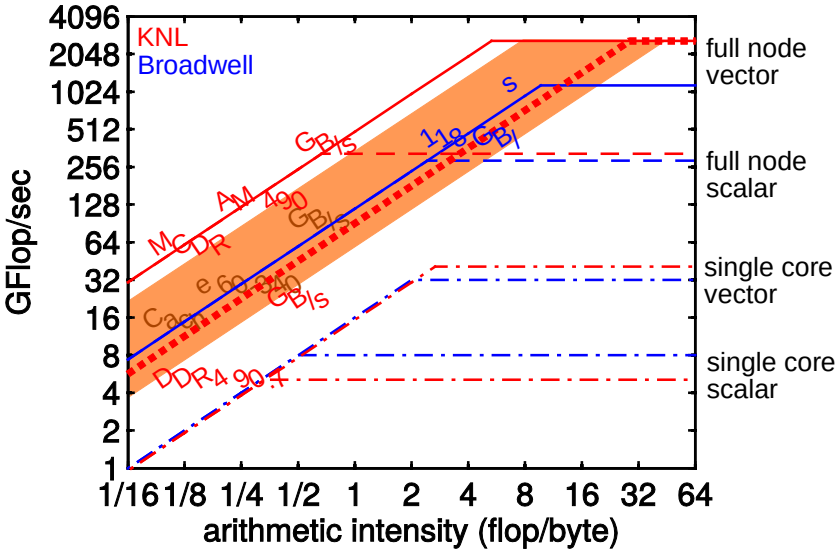


Fig. 10 Roofline model: comparison of a KNL node (red) and a Broadwell node (blue).

The dash-dotted lines compare the single core performance of KNL and Broadwell. The single core scalar performance of KNL is lower because of the lower core frequency, but the wider vector registers make the KNL cores competitive with the Broadwell cores when vectorization is fully exploited. The solid lines give the maximum performance of the node as a function of the arithmetic intensity. There is a factor of 4.1 potential gain in memory speed, and a factor of 2.25 for compute bound applications. For memory bound applications the challenge is to make efficient use of the MCDRAM which has a limited size of 16 GiB. Otherwise the dotted line will be the memory roofline. The hope is that the cache mode can help utilizing this memory without any modifications on the user side.

The dashed horizontal lines denote the peak performance of the node without vectorization. This is similar for both architectures. The dashed red roofline intersects the memory roofline at arithmetic intensity of 0.67 for the KNL node. The vectorization does not matter for computational kernels below this arithmetic intensity; such kernels are memory bound even in scalar mode. On the KNL node, vectorization can become important for kernels that have an arithmetic intensity higher than 0.67 (the exact value depends on the memory model).

To summarize, the KNL node has a higher peak performance, and (using MCDRAM) higher memory bandwidth than the Broadwell node on Marconi. Lack of vectorization, or inefficient use of the MCDRAM might prevent reaching the peak performance, but the roofline model suggests that the KNL and the Broadwell nodes could still have comparable performance in such cases. We should keep in mind that the roofline model is a simplified model, which does not include the latency differences (discussed in the previous section). Additionally, the horizontal rooflines in Fig. 10 are applicable to addition and multiplication operations only. In Section 2.7.1, we will see

that the relative performance of the KNL processor is worse for kernels with division operation.

2.5. *OpenMP benchmarks*

The EPCC OpenMP Microbenchmark Suite (Bull 1999) was used to measure the overhead time of OpenMP constructs on the KNL architecture. Fig. 11 shows how the overhead depends on the number of threads. The overhead on the Marconi KNL nodes were compared to the overhead on different Xeon processor architectures (Marconi A1: Broadwell, Draco: Haswell, Helios: Sandy Bridge).

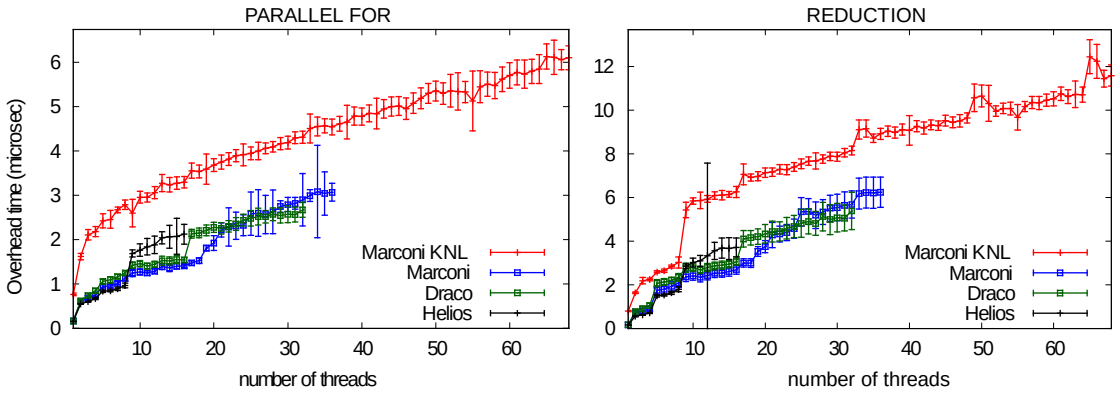


Fig. 11 Overhead time of OpenMP constructs: `PARALLEL FOR` (left) and `REDUCTION` of a single scalar (right). The overhead is measured on Marconi KNL nodes, on the Marconi Broadwell node, on a Haswell node (Draco) and on a Sandy Bridge node on Helios.

Because of the lower clock frequency, the overhead on KNL is generally larger than on a regular Xeon processor. In case of the `PARALLEL FOR` OpenMP construct, the overhead is usually 1–2 μs larger than on Marconi Broadwell, if the same number of threads is used. Using all cores of the node, the overhead becomes 2x larger. The `REDUCTION` construct (for a single scalar variable), has a larger absolute overhead, but the relative difference between Xeon Phi and Xeon is the same.

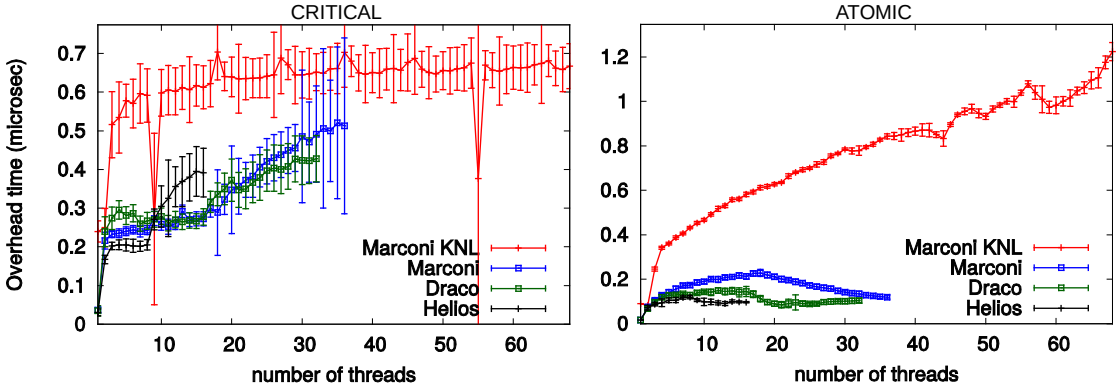


Fig. 12 The overhead time of the OpenMP `CRITICAL and ATOMIC` directives.

The overhead time for the synchronization constructs are in agreement between Marconi and other computers (see Fig. 12). We should note that the `ATOMIC` directive has a relatively large overhead on the KNL (up to 5x larger than on Broadwell).

The two graphs in Fig. 11 are representative for other OpenMP scheduling constructs: the overhead on KNL can be a factor of two larger than on Broadwell if the whole node is utilized with one thread per core (the `ATOMIC` directive is an exception with a larger overhead). In general we can say that the KNL architecture performs reasonably well in the OpenMP overhead measurements.

2.5.1. **Hyper-threading**

One can use up to four hyper-threads per core on the KNL architecture. The curves in Fig. 13 show the overhead of the `PARALLEL FOR` and the `REDUCTION` directives increases in hyper-threading mode. The red curve shows that if we use 68 cores with four hyper-threads, then the overhead is 2.5 times larger than without hyper-threading.

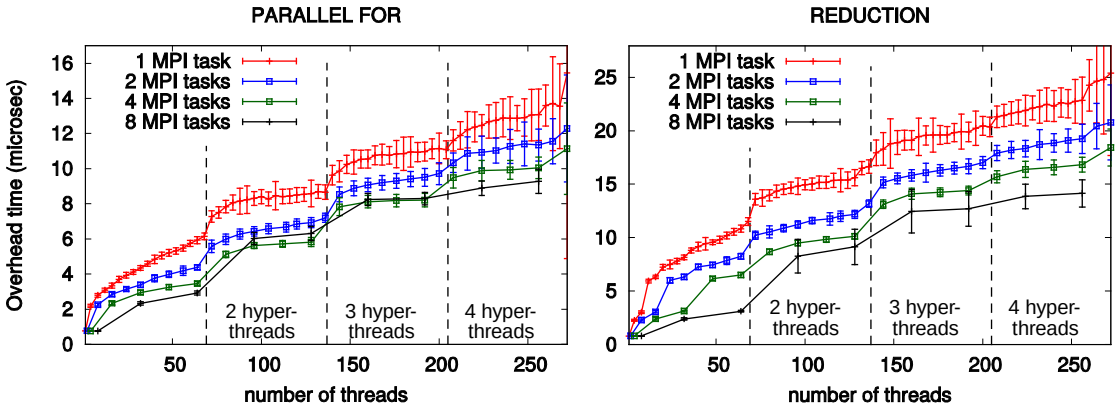


Fig. 13 OpenMP overhead of the `PARALLEL FOR` and `REDUCTION` constructs measured in hyper-threading mode for a hybrid MPI-OpenMP application.

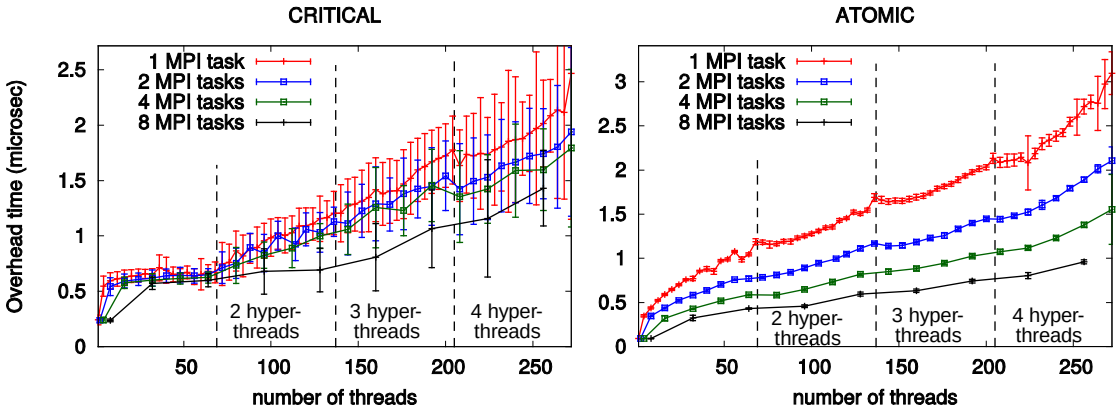


Fig. 14 Overhead of the `CRITICAL` and `ATOMIC` synchronization constructs in hyper-threading mode for a hybrid MPI-OpenMP application.

One way to decrease the OpenMP overhead is to employ hybrid MPI-OpenMP parallelization. The blue, green, and black curves in Fig. 13 show the overhead with different number of MPI tasks. A modified version of the EPCC OpenMP benchmark was used to measure the overhead. The *x* axis denotes the total number of threads (summed over all MPI task). For example, at *x=64*, the green curve corresponds to a benchmark with 4 MPI tasks and 16 threads for each MPI task. OpenMP synchronization is only necessary among threads that belong to the same MPI task. Therefore, if we decrease the number of threads per task, the OpenMP overhead decreases. The overhead of the `CRITICAL` and `ATOMIC` synchronization constructs in hyper-threading mode show a similar trend (Fig. 14).

2.6. *Gysela KNL performance*

After comparing the basic characteristic of the available hardware, we are ready to investigate the actual performance of the Gysela code. The Gysela code is a hybrid MPI-OpenMP code which can be used to simulate gyrokinetic turbulence. In this section, we measure the performance of the whole code on the Broadwell and KNL architectures.

2.6.1. **Node to node comparison**

We started by comparing the single node performance of Gysela on different platforms. The main parameters for our test case were *Nr*=127, *Ntheta*=256, *Nphi*=64, *Nvpar*=63, *Nmu*=0. Domain decomposition is used in the *r* and *theta* direction, to distribute the work among four MPI tasks. On the Broadwell node each MPI task had 8 threads, while on the KNL node each MPI task had 16 threads. The execution time is compared in Fig. 15. The execution time of this test on the KNL node is 50% longer than on Broadwell. The cache and flat MCDRAM modes have approximately the same execution time on the KNL node. The execution time in flat DDR4 mode is around 15% longer than in cache mode.

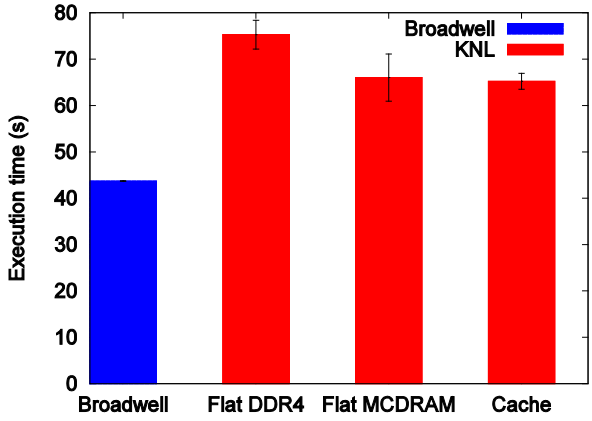


Fig. 15 Gysela single node performance on Marconi.

2.6.2. **OpenMP scaling**

Even though our test on KNL used more threads than our test on Broadwell, we do not expect that the OpenMP overhead is responsible for the performance difference. To confirm this, we measured OpenMP scaling for a relatively small test case (see Fig. 16). Up to 32 cores, the speedup is close to the theoretical maximum given by Amdahl's law. Above 32, we measure a larger fluctuation in the performance. These fluctuations will be investigated later. Even in hyper-threading mode, we will only use a maximum of 32 threads per task; therefore these fluctuations are not expected to influence the performance. Repeating the test with different grid sizes results in similar OpenMP scaling, slightly farther away from the theoretical maximum, but still having an adequate speedup.

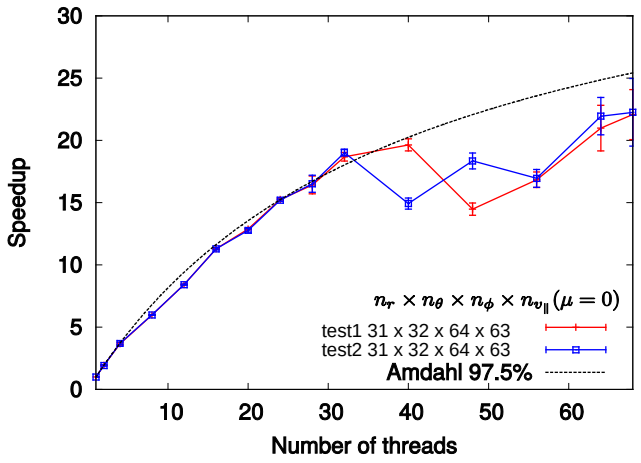


Fig. 16 OpenMP scaling within a KNL node (only 1 MPI task). Each point on each curve was measured on a different node. Within a node, the test was repeated five times, and the error bars show the standard deviation of these repetitions.

2.6.3. **MPI scaling**

The scaling of the hybrid MPI-OpenMP parallelization was also tested on KNL. We used 16 OpenMP threads per MPI task, and varied the number of MPI tasks (while keeping the problem size constant). Fig. 17 shows the achieved speedup (compared to the test case with a single MPI task). Up to four MPI tasks (using one KNL node) the MPI scaling is perfect, and it remains good when multiple nodes are used. This test indicates that the performance difference between KNL and Broadwell is not caused by any problems in the hybrid parallelization of Gysela. Therefore, in the next section we will investigate the single-core performance.

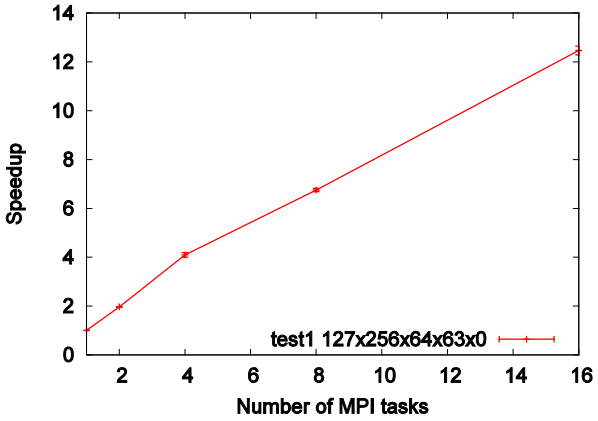


Fig. 17 MPI scaling of Gysela (4 MPI tasks per node, 16 OpenMP threads per MPI task).

2.6.4. **Single-core performance**

A test case was prepared to test the single core performance of Gysela. Fig. 18 shows the execution time of different parts of Gysela. The execution times of different code parts were provided using Gysela's own instrumentation framework. On all three platforms it is the same set of subroutines that takes most of the execution time, but the relative shares of individual subroutines are different between KNL and Broadwell. The KNL nodes on Frioul and Marconi show similar performance.

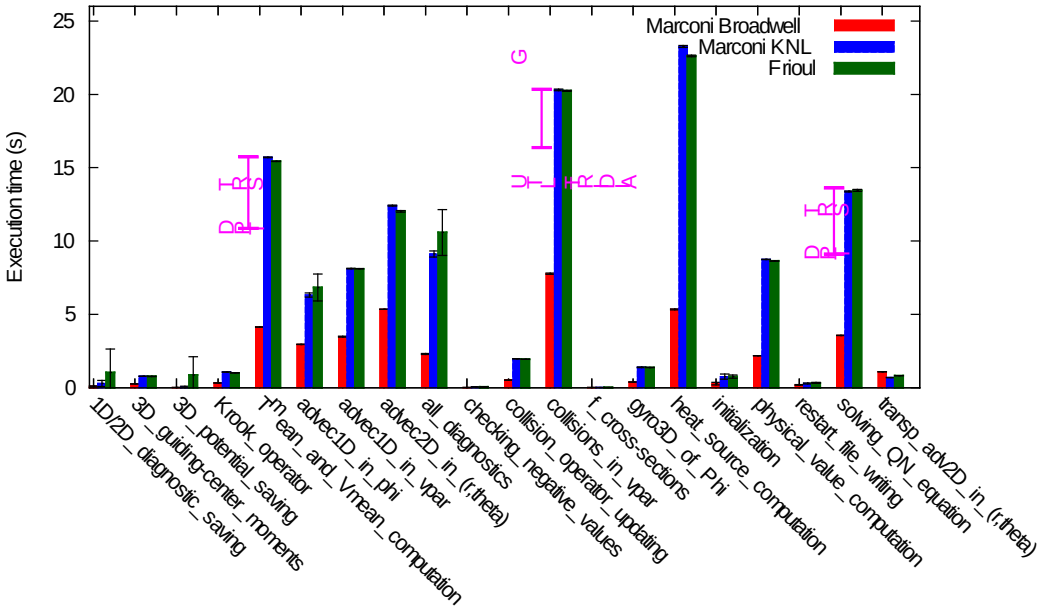


Fig. 18 Execution time of different parts of Gysela (test case size: *nr*=63, *ntheta*=64, *nphi*=32, *nvpar*=31, *nmu*=0). The tridiagonal solvers DPTTRS and UTL_TRIDIAG have a long execution time on KNL. The purple marks show the time difference of these solvers between KNL and Broadwell.

For scalar code, the roofline model suggests that a KNL core should have around 60% longer execution time than a Broadwell core. But some parts of the code take at least two times longer, in certain cases more than four times longer on KNL in comparison to Broadwell. The Broadwell microarchitecture is more advanced than the KNL microarchitecture, and the roofline model does not take the details of the microarchitecture into account. To understand the observed performance differences, we have to study the individual subroutines in more detail.

2.7. *Gysela kernels*

The measurements presented in Fig. 18 can be used to identify the kernels that take the longest time to execute on KNL. In this section we describe the optimization of some of these kernels.

2.7.1. **Tridiagonal solvers**

During single core execution on KNL, 14% of the execution time is spent in solving tridiagonal systems. The Gysela code contains two tridiagonal solvers: `UTL_TRIDIAG` and `DPTTRS`. The first one solves a tridiagonal system using the Thomas algorithm (simplified Gauss elimination), and `DPTTRS` is a Lapack (Anderson, et al. 1999) subroutine that solves symmetric tridiagonal systems. These subroutines contribute significantly to the KNL overhead in three regions (see the purple marks in Fig. 18).

The execution time of `UTL_TRIDIAG` using different compiler options is shown in Fig. 19. Comparing the first two bars, we can see that the execution time on KNL is three times longer than on Broadwell. The center (-no-vec) bar shows the execution time on KNL without vectorization, which is almost the same as the execution time with vectorization. This is to be expected, because the loops in the algorithm have a vector dependency, and therefore they are not suitable for vectorization. For the last two bars, the precision of floating point arithmetic was changed, which influences the accuracy of divisions. The yellow bar (11 bit) is the execution time for the case when the code is compiled with the following options: `-no-prec-div -fp-model fast=2 -fimf-accuracy-bits=11`, while the last one is the execution time with the `–fp-model precise` compiler option.

To understand the measured values, we need to take a detailed look at the algorithm. There is a forward elimination and a back substitution that go through the rows of the matrix. To solve a system with a vector size N, we have to perform $3\times (N-1)$ FMA operations and $2\times N-1$ divisions. Table 3 lists the latency and the throughput of these operations on the two processor architectures. Every loop iteration depends on values calculated in the previous iteration, therefore we have to wait for the whole latency of the instructions. The latency of a division can be 5–7 times higher than the latency of an FMA operation on KNL, which is why the divisions determine the overall execution time.

To remedy this situation, the KNL architecture offers fast instructions to calculate approximate divisions. The `VRCP28SD` instruction (listed in Table 3), for example, approximates the division with 2^{-28} relative accuracy. The yellow bar (11 bit) in Fig. 19 uses such an approximated division operation and that improves the execution time significantly. Unfortunately, this precision is not sufficient for our simulations.

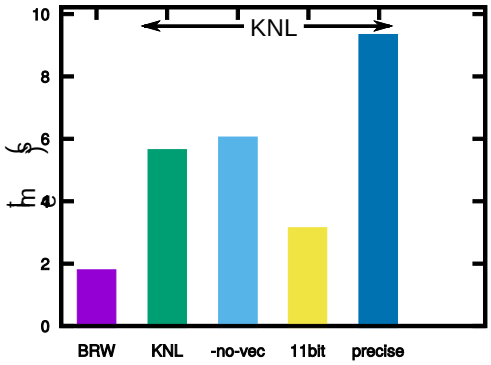


Fig. 19 Single core execution time of the `UTL_TRIDIAG` subroutine (tridiagonal solver) using different compilation parameters.

		Broadwell		KNL		
count	instruction	latency	inverse throughput	latency	inverse throughput	T_{KNL}/T_{BRW}
3(N-1)	FMA	5	0.5	6	0.5	2
0 or 2N-1	MULSD/MULPD	3	0.5	6	0.5	3.3
2N-1 or N	VDIVSD	10-14	4-5	42	42	5
	VDIVPD	19-23	16	32	32	3
	VRCP28SD	n/a		7	2	

Table 3 Latency and throughput (in cycles) of different instructions on the Broadwell and KNL architectures. The inverse throughput signifies the number of cycles we have to wait between issuing two instructions of the same kind (0.5 means we can issue two instruction per cycle). The first column gives the number of operations that are needed for solving a tridiagonal system with vector size *N*. The instructions are FMA: Fused Multiply-Add, MULSD: scalar multiplication, MULPD: vector multiplication, VDIVSD: scalar division, VDIVPD vector division, and VRCP28SD: approximate division. The last column shows the ratio of the execution time of a single instruction on KNL and Broadwell considering the different latencies as well as the different CPU frequencies (T=latency/frequency). Sources: (Intel Corporation 2016) and (Fog 2017).

Using iterative refinement after the approximate division, the result can be improved, and can reach double precision accuracy with a high throughput. This optimization is enabled by default on KNL. Unfortunately, the latency remains high, which results in the long execution time.

We should note that using the `-no-prec-div` compiler flag, the division *x/y* can be calculated as a multiplication with the reciprocal: *x*(1/y).* This is advantageous if more than one division is performed by the same number. Using this option, we can trade half of the divisions in `UTL_TRIDIAG` for additional multiplications. But those divisions can be performed in parallel with the other half of the divisions, so we do not gain much. To summarize, investigations into the architectural differences explains why the UTL_TRIDIAG solver is three time slower on KNL.

2.7.2. **Vectorized UTL_TRIDIAG**

For a single matrix, it is not possible to improve the performance of the subroutine, but Gysela is solving several independent tridiagonal systems at the same time, and in that case vectorization is possible. Such a method was already successfully applied in a tokamak equilibrium solver (Preuss, Fischer and Rampp 2012).

The original version of UTL_TRIDIAG solves the linear system $M_k\, x_k = b_k$, where $k \in [1, \ldots, N_{eq}]$ is the index of the equation. The matrix $M_k = M_k(A_k, B_k, C_k)$ is defined by its diagonals A_k, B_k, and C_k. The original UTL_TRIDIAG solver takes the vectors A_k, B_k, C_k, b_k and calculates the solution vector x_k for each k individually.

To vectorize the solver, we first pack all the A_k, B_k,… vectors into 2D arrays, and pass these 2D arrays to the solver, to solve all the equations concurrently. After these changes, the solver can be vectorized over the independent index k. Additionally, a conditional statement was factored out from the main loop of the solver, which further improved the performance on KNL.

The execution time of the modified solver is shown in Fig. 20, measured during single core execution. Apart from the Broadwell and KNL partitions, the performance of the subroutine was also measured on the Skylake partition of Marconi, which was made accessible in August 2017. The vectorized subroutine is 3.3x faster on Broadwell, 9.1x faster on KNL and 6.0x faster on Skylake. This saves overall 3% to 5% of the total execution time.

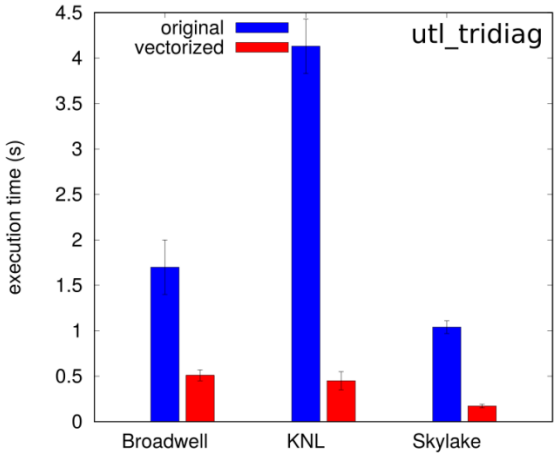


Fig. 20 Execution time of the original and the vectorized UTL_TRIDIAG subroutine (single core execution).

2.7.3. **Vectorized DPTTRS solver**

Let us consider the other tridiagonal solver that is used in Gysela. The DPTTRS subroutine solves a tridiagonal system, where the coefficient matrix *M* is factorized into the product of a diagonal *D* and a lower unit bidiagonal *L* matrix: *M=LDL'* (this can be considered a special case of LU decomposition).

The execution time of DPTTRS is around five times longer on KNL than on Broadwell. DPTTRS is just a driver subroutine that calls DPTTS2 which does the actual work. This subroutine is actually capable of solving for multiple right hand sides simultaneously, but the original code always calls it with a single right hand side.

To speed up the solver, the call chain that leads to DPTTRS (Fig. 21) was modified to be able to pass multiple right hand sides to the solver. Additionally, the arrays were transposed to allow unit stride memory access in the vectorized loop. These changes brought 40% speedup on Broadwell, and a factor of 5 speedup on KNL.

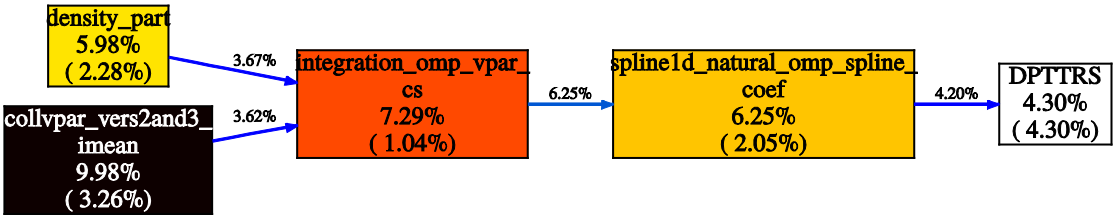


Fig. 21 Call path to DPTTRS. The boxes represent subroutines, the arrows denote subroutine calls. The numbers below the subroutines show the relative share of the (single core) execution time: total time, and self time without children (in parenthesis).

In the DPTTS2 solver, coefficient vector *D* appears as a divisor. The value of *D* is defined during spline initialization, and does not change afterwards. Since the solver

is called from triple nested loops, it is advantageous to calculate the reciprocal of *D* in advance, and replace the divisions with multiplications by the reciprocal. This way we can avoid divisions in the DPTTS2 kernel (in contrast, the coefficient matrix for the UTL_TRIDIAG subroutine changes at every iteration, therefore we cannot avoid divisions there). Without the divisions, the cost of the DPTTS2 solver reduces significantly, and we can save around 4% of the total execution time (see Fig. 22).

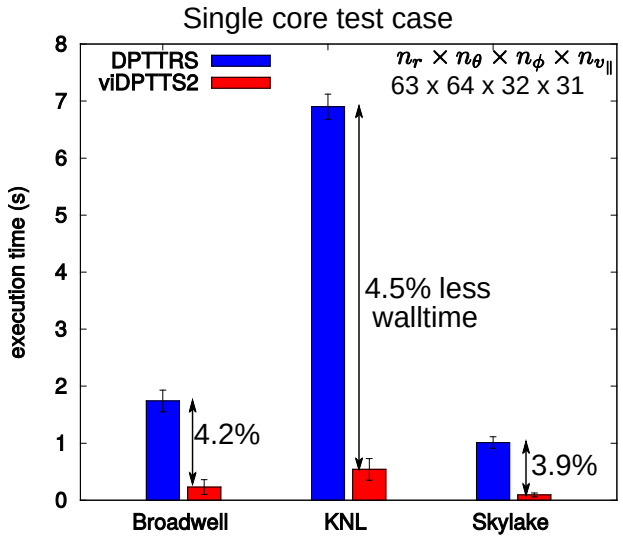


Fig. 22 Execution time of the original DPTTRS subroutine and its optimized version (viDPTTS2). Walltime reductions are given in relation to the execution time of the complete Gysela code.

2.7.4. **Heat source computation**

According to Fig. 18, the largest share of KNL single-core execution time is spent in the heat source computation, which is implemented in the SOURCES_ALL_ADVANCE subroutine. The execution time is around four times longer on KNL than on Broadwell. The SOURCES_ALL_ADVANCE subroutine calls several small (inlined) procedures, to compute different physical quantities. A call graph of these subroutines is shown in Fig. 23. The execution time is limited by the slow division operations in four of these subroutines (middle row of Fig. 23).

The divisions appear inside quadruple nested loops, but the divisors are independent of the phi index (see Fig. 24 left column). This allows us to precalculate these values and reuse them during subsequent iterations in phi direction. Three 2D buffers (*inv_JB, inv_Bij, inv_Bstar_ijl*) have been defined in order to store the reciprocal of quantities that appear as divisors (Fig. 24 right column), and a modified version of the compute subroutines have been introduced that use these arrays. The red bars in Fig. 23 show that the execution time of the optimized subroutines is significantly smaller.

Fig. 25 shows the execution time of the SOURCES_ALL_ADVANCE subroutine, measured during a single core execution of the Gysela code (test case size: *nr*=63, *ntheta*=64, *nphi*=32, *nvpar*=31, *nmu*=0). Due to differences in interprocedural optimization, the results are slightly different than in Fig. 23, which was measured in a test program that only called the SOURCES_ALL_ADVANCE subroutine, but the overall execution time is still significantly improved for the KNL architecture.

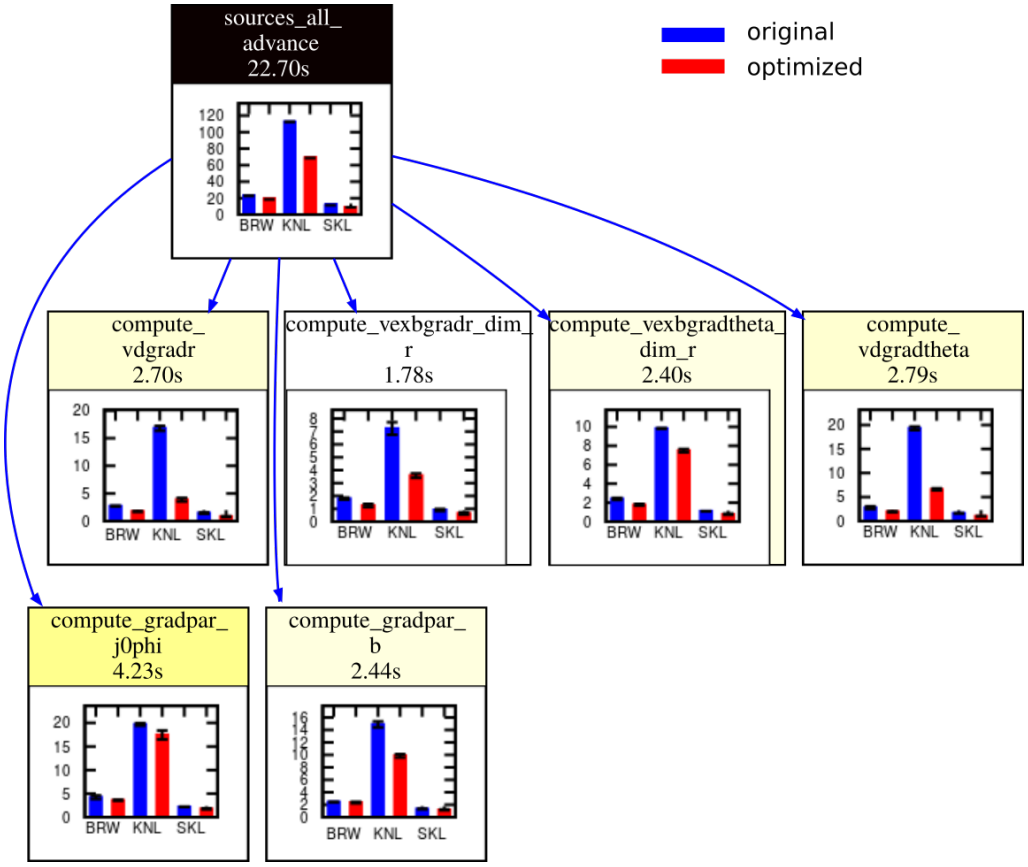


Fig. 23 Call graph of the SOURCES_ALL_ADVANCE subroutine. The bars show the execution time (in seconds) of the code measured on different architectures, blue bars for the original code, and red bars for the code with precalculated divisions. The execution time was measured in a separate test program, using one core. The original version of the subroutines in the middle row contains division operations. The number below the subroutine name shows the execution time of the original code on the Broadwell node.

Original algorithm	Optimized algorithm

`do ivpar = 0, n4`
`  do iphi = 0, n3`
`  do itheta = jstart, jend`
`  do ir = istart, iend`
`  ! do calculation`
`  ! divide with`
`  ! B_ij(ir,itheta),`
`  ! JB(ir,itheta), and`
`  !B*_ijl(ir,itheta,ivpar)`
`  end do`
`  end do`
`  end do`
`end do`
	`do ivpar = 0, n4`
	`  do itheta = jstart, jend`
	`  do ir = istart, iend`
	`  ! precalculate reciprocals`
	`  inv_Bij(ir,itheta)=1/Bij(ir,itheta)`
	`  inv_JB(ir,itheta) = 1/JB(ir,itheta)`
	`  inv_B*_ijl(ir,itheta)=`
	`  1/B*_ijl(ir,itheta,ivpar)`
	`  end do`
	`  end do`
	`  do iphi = 0, n3`
	`  do itheta = jstart, jend`
	`  do ir = istart, iend`
	`  ! do calculation without divisions`
	`  ! use the inv_ variables`
	`  end do`
	`  end do`
	`  end do`
	`end do`

Fig. 24 The original and the modified algorithm of the SOURCES_ALL_ADVANCE subroutine.

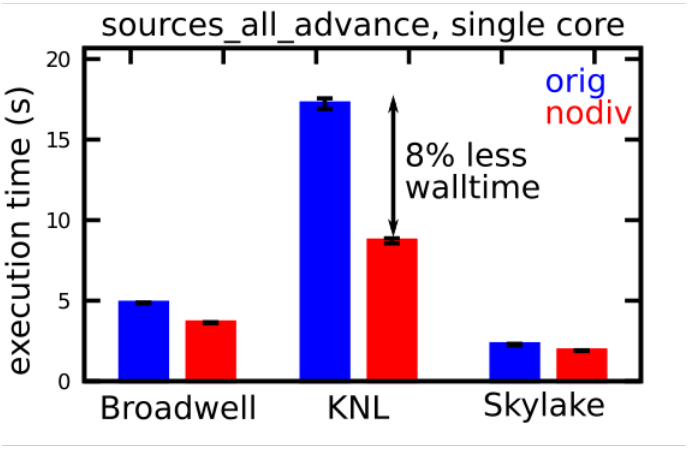


Fig. 25 Single core execution time of the SOURCES_ALL_ADVANCE subroutine. The walltime reduction is given in relation to the execution time of the complete Gysela code.

2.7.5. **Contiguous pointers**

During the optimization of the previous kernels, it was noticed that the compiler is often not able to generate vector load instructions for pointer variables. This results in poor performance. In many cases, the optimization logs highlight this problem with messages like: "non-unit strided load was emulated for the variable".

This situation can be improved by adding the `contiguous` attribute to the pointers, which is defined in the Fortran 2008 standard. In several places, this attribute is already present, for example in the `omputils.F90` file, which defines buffers that are used in OpenMP regions. But often the contiguous arrays are associated with pointers that are defined without the `contiguous` attribute, like in Fig. 26.

```
use omputils_mod, only: OMPr1_0Ntheta ! defined as contiguous
real(F64), dimension(:), pointer :: A
A => OMPr1_0Ntheta(tid)%val
! do calculations with A
```

Fig. 26 Example usage of pointer A without the `contiguous` attribute

In such a case, the compiler does not necessarily understand that pointer A is contiguous as well. One has to explicitly add the `contiguous` attribute to A.

Fig. 27 shows the execution time of Gysela, before and after adding the missing `contiguous` attributes to all pointers (there are more than a thousand pointers that have been changed, most of this work was done automatically with the `sed` and `grep` linux commands).

The bars show the execution time of Gysela on a single node using 1 thread. The code is executed 4 times on 5 different nodes (20 runs in total for each bar), and the bars show the average of the total time without initialization and diagnostics. The error bars depict the standard deviation. The left figure shows the execution time in serial mode (compiled without `-qopenmp`), and the right in OpenMP mode.

The red bars correspond to the case, where the `contiguous` attribute was added only in the `globals.F90` file. This already gives a few percent speedup on all three CPU architectures. The green bars show the execution time after adding the `contiguous` attribute to all other pointers. The green bars are 12% faster in serial mode, and 6% faster in OpenMP mode (on all three architectures).

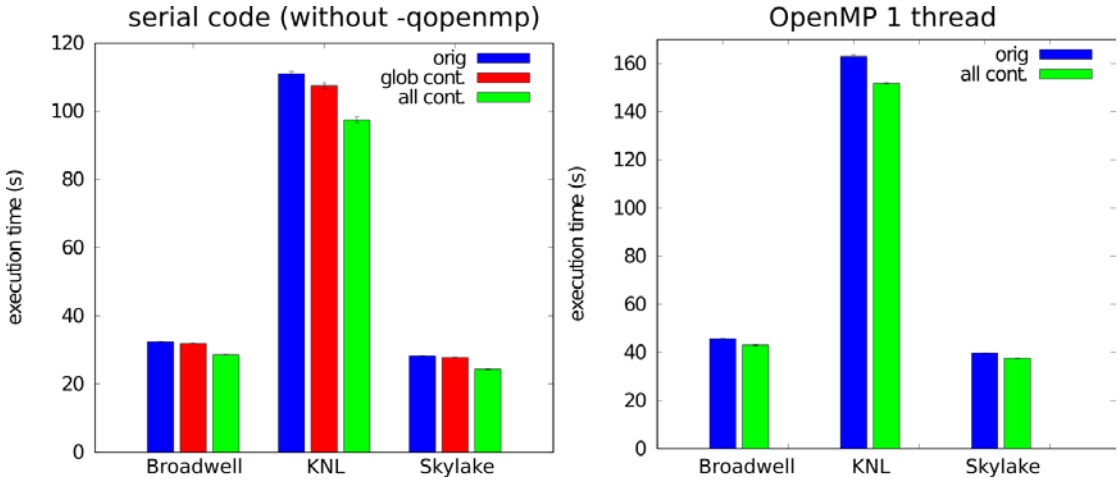


Fig. 27 Gysela single core execution time with additional contiguous directives. Test case size is *nr*=63, *ntheta*=64, *nphi*=32, *nvpar*=31, *nmu*=0.

2.7.6. **OpenMP compilation**

Comparing the left and right hand sides of Fig. 27 reveals that the executable compiled with the `-qopenmp` option is 30 to 50% slower than the executable without the `-qopenmp` flag (note the different scale). It turns out, that some of the subroutines are not optimized properly in OpenMP mode. Table 4 lists the subroutines which are responsible for most of these differences. The almost 14 seconds overhead in the listed five subroutines corresponds to 49% of the total serial execution time.

The largest chunk of these differences stems from the subroutine `COLLVPAR_VERS2AND3_DPARNUPAR`. This subroutine processes first the even, and later the odd elements of an array, which results in a memory access with stride. The exponential and error functions are called for these elements. When the code was compiled without the `-qopenmp` flag, the compiler could vectorize the loop, and could use the fast vector versions of the transcendent functions from the MKL library.

When the code was compiled with the `-qopenmp` flag, the compiler was not able to vectorize the code so efficiently, and used the less efficient implementation of the transcendent functions from the standard math library. This resulted in a 3x longer execution time.

	time (sec)		
name	**serial**	**OpenMP**	**difference**
collvpar_vers2and3_dparnupar	3,07	9,79	6,72
bsl_1d_vpar_lag_advec	3,49	5,65	2,16
bsl_1d_phi_lag_advec	2,53	5,11	2,58
collvpar_vers2and3_imean	2,53	4,05	1,52
density_part	1,36	2,37	1,01
Total	**12,98**	**26,98**	**13,99**
Relative to serial time (29 sec):	45%		49%

Table 4 Execution time (on Broadwell) of subroutines with large OpenMP overhead

The array that was processed with a stride, was also constructed in the same subroutine. By changing the array construction, we can avoid strided memory access in the following computation loops. This way the compiler is able to vectorize even in OpenMP mode, and uses the fast vector math functions from MKL.

The execution time of the original and the modified subroutine was measured both in serial and in OpenMP mode, and the results are shown in Fig. 28. While the original

code (blue bars) has poor OpenMP performance on Broadwell and Skylake, the optimized code (red bars) does not have this problem.

The bars in the middle of Fig. 28 indicate that the OpenMP performance of the original code was fine on KNL. This is misleading, the profiler has attributed the extra overhead of the original code to the caller subroutine. Taking this into account, one can see similar improvement on KNL as on the other architectures.

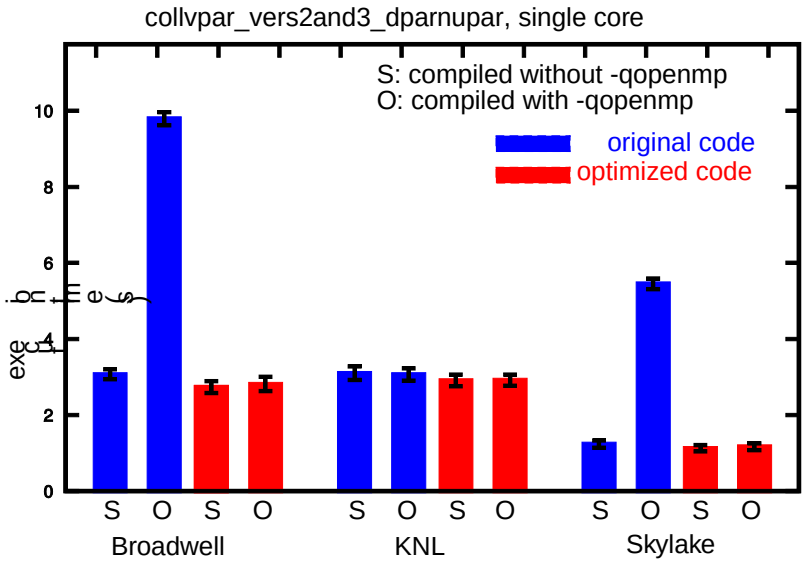


Fig. 28 The large OpenMP overhead (difference between the S and O blue bars) disappears in the optimized version (red bars) of the `COLLVPAR_VERS2AND3_DPARNUPAR` subroutine.

Unfortunately, within the timeframe of the GOKE project, it was not possible to optimize the other four subroutines listed in Table 4.

2.8. *Gysela speedup*

In the previous section, the performance improvements of individual subroutines were discussed. The cumulative effect of all these changes is bars show the total computational time (without initialization and diagnostics) before and after the optimizations. The left hand side of the figure measures the single core execution time using the same small test case as in the previous sections. For such simulation parameters, there is a 40% speedup on the KNL architecture, and around 30% on the Broadwell and Skylake CPU architectures.

Fig. 29b shows the execution time of a test case, where the full node is utilized using hybrid MPI-OpenMP parallelization. The optimized code is 23–25% faster for this parallel test case. The best configuration for KNL was cache mode and two hyper-threads per core.

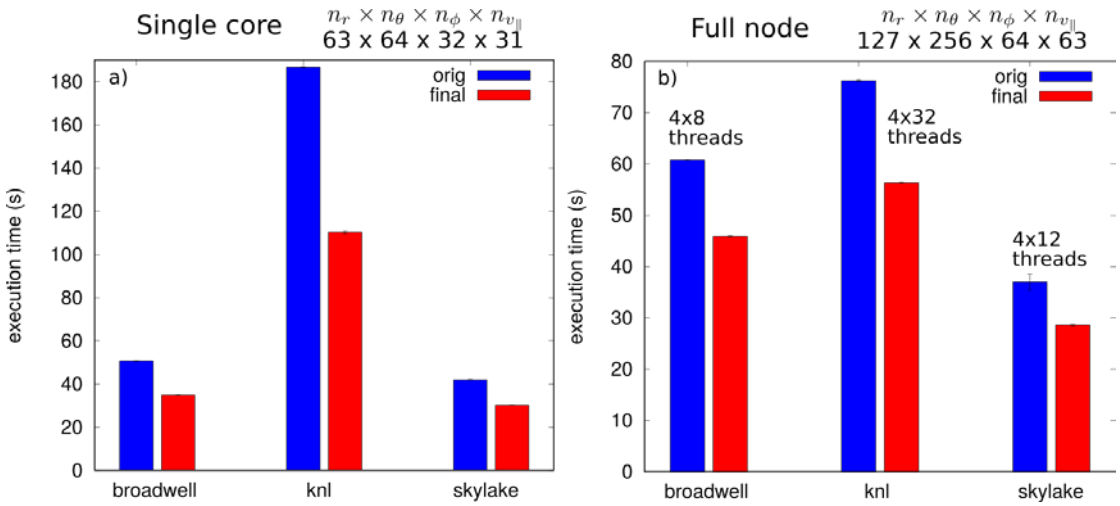


Fig. 29 Overall performance improvements in two test cases: a) small single core test, b) larger full node test case. Blue bars: original code, red bars: optimized code.

2.9. ***Summary***

The Gysela code is a nonlinear global full-f gyrokinetic code that can be used to model turbulence and heat transport in tokamaks close to reactor conditions. The aim of the GOKE project is to optimize kernels of the Gysela code for the Knights Landing (KNL) architecture. The KNL processor is more powerful and easier to use than its predecessor, the KNC coprocessor. This allowed us to extend the focus of the project from working with isolated kernels to working with the full Gysela code.

First, a set of benchmarks was performed on both the KNL and Broadwell partitions on Marconi, to compare their performance, and understand how to make best use of the new hardware.

The STREAM benchmark was used to measure the memory bandwidth of the KNL nodes on the Marconi A2 partition and on the Frioul cluster. In flat mode, using DDR4, we measured a bandwidth of 90 GiB/s.

In flat mode, using MCDRAM, the measured bandwidth depends strongly on the array alignment. With the arrays aligned at 2 MiB, we can reach a bandwidth of 490 GiB/s. In cache mode we can achieve around 2/3 of the peak performance, which is 330 GB/s for arrays smaller than 2 GiB, and 59 GiB/s for arrays that fill the system memory. There is a wide transition phase in between.

A set of OpenMP benchmarks was performed in order to test the overhead of different OpenMP constructs. The overhead time increases by a factor of two on the KNL architecture because of the reduced clock frequency and the larger number of threads.

The roofline model was used to draw a general comparison between the new Knights Landing nodes and the Broadwell nodes of Marconi. Theoretically, the KNL nodes can offer a performance gain up to a factor of two and four for compute and memory bound applications, respectively. In practice these numbers are difficult to reach, but the KNL node can still deliver comparable results to the Broadwell nodes.

The execution time of the Gysela code was measured on the KNL nodes at Frioul and Marconi, and the results were compared to a Broadwell node on Marconi. The wall-clock time for the numerical calculation is 50% longer on KNL for the studied test case. The key to decrease the KNL execution time is to improve single core performance, which translates to improving vectorization, and avoiding divisions which can be very costly on KNL.

There are two tridiagonal solvers used in Gysela, a vectorized version was implemented for both of them. In one case, it offers up to 9x performance

improvement on KNL. For the other solver it was even possible to avoid divisions altogether, which further improved the performance.

Factoring out divisions improved the execution time of the heat source computation and decreased the total computation time by 8% on KNL.

Five subroutines were identified, where the compiler is not able to optimize the code in OpenMP mode. Improving the vectorization solved the problem for one of the subroutines and significantly decreased the execution time of the OpenMP code.

To aid the compiler in vectorization, the `contiguous` attribute was added to more than a thousand pointers, which gave around 6% performance improvement.

These optimizations decreased the execution time not only on the KNL architecture, but also on the Broadwell and Skylake architectures. There is a 23–25% improvement in the single node computation time on all three architectures. Certain optimized kernels are faster on KNL than on Broadwell. The KNL and Skylake processors have both 512 bit wide vector registers. Therefore, improving the vectorization for KNL immediately improved the performance on Skylake. Currently the Skylake processors provide the best performance for the Gysela code, several kernels are 20–50% faster on Skylake than on KNL.

2.10. ***Bibliography***

Bull, J M. "EPCC OpenMP Microbenchmark." 1999. http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_oldversion/ewomp.pdf.

Fog, Agner. "Instruction Tables." Technical University of Denmark, 2017.

Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. 2016.

McCalpin, John D. "Memory Bandwidth and Machine Balance in Current High Performance Computers." IEEE TCCA Newsletter, 1995: December.

McCalpin, John D. "Memory Latency on the Intel Xeon Phi x200 “Knights Landing” processor." 2016. https://sites.utexas.edu/jdm4372/2016/12/06/memory-latency-on-the-intel-xeon-phi-x200-knights-landing-processor/ (accessed March, 2017).

Preuss, R, R Fischer, and M Rampp. "Parallel equilibrium algorithm for real-time control." IPP Report, Max Planck Institute for Plasmaphysics, 2012.

Raman, Karthik. "Optimizing Memory Bandwidth in Knights Landing on Stream Triad." Intel. 2016. https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-in-knights-landing-on-stream-triad (accessed March 2017).

Williams, S, A Waterman, and D Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures." Communications of the ACM 52, no. 4 (2009): 65-72.

3. **Final report on HLST project SOLPSOPT**

3.1. *Introduction*

The Scrape-Off Layer (SOL) is directly related to the heat exhaust in tokamaks, therefore understanding the physics of this layer is crucial for improving the performance of present and future fusion devices. The SOLPS code package is widely used to simulate SOL plasmas. Numerical modelling plays an important role in understanding the basic physical concepts, interpreting results from experiments and making predictions for future experiments.

SOLPS is a collection of several codes; two main components are the Eirene and the B2 codes. The B2 code is a plasma fluid code to simulate edge plasmas and the Eirene code is a kinetic Monte-Carlo code for describing neutral particles.

The OpenMP parallelization of the SOLPS 5.0 version of B2 was done in two steps. F. Reid parallelized five time-consuming loops and many subroutines called from these loops. This resulted in almost a factor of two speedup for the ITER test case (Reid 2010). Afterwards, in the framework of the PARSOLPS and SOLPSOPT projects, T. Fehér and L. Hüdepohl parallelized several additional subroutines, and reached a factor of six speedup and above 90% parallelization of the B2 code.

Since then, the SOLPS-ITER version of the SOLPS package became available, and it has developed into the recommended version for new SOLPS simulations. Unfortunately, SOLPS-ITER did not include any of the parallelization mentioned in the previous paragraph. The aim of this extension of the SOLPSOPT project is to merge the OpenMP parallelization from SOLPS 5.0 to SOLPS-ITER.

In the next sections the parallelization of B2 is described, and the modified code is benchmarked both in standalone and in coupled mode.

3.2. *Test cases*

Let us start with an overview of the test cases used in this report. Table 5 gives a summary of the main parameters. The main target of the OpenMP optimization is the 98 species ITER test case. We have two versions of it: `ITER3` (Bonnin 2017), a standalone B2 only test case; and `ITER5` (SOLPS ITER 2018), which is a coupled B2-Eirene test case. The performance was tested for other test cases as well. The `AUG2` and `ITER4` test cases are downloaded from the SOLPS-ITER repository (SOLPS ITER Examples 2016). The source of the `ITER2` test case is the SVN repository at IPP (SOLPS ITER2 test case 2016).

		B2 parameters			Eirene parameters			
name	species	nx	ny	ns	particles	strata	background species	reactions
`AUG2`	D+C+He	98	36	12	67k	9	13	59
`ITER2`	D+Ar+He	92	36	24	87k	13	37	34
`ITER3`	D+T+He+Be+Ne+W	98	36	98	n.a.			
`ITER4`	D+He+N	92	36	13	96k	13	26	46
`ITER5`	D+T+He+Be+Ne+W	92	36	98	363k	39	92	35

Table 5 Main parameters of the test cases used in this report.

3.3. *OpenMP parallelization*

Around 400 lines of OpenMP directives were added into the SOLPS-ITER version of the B2 code. These are distributed over 50 source files, containing around 18k lines of code. There are around six thousand different lines between SOLPS 5.0 and SOLPS-ITER in these code regions, which made the inclusion of the OpenMP directives slightly more complicated than a straightforward merge.

To be able to emulate other SOLPS versions, SOLPS-ITER keeps multiple versions of certain subroutines (`b2news`, `b2news_`, `b2sifr`, `b2sifr_`, ...). The OpenMP directives were introduced in these duplicate subroutines as well. SOLPS-ITER introduced some new time consuming routines (e.g. `b2sifrtf`), these were also parallelized.

Some subroutines in SOLPS 5.0 were fine-tuned in order to reach a maximum performance determined by the memory bandwidth bottleneck. These tunings often required conceptually simple, but practically extensive changes to the source code. In the limited timeframe of this project, it was not possible to apply the extensive changesets to SOLPS-ITER, therefore we reverted to a simpler parallelization of these subroutines.

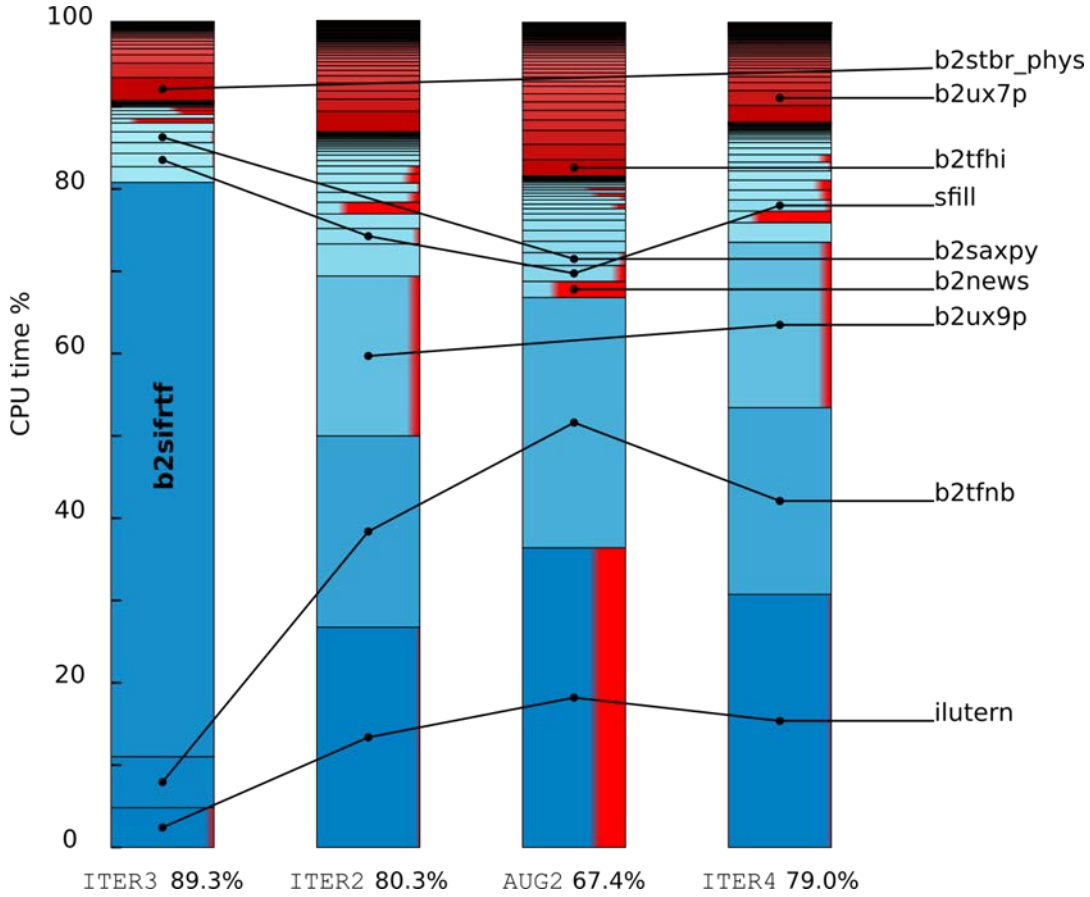


Fig. 30 Execution time of B2 using one thread. The four stacks of bars represent four test cases, the individual boxes denote different subroutines. Blue color: time spent in potentially parallel region, red color: time spent in sequential region. The number behind the test name (below the bars) signifies the time spent in parallel regions.

The FTimings performance library (Hüdepohl 2015) was used to profile the B2 code. The execution time of different subroutines was measured using four test cases. Fig. 30 shows how the execution time is divided between different subroutines[1]. Blue

[1] We only considered the execution time of the main driver subroutine of the calculation (b2mndr_1) in Fig. 30. We used only a few time steps in these test, therefore the initialization and finalization of

color denotes time spent in a parallel region, red denotes time spent in sequential regions. Some boxes have both red and blue colors; these denote either subroutines that have a significant sequential fraction, or subroutines that are called both from parallel and sequential parts of the code.

We have reached almost 90% parallelization for the 98 species ITER test case (first column), 80% for the other two ITER test cases, and close to 70% parallelization for the AUG2 test case. Due to differences in the input parameters, there are different subroutines used in the four test cases. The ITER3 test case uses a new subroutine to calculate friction force (b2sifrtf). The execution time of this subroutine depends on the square of the number of species, therefore it takes a large fraction of the execution time.

The speedup of the modified code is shown in Fig. 31. The red curve shows the measured speedup on the Skylake partition of Marconi. The maximal achieved speedup using 48 cores is 6.7x. The dotted blue line shows the theoretical upper estimate using Amdahl's law. By fine-tuning the parallelization of the existing OpenMP regions, we could push the red curve towards the blue curve. By increasing the parallel fraction of the code, we could move both the blue and red curves upwards. Unfortunately, the execution time is divided between many subroutines, therefore both strategies are time consuming. Before moving forward in further optimizations, we will test the parallel performance of the coupled B2-Eirene code.

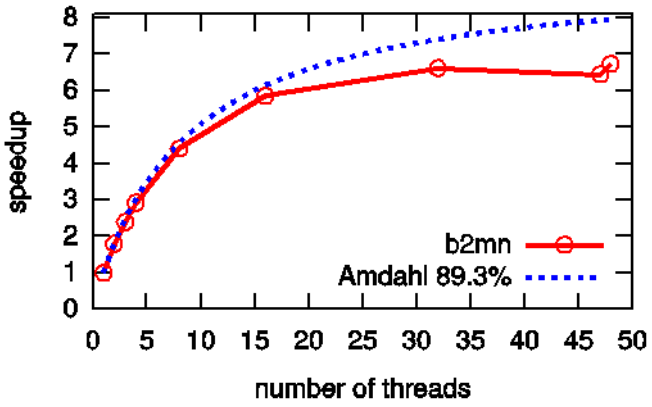


Fig. 31 Speedup of the standalone B2 code for the ITER3 test case on the Skylake partition of Marconi.

3.4. *B2-Eirene coupling*

Unlike the SOLPS 5.0 version, the SOLPS-ITER version of Eirene has a working MPI parallelization. The technical details of the coupling between the OpenMP B2 code and the MPI Eirene code were investigated in 2016. The synchronization between the MPI processes was replaced with the method proposed in (Feher 2016), to avoid wasting resources by spin-waiting.

The execution time of the coupled B2-Eirene code was measured using the newly available hybrid MPI-OpenMP parallelization. The results are shown in Fig. 32. The bars show the execution time of B2 and Eirene separately. Using one core, the Eirene code takes 86% of the runtime for the AUG2 test case, and 92% of the runtime for the ITER2 and ITER4 test cases. The B2 code takes a large share (75%) of the execution time for the ITER5 test case. The execution time of both B2 and Eirene decreases as we increase the number of cores. Even if Eirene has a large fraction of the runtime using one core, the runtime of B2 becomes significant when we use a larger number of cores. The execution time of Eirene is measured separately for the

---

the results (b2mndr_0 and b2mndr_2) take also a significant amount of the runtime. In the figure, we omit these subroutines, since their cost would be amortized during longer simulations.

initialization and coupling interface (red) and for the main calculation (green). While the calculation part is parallelized, the initialization and some of the coupling routines (red) are not. Further optimization efforts should also consider improving these sequential parts of Eirene.

The parallel efficiency of B2 is not as good as the one of Eirene, but the OpenMP parallelization still gives a contribution to the overall speedup. For the `AUG2` test case, a factor of 4.5 speedup could be reached, which would be only 3.4x without the OpenMP parallelization. Due to a problem with the parallelization of the `b2news` subroutine, the speedup for the B2 part of the `ITER5` test case is less than for the `ITER3` test case. The overall speedup for the `ITER5` test case is currently 3.5x, and it is expected to increase to 4.4x after this problem is fixed.

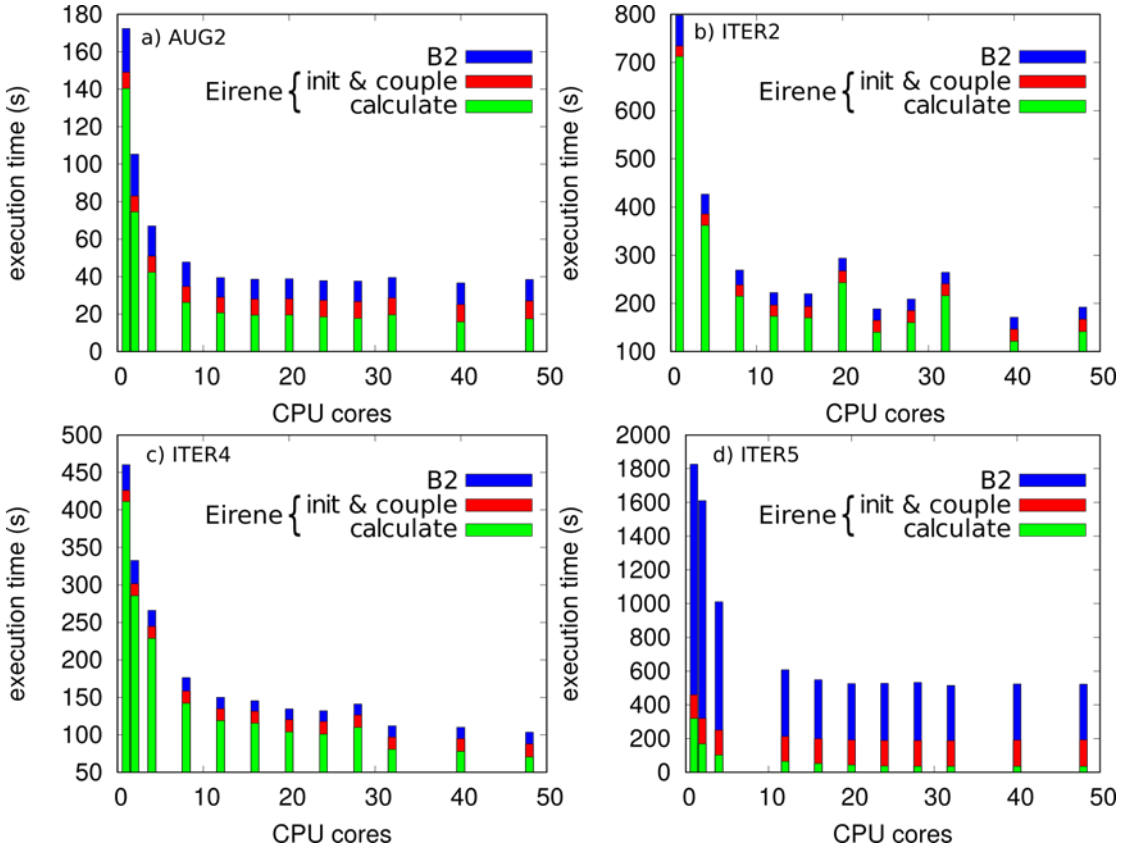


Fig. 32 Execution time of the coupled B2-Eirene code using the hybrid MPI-OpenMP parallelization, measured on the Skylake partition of Marconi.

3.5. *Summary*

The OpenMP parallelization of the SOLPS 5.0 version of the B2 code was merged into the SOLPS-ITER version of the B2 code. Up to 90% parallel fraction was reached for a standalone B2 simulation of the target test case, which is a 98 species ITER scenario. This results in a factor of 6.7 speedup for the standalone B2 code.

The Eirene code in SOLPS-ITER already had an MPI parallelization, and now it is possible to run the coupled B2-Eirene code with hybrid MPI-OpenMP parallelization. The coupled B2-Eirene simulation was tested with different test cases and it achieves a factor of 4-5x speedup.

3.6. *Bibliography*

Bonnin, Xavier (2017) "SOLPS-ITER 98 species test case." Private communication

Feher, Tamas (2016) "Final Report on HLST project SOLPSOPT."

Hüdepohl, Lorenz. (2015) FTimings
https://gitlab.mpcdf.mpg.de/loh/ftimings (accessed 2017).

Reid, J L Fiona (2010) Porting and parallelising SOLPS on HECTOR.
EUFORIA Report

SOLPS ITER Examples (2016)
https://portal.iter.org/departments/POP/CM/IMAS/SOLPS-ITER.

SOLPS ITER Examples (2018) ssh://git@git.iter.org/bnd/solps-iter.git
runs/examples/ITER_Be-W_D+T+He+Ne

SOLPS ITER2 test case (2016) https://solps-mdsplus.aug.ipp.mpg.de/
repos/SOLPS/trunk/solps_examples/Eirene_5.3/ITER_2054A_Eirene.

4. **Final report on HLST project GRIMG2**

4.1. ***Introduction***

Accurate modeling and a deeper understanding of the edge and scrape-off layer (SOL) in diverted magnetic fusion devices are important tasks in order to predict the performance of present and future fusion reactors like ITER or DEMO.

In many codes aimed at simulating the edge/SOL the turbulent dynamics are not treated by first principles but are just modeled as an effective diffusion, which may be an insufficient approximation in many cases. On the other hand the complex geometry of divertors in presence of a separatrix poses a special challenge for 3D codes: In order to exploit the characteristic flute mode property of structures, turbulence codes are usually based on field aligned coordinates, where the computational grid is sparsified along the resulting parallel direction. Unfortunately, field aligned coordinate systems become singular on the separatrix.

The recently proposed field line map approach (also called flux-coordinate independent approach) addresses the geometrical issues: A cylindrical grid (with coordinates *R,Z,phi*) is used, which is Cartesian within poloidal planes. The discretisation of perpendicular operators is straight-forward and simple as their stencils remain within (Cartesian) poloidal planes. Parallel operators are discretised using finite differences along magnetic field lines. Field line tracing from plane to plane is performed and the required values on a magnetic field line are thereby obtained by interpolation within poloidal planes. A grid sparsification in the toroidal direction is employed to exploit the flute mode character. Ultimately, simulations in tokamaks (and even stellarators) with an arbitrary poloidal cross section are possible. In particular, simulations using domains which span across the separatrix in diverted devices do not pose problems any more.

The recently developed GRILLIX code has proven the validity and feasibility of the field line map approach. Initially, the code was based on a simplified model (Hasegawa-Wakatani), which is currently being extended towards a realistic physical model: In order to cope with the strong fluctuations present in the SOL a full-*f* model has to be used, e.g. full-*f* drift reduced Braginskii equations or even a full-*f* gyro-fluid model. In any case, from a technical point of view an elliptic problem with varying coefficients (non-linear polarization) has to be solved for each time step in order to fulfill the quasi-neutrality condition.

For the simplified delta-*f* model a direct solver was still sufficient, but GRILLIX has very recently been equipped with a prototypical version of a geometric multigrid solver, which is aimed at solving the non-linear polarization equation efficiently. The multigrid solver works on a Cartesian non-rectangular grid with non-conforming boundaries and first results concerning accuracy and performance look promising. However, the solver is currently based on homogeneous Dirichlet boundary conditions in the radial direction which must be changed in order to treat the ultimately required more complex boundary conditions, e.g. of Neumann type.

Because the multigrid solver has been identified as a main bottleneck in GRILLIX, a performance tuning is also necessary in order to make simulations of larger tokamaks accessible. A Jacobi smoother is currently used in GRILLIX and more advanced smoothers, e.g. Gauss-Seidel, might lead to a significant performance improvement. Moreover, GRILLIX is MPI parallelised over the toroidal direction, but the solver works independently of the toroidal direction within poloidal planes, where an OpenMP parallelisation is employed. The HLST could help with increasing the parallel efficiency.

Finally, it shall be noted that the desired result of this project, i.e. an efficient and versatile multigrid solver which works on a Cartesian grid with non-conforming boundaries, could be widely reused, as the problem is very general and common to many other codes and even scientific fields.

4.2. *Discretization of the elliptic problem in GRILLIX*

The GRILLIX code solves the following elliptic partial differential equation (PDE) on poloidal planes as shown in Fig. 33:

$$-\nabla \cdot [c_0 \nabla_\perp u] = f. \tag{4.2.1}$$

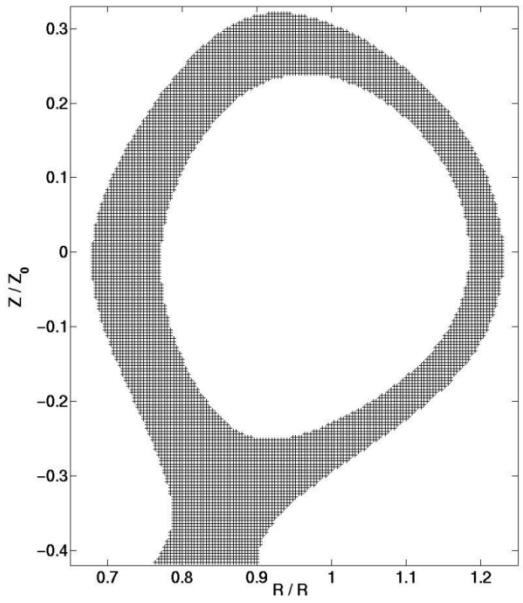


Fig. 33 The Cartesian coordinates of the GRILLIX code.

When Eq. (4.2.1) is discretised with finite differences on the uniform Cartesian grid along each direction (h_x and h_y), we get the following five-point stencil equation

$$-\frac{c_{0,i,j-\frac{1}{2}}}{h_y^2}u_{i,j-1} - \frac{c_{0,i-\frac{1}{2},j}}{h_x^2}u_{i-1,j} + \left(\frac{c_{0,i-\frac{1}{2},j}+c_{0,i+\frac{1}{2},j}}{h_x^2} + \frac{c_{0,i,j-\frac{1}{2}}+c_{0,i,j+\frac{1}{2}}}{h_y^2}\right)u_{i,j} - \frac{c_{0,i+\frac{1}{2},j}}{h_x^2}u_{i+1,j} - \frac{c_{0,i,j+\frac{1}{2}}}{h_y^2}u_{i,j+1} = f_{i,j}$$

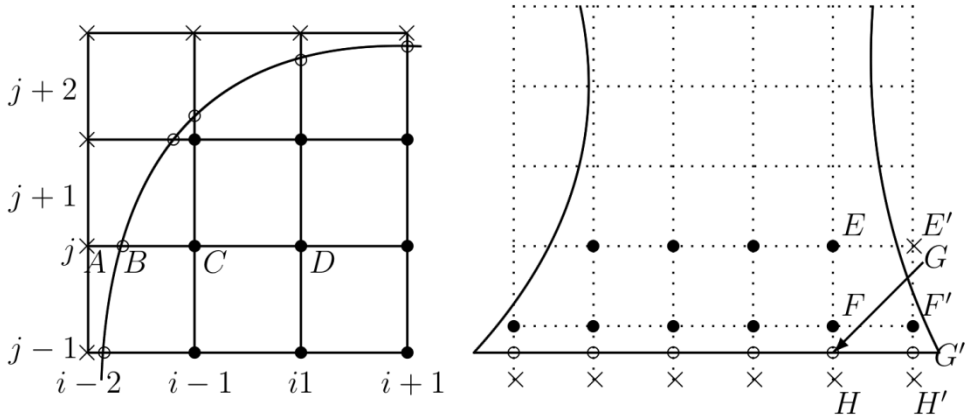
where $c_{0,i,j\pm\frac{1}{2}} = \dfrac{c_{0,i,j}+c_{0,i,j\pm1}}{2}$ and $c_{0,i\pm\frac{1}{2},j} = \dfrac{c_{0,i,j}+c_{0,i\pm1,j}}{2}$.

This discretisation is well developed and well analyzed. The derived linear system can be solved by the multigrid method with very good performance and scaling properties.

An example of how to handle a Dirichlet boundary condition is given in Fig**.** 34 a). We consider a point *C* where the five stencil notation has to use the boundary point *B* instead of point *A* which is outside the domain. There are two ways to generate the linear system: one is by directly using point *B*, the other one is to approximate the value at point *A* with the value of points *B* and *C*.

To do this, let us assume that the distance between *B* and *C* is *h*. Then we have, at point *C*:

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u(C) \approx -\frac{c_{0,i-\frac{1}{2},j}}{h_x^2}u(A) + \frac{c_{0,i-\frac{1}{2},j}+c_{0,i+\frac{1}{2},j}}{h_x^2}u(C) - \frac{c_{0,i+\frac{1}{2},j}}{h_x^2}u(D).$$



a) b)

Fig. 34 The Cartesian coordinate system of the GRILLIX code near a boundary perpendicular to the *y*-axis with a) a Dirichlet boundary condition and b) a Neumann boundary condition. Points inside the domain are denoted by •, points on the boundary are denoted by ○ and ghost points are denoted by ×.

If we use the first order approximation for *B* making use of the values at the points *A* and *C*

$$u(B) = \frac{h}{h_x}u(A) + \left(1 - \frac{h}{h_x}\right)u(C), \text{ i.e., } u(A) = \left(1 - \frac{h_x}{h}\right)u(C) + \frac{h_x}{h}u(B),$$

we have

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u \approx -\frac{c_{0,i-\frac{1}{2},j}}{h_x h}u(B) + \frac{c_{0,i-\frac{1}{2},j}h_x + c_{0,i+\frac{1}{2},j}h}{h_x^2 h}u(C) - \frac{c_{0,i+\frac{1}{2},j}}{h_x^2}u(D).$$

If we use the second order approximation for *B* making use of the values at points *A*, *C*, and *D*,

$$u(B) = \frac{h(h+h_x)}{2h_x^2}u(A) - \frac{(h_x^2 - h^2)}{h_x^2}u(C) + \frac{h(h-h_x)}{2h_x^2}u(D), \text{ i.e.,}$$
$$u(A) = \frac{2h_x^2}{h(h+h_x)}u(B) - \frac{2(h_x - h)}{h}u(C) + \frac{h_x - h}{h_x + h}u(D),$$

we get

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u \approx -\frac{2c_{0,i-\frac{1}{2},j}}{h(h+h_x)}u(B) + \frac{c_{0,i-\frac{1}{2},j}(2h_x - h) + c_{0,i+\frac{1}{2},j}h}{h_x^2 h}u(C)$$
$$- \frac{c_{0,i-\frac{1}{2},j}(h_x - h) + c_{0,i+\frac{1}{2},j}(h_x + h)}{h_x^2(h_x + h)}u(D)$$

.

We get equivalent results for the other direction:

$$-\frac{\partial}{\partial y}c_0\frac{\partial}{\partial y}u.$$

For the Neumann boundary condition on a boundary which is perpendicular to the *y*-axis as shown in Fig. 34 b), we calculate the approximating value at point *H* with the normal derivative at point *G* using the values at points *E* and *F*. With a second order approximation and $h = |FG|$, we have

$$g_n(G) = -\frac{\partial}{\partial y}u(G) = \frac{2h - h_y}{2h_y^2}u(H) + \frac{2h - h_y}{2h_y^2}u(E) - \frac{2h}{h_y^2}u(F), \text{ i.e.,}$$
$$u(H) = \frac{4h}{2h + h_y}u(F) - \frac{2h - h_y}{2h + h_y}u(E) + \frac{2h_y^2}{2h + h_y}g_n(G).$$

When the corresponding point *E'* is not located inside the domain, which is the case for *F'*, we have to use the first order approximation, i.e.,

$$g_n(G) = \frac{\partial}{\partial n}u(G) = -\frac{\partial}{\partial y}u(G) = \frac{u(F) - u(H)}{h_y}, \text{ i.e., } u(H) = u(F) - h_y g_n(G).$$

We use these ghost values to complete the finite difference approximation of the five-point stencil.

We now consider the Neumann boundary conditions on generically shaped boundaries. A typical boundary shape near the diverter is shown in Fig. 35. A typical boundary condition is a Dirichlet boundary condition on the outer boundary except at the diverter, and a Neumann boundary condition on the inner boundary and the diverter.

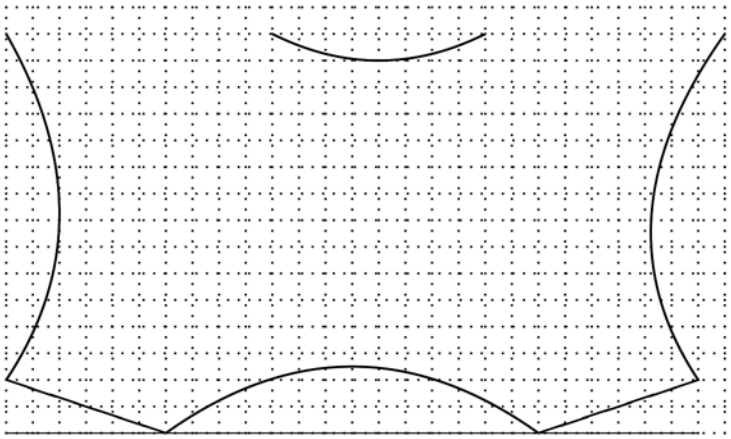


Fig. 35 The typical boundary of the GRILLIX code near the diverter.

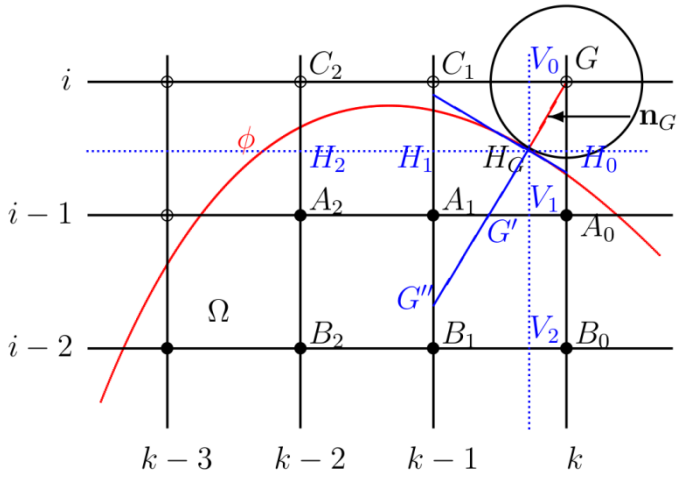


Fig. 36 The stencil is in the upwind direction with respect to the normal vector $\boldsymbol{n}_G$ computed at H_G, while the discretization is performed along the co-normal direction. Points inside the domain are denoted by •, points and ghost points are denoted by ○.

To handle the Neumann boundary condition on a generically shaped boundary including the region near the diverter, we may choose zero-, first-, or second-order approximation schemes. To get the value at a ghost point $G(x,y)$, we find the closest boundary point $H_G(x_h,y_h)$ by drawing a circle with its center at G. Then we compute the outward unit normal at H to G,

$$\mathbf{n}_G = \frac{(x-x_h, y-y_h)}{\sqrt{(x-x_h)^2+(y-y_h)^2}} = \frac{(x-x_h, y-y_h)}{|\mathbf{n}_G|} = (n_x, n_y).$$

We extend GH_G in order to get points G' and G'' which are the crossing points with the horizontal and vertical lines of the grid (see Fig**.** 36). We then discretize the Neumann boundary condition

$$g_n(H_G) = \frac{\partial u_h}{\partial n}(H_G) = (\nabla u \cdot \mathbf{n})|_{H_G},$$

where u is the linear interpolation along GH_G, or a bilinear or a biquadratic interpolation on the following upwind stencil St_m:

$$St_m = \left\{ G + (s_x k_x h_x, s_y k_y h_y) : (k_x, k_y) \in \{0, 1, m\}^2 \right\},$$

with $s_x = \mathrm{sgn}(x-x_h)$ and $s_y = \mathrm{sgn}(y-y_h)$.

For the zero-order approximation (m=0) which is used in the current version of GRILLIX, we choose a point closer to A_1 among G' and G'' and compute $G_1(x_p,y_p)$. We compute the normal derivative along GG_1, i.e.,

$$g_n(H_G) = \frac{\partial u}{\partial n}(H_G) = \frac{(u(G) - u(G_1))}{|GG_1|}, \text{ i.e., } u_G = |GG_1| g_n(H_G) + u(G_1).$$

We can also get $u(G_1)$ by using linear interpolation with the two nearest points A_0 and A_1 when $G_1 = G'$ in Fig**.** 36**,** i.e.,

$$u(G_1) = \frac{h}{h_x} u(A_0) + \left(1 - \frac{h}{h_x}\right) u(A_1)$$

with $h = |A_1G_1|$.

The first-order approximation (m=1) of $g_n(H_G)$ along $\boldsymbol{n}_G$ can be calculated as

$$g_n(H_G) \approx \frac{\partial u}{\partial x}(H_G) n_x + \frac{\partial u}{\partial y}(H_G) n_y = \frac{u_{H_0} - u_{H_1}}{h_x} n_x + \frac{u_{V_0} - u_{V_1}}{h_y} n_y$$

where the values $u(H_1)$, $u(H_0)$, $u(V_0)$, and $u(V_1)$ can obtained using the values $u(G)$, $u(A_1)$, $u(C_1)$ (or $u(B_1)$), and $u(A_0)$ (or $u(A_2)$) according to the properties of the points. If three points C_1, A_0 and A_1 are real points, then we have

$$\begin{aligned} g_n(H_G) = {} & \frac{h_y n_y + h_x n_y - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(G) - \frac{h_y n_x - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(C_1) \\ & - \frac{h_x n_y - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(A_0) - \frac{n_x n_y |\mathbf{n}_G|}{h_x h_y} u(A_1). \end{aligned}$$

If the point C_1 is a ghost point, then we can use A_0, A_1, and B_1 (instead of C_1) and get

$$\begin{aligned} g_n(H_G) = {} & \frac{h_y n_x + h_x n_y - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(G) + \frac{h_y n_x - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(B_1) \\ & - \frac{h_x n_y - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(A_0) - \frac{2h_y n_x - n_x n_y |\mathbf{n}_G|}{h_x h_y} u(A_1). \end{aligned}$$

For the second-order approximation (m=2) of $g_n(H_G)$ along $\boldsymbol{n}_G$, we use the nine-point stencil St_2, i.e., for the x-directional difference with three points we use H_0, H_1, and H_2 and for the y-directional difference with three points we use V_0, V_1, and V_2:

$$\frac{\partial u}{\partial x}(H_G) = \frac{3h_x - n_x|\mathbf{n}_G|}{2h_x^2}u(H_0) - \frac{4h_x - 2n_x|\mathbf{n}_G|}{2h_x^2}u(H_1) + \frac{h_x - n_x|\mathbf{n}_G|}{2h_x^2}u(H_2)$$
$$\frac{\partial u}{\partial y}(H_G) = \frac{3h_y - n_y|\mathbf{n}_G|}{2h_y^2}u(V_0) - \frac{4h_y - 2n_y|\mathbf{n}_G|}{2h_y^2}u(V_1) + \frac{h_y - n_y|\mathbf{n}_G|}{2h_y^2}u(V_2).$$

For the typical case of a nine-point stencil as shown in Fig. 36 and with eight real points except G, we get

$$\begin{aligned}
u(H_0) &= M_{0,y}u(G) + M_{1,y}u(A_0) + M_{2,y}u(B_0)\\
u(H_1) &= M_{0,y}u(C_1) + M_{1,y}u(A_1) + M_{2,y}u(B_1)\\
u(H_2) &= M_{0,y}u(C_2) + M_{1,y}u(A_2) + M_{2,y}u(B_2)\\
u(V_0) &= M_{0,x}u(G) + M_{1,x}u(C_1) + M_{2,x}u(C_2)\\
u(V_1) &= M_{0,x}u(A_0) + M_{1,x}u(A_1) + M_{2,x}u(A_2)\\
u(V_2) &= M_{0,x}u(B_0) + M_{1,x}u(B_1) + M_{2,x}u(B_2)
\end{aligned}$$

where

$$M_{0,x} = \frac{n_x^2|\mathbf{n}_G|^2 - 6n_xh_x|\mathbf{n}_G| + 4h_x^2}{4h_x^2}, \quad M_{0,y} = \frac{n_y^2|\mathbf{n}_G|^2 - 6n_yh_y|\mathbf{n}_G| + 4h_y^2}{4h_y^2},$$
$$M_{1,x} = -\frac{n_x^2|\mathbf{n}_G|^2 - 4n_xh_x|\mathbf{n}_G|}{2h_x^2}, \qquad M_{1,y} = -\frac{n_y^2|\mathbf{n}_G|^2 - 4n_yh_y|\mathbf{n}_G|}{2h_y^2},$$
$$M_{2,x} = \frac{n_x^2|\mathbf{n}_G|^2 - 2n_xh_x|\mathbf{n}_G|}{4h_x^2}, \qquad M_{2,y} = \frac{n_y^2|\mathbf{n}_G|^2 - 2n_yh_x|\mathbf{n}_G|}{4h_y^2}.$$

We need to use different types of stencils according to the position of the boundary and the ghost points. Both schemes may have more nonzero elements on each row when generating the linear system than a five-point stencil.

There are two ways to generate the linear system, namely, using the value of the ghost points explicitly or implicitly. In order to implicitly take the value of the ghost points into account, we have to solve additional equations for the ghost points which are connected with interior points. This will lead to a linear system which has additional nonzero elements. For an explicit implementation we need to write a routine to compute the value of the ghost points from the value of the interior points. Then we can just use the five-point stencil connecting the interior points with the ghost points. In the end we decided to use the value of the ghost points explicitly in the matrix-vector multiplication and smoothing routines.

We now consider how to calculate the Laplace equation on each point. We start with the classification of a ghost point near a boundary with a Dirichlet boundary condition (ϕ_d), of a ghost point near a boundary with a Neumann boundary condition (ϕ_n), and the remaining real (ϕ_r) points. In the current GRILLIX version, we use the following numbering: a negative integer for the ghost points and a positive integer for the real points with dummy value zero, i.e., from **ghost%lpnti** (< 0, the absolute value equals the number of ghost points on the grid) to **ghost%lpntf** (= -1) and from **grid%lpnti** (= 1) to **grid%lpntf** (equals the number of real points on the grid). After rearranging the ghost points according to the boundary condition near the ghost points (first ghost points for the Dirichlet boundary condition then the ones for the Neumann boundary condition) Eq. (4.2.1) becomes the following system,

$$\begin{pmatrix} B_d \\ B_n \\ 1 \\ A_r \end{pmatrix} \begin{pmatrix} \phi_d \\ \phi_n \\ 0 \\ \phi_r \end{pmatrix} = \begin{pmatrix} B_{dd} & \mathbf{0} & \mathbf{0} & B_{dr} \\ \mathbf{0} & B_{nn} & \mathbf{0} & B_{nr} \\ \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & A_{rn} & \mathbf{0} & A_{rr} \end{pmatrix} \begin{pmatrix} \phi_d \\ \phi_n \\ 0 \\ \phi_r \end{pmatrix} = \begin{pmatrix} f_d \\ f_n \\ 0 \\ f_r \end{pmatrix}$$

where B_d, B_n, and A_r are rectangular matrices and B_{dd} and B_{nn} are diagonal matrices. For the homogeneous boundary condition (zero valued Dirichlet or Neumann boundary conditions), f_d and f_n will be zero. From the above linear system, we see that ϕ_n and ϕ_r do not depend on ϕ_d and ϕ_d can be calculated after solving for ϕ_n and ϕ_r .We can get a solution (ϕ_n, ϕ_r) from the initial approximation (ϕ_n^0, ϕ_r^0) by performing the following iterations

$$\begin{aligned} \phi_n^{m+\frac{1}{2}} &= B_{nn}^{-1}(f_n - B_{nr}\phi_r^m), \\ \phi_r^m &= A_{rr}^{-1}(f_r - A_{rn}\phi_n^{m+\frac{1}{2}}), \\ \phi_n^{m+1} &= B_{nn}^{-1}(f_n - B_{nr}\phi_r^m) \end{aligned}$$

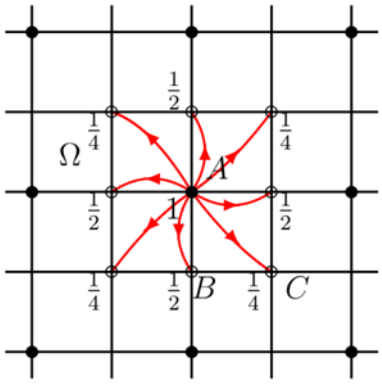
and get

$$\phi_d = B_{dd}^{-1}(f_d - B_{dr}\phi_r).$$

4.3. *Multigrid algorithm*

For the multigrid algorithm, we need to define coarser grid level operators A_{2h}, intergrid transfer operators, and smoothing operators on each level.

To generate coarser grid level operators, we may use the same scheme to generate the linear operator on the finest level which was explained in the previous section (which are used currently in GRILLIX) or use the algebraic relation $A_{2h} = c\, I_h^{2h} A_h I_{2h}^h$ where I_h^{2h} and I_{2h}^h are the intergrid transfer operators. In general, the generated linear system with the algebraic relation has more nonzero elements in each row than the system of the finest level.



$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Fig. 37 The stencil notation of the prolongation transfer operator.

The intergrid transfer operators on the Cartesian grid are well known, but we need a modification near the boundary. For the prolongation operators, we can use a typical linear interpolation, as shown in Fig. 37, after all values of the ghost points of the coarser grid are computed.

For the projection operators, we use a typical linear projection operator which is well defined and tested. This operator is shown on the left of Fig. 38. We need some modifications near the boundary when some of the neighbouring points are ghost points. Usually, we restrict the residual on the finer grid and cannot impose the boundary condition. However, in this case, we need contributions from ghost points as well. One way is to set the value of the ghost point using the value of the reflected

point in respect to the point on the coarse grid. Then we get the stencil notation as shown in the second and third graph of Fig. 38.

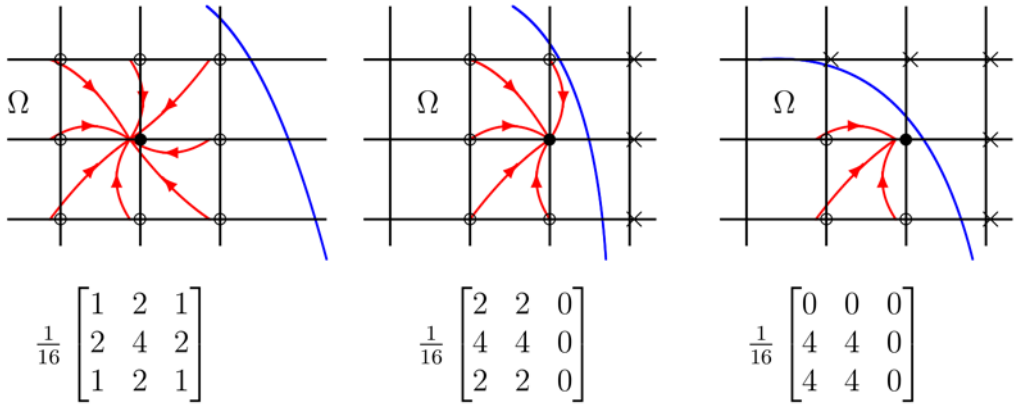


Fig. 38 The stencil notation of the projection transfer operator near the boundary (blue line). Points on the coarse grid are denoted by •, points on the fine grid are denoted by ○ (real points) and ghost points are denoted by ×.

4.4. *Numerical studies*

To verify the implementation of the multigrid algorithm, we consider a domain with a diverter which is parallel to the *x*-axis and which is discretized with a uniform grid, i.e., $h_x=h_y=h$. We assume that $c_0 = 1$ on the domain and that the problem has a Dirichlet boundary condition except for the diverter area. We label the grids from the coarsest level ($L=1$) with a mesh size of $h(1)=0.032$ and $h(K+1) = h(K)/2$. The number of real points and ghost points is set according to Table 6.

L	h	# of real points	# of ghost points
1	0.032	99	140
2	0.016	383	286
3	0.008	1 573	575
4	0.004	6 293	1 152
5	0.002	25 115	2 298
6	0.001	100 392	4 592
7	0.0005	401 427	9 180
8	0.00025	1 605 884	18 364

Table 6 The number of real and ghost points with respect to level (L) and mesh size (h).

level	# of real points	# of iterations	time [s]
4	6 293	321	0.128
5	25 115	1 121	1.831
6	100 392	4 039	26.321
7	401 427	12 573	350.437
8	1 605 884	60 301	7147.670

Table 7 The required number of iterations for convergence and the solution time in seconds of GMRES with respect to the level (L).

Consecutive Levels	Ratio		
	# of real points	# of iterations	time
5/4	4.025	3.492	14.30
6/5	3.997	3.600	14.38
7/6	3.999	3.113	13.31
8/7	4.000	4.996	20.40
Theoretical	4.000	4.000	16.00

Table 8 The ratio of the required number of real points, the required number of iterations for convergence and the solution time for two consecutive levels. The theoretical value, in the last row, is the theoretical estimation for any two consecutive levels.

The current implementation of the program is not parallelized, so we can only use one core. We set a residual of 10^{-10} as the stop criterion for the iterative solver. We report on required number of iterations and the respective solution times of GMRES in Table 7. In addition, we show the ratio of the number of points and the number of iterations for two consecutive levels in Table 8. The numerical results in Table 7 and Table 8 confirm our expectations about the required number of iterations and execution time. The typical required number of iterations of GMRES is C_1×(number of nodes) and the typical number of operations per iteration is also C_2×(number of nodes), i.e., the number of operations (which is proportional to the execution time on a single core) of GMRES is C_3×(number of nodes)^2 where C_1, C_2 *and* C_3 are constants.

To verify the implementation of the multigrid algorithm, we test it using several finest levels with different grid resolutions, different numbers of levels and different numbers of smoothing steps with the Jacobi and the Gauss-Seidel smoothers. We use GMRES as a coarsest level solver. We report on required number of iterations for convergence for each case in Table 9. The numerical results in Table 9 show that the required number of smoothing steps for convergence is at least two for both kinds of smoothers. The required number of smoothing steps for convergence is more than three with the Jacobi smoother and two with the Gauss-Seidel smoother. They are slightly increasing with the number of levels. This is a typical property of the multigrid method.

L_f	levels	Jacobi					Gauss-Seidel				
	(L_c)	1	2	3	4	5	1	2	3	4	5
4	3 (2)	*	15	11	10	9	*	10	8	7	7
	4 (1)	*	16	12	11	10	41	10	9	8	8
5	3 (3)	*	16	12	10	9	18	10	8	7	7
	4 (2)	*	16	12	10	10	17	10	9	8	8
	5 (1)	*	16	13	11	10	17	11	10	8	8
6	4 (3)	*	16	12	11	10	15	10	9	8	8
	5 (2)	*	16	13	12	11	17	11	11	9	9
	6 (1)	*	17	14	12	11	19	12	12	9	9
7	5 (3)	*	17	13	12	11	24	11	12	13	9
	6 (2)	*	19	15	13	12	26	12	13	10	11
	7 (1)	*	18	14	12	11	24	13	13	10	11
8	5 (4)	*	48	16	14	12	*	14	15	11	11
	6 (3)	*	*	19	16	14	*	24	17	12	14
	7 (2)	*	28	20	17	15	*	17	18	13	14
	8 (1)	*	26	19	16	14	*	16	16	12	13

Table 9 The required number of iterations of the multigrid algorithm with respect to the finest level, the number of levels, and the number of smoothing steps with the Jacobi and Gauss-Seidel smoothers. * stands for not converged results. L_c denotes the coarsest level.

Next, we report on solution time in seconds for each case in Table 10. We only report on cases with two, three, and four smoothing steps and emphasize the best cases for each level and smoothing step in boldface. The numerical results show that the optimal number of smoothing steps is three for the Jacobi smoother and two for the Gauss-Seidel smoother. We are also investigating the scaling properties as a function of the number of degrees of freedom (DoF) and report on best cases with the ratio of the solution times of consecutive levels in Table 11. The ratio of the solution times is slightly greater than four which is equal to the ratio of the numbers of DoF. This numerical result is relevant to the typical multigrid algorithm which is *N*log*N*, where *N* is the number of DoF. Also, we plot the solution time of the multigrid algorithm in comparison with the iterative solver GMRES in Fig**.** 39.

L_f	levels	Jacobi			Gauss-Seidel		
	(L_c)	2	3	4	2	3	4
4	3 (2)	0.019	0.015	0.015	0.013	0.012	0.012
	4 (1)	0.012	**0.011**	0.012	**0.008**	0.010	0.010
5	3 (3)	0.124	0.099	0.087	0.084	0.076	0.068
	4 (2)	0.053	0.047	**0.045**	0.037	0.040	0.042
	5 (1)	**0.045**	**0.045**	**0.045**	**0.035**	0.040	0.038
6	4 (3)	0.269	0.224	0.229	0.179	0.193	0.190
	5 (2)	0.181	**0.180**	0.197	**0.141**	0.179	0.175
	6 (1)	0.185	0.188	0.193	0.150	0.190	0.172
7	5 (3)	0.880	0.809	0.869	0.623	0.844	0.745
	6 (2)	0.849	0.829	0.856	**0.606**	0.839	0.774
	7 (1)	0.797	**0.769**	0.788	0.650	0.834	0.771
8	5 (4)	11.090	4.838	4.881	3.627	4.779	4.047
	6 (3)	*	**4.741**	4.789	4.990	4.515	3.806
	7 (2)	5.407	4.872	4.997	3.436	4.670	4.050
	8 (1)	5.181	4.784	4.867	**3.298**	4.229	3.808

Table 10 The solution time in seconds of the multigrid algorithm with respect to the finest level, the number of levels, and the number of smoothing steps with the Jacobi and the Gauss-Seidel smoothers. The fastest cases for each level and smoothing step are in boldface. * stands for not converged results. L_c denotes the coarsest level.

level	# of real points	Jacobi [s]	Ratio	G-S [s]	Ratio
4	6 293	0.011		0.008	
5	25 115	0.045	4.09	0.035	4.38
6	100 392	0.180	4.00	0.141	4.03
7	401 427	0.769	4.27	0.606	4.30
8	1 605 884	4.741	6.17	3.298	5.44

Table 11 The best solution times of the multigrid algorithm for each level and the ratio of the solution times of consecutive levels.

A typical property of a multigrid solver is that the number of operations is *CN*log*N,* i.e., the ratio of the numbers of operations is 4*N*log(4*N*)/(*N*log*N*) = 4.4 when *N* = 1,000,000. The numerical results in Fig. 39 show the typical behavior of a multigrid solver and are a good indication that the implementation of the multigrid algorithm is correct.

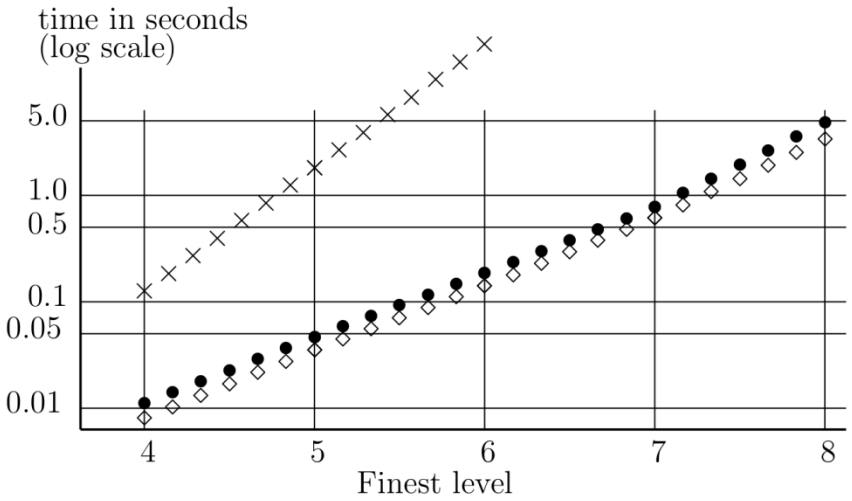


Fig. 39 The best solution times of the multigrid algorithm with respect to the level (××× for GMRES, ••• for a multigrid solver with a Jacobi smoother, ◊◊◊ for a multigrid solver with a Gauss-Seidel smoother).

4.5. *Current status and future work*

The original implementation by the project coordinator consisted of the multigrid algorithm and the Laplace operator. Both resulted in a linear system in GRILLIX which was implemented on a matrix-free basis. We added a matrix generation function which provides several different orders of approximation for the boundary conditions. The ELLPACK format is used for storing and handling the generated matrix. This format allows an easy implementation of the Gauss-Seidel smoother. As main task we have also implemented the multigrid algorithm with GMRES as a solver for the coarsest level. We showed the validity of the code with numerical tests on a single core.

We found that the required number of smoothing steps for convergence is at least two for both smoothers. The required number of smoothing steps for convergence is more than three for the Jacobi smoother and two for the Gauss-Seidel smoother. Numerical results show that previously used five smoothing steps are too costly in terms of performance. We also report on scaling properties of the solver.

We have sent the code to the project coordinator in order for it to be merged with the GRILLIX code. The project coordinator plans to modify the numbering system of the ghost nodes which would result in the sole use of positive numbers, removing negative numbers from the numbering. He asked the HLST for help with this modification. We support to help the project coordinator with the implementation of this modification as a follow-up project.

5. **Report on the BIT2-3 project**

5.1. ***Introduction***

Three dimensional kinetic modelling of the plasma edge is one of the most ambitious topics in numerical plasma study. It is required for performing ab initio modelling of linear plasma devices and divertor plasmas, for answering a number of long-standing questions on the role of drifts in the plasma wall transition layer (PWT), for formulating reliable multi-dimensional boundary conditions in the PWT and for qualitative estimation of the impurity sputtering and the plasma facing components (PFC) erosion rates. Therefore development of the corresponding 3D kinetic codes has a high priority.

The 3D massively parallel particle in cell and Monte Carlo code BIT3, which represents a generalization of the BIT1 and BIT2 codes to three dimensions, is being developed. The simulation geometry of this code corresponds to an elongated rectangular parallelepiped. Typical numbers for the size of the 3D Cartesian uniform grid, to be used in BIT3, range from $50 \times 50 \times 10^3$ to $10^3 \times 10^3 \times 10^4$. All the components of the BIT3 code are ready with the exception of the massively parallel 3D Poisson solver.

The aim of the project is to develop this 3D Poisson solver based on the experience in developing the corresponding 2D solver in BIT2 and to incorporate it into BIT3. The solver will be based on the multigrid method using a mixed domain decomposition/finite element discretization. The optimized 2D solver has good scaling properties (up to 1000 cores) for the complex scrape-off-layer (SOL) geometry (overall BIT2 scales up to 16 000 cores). The 3D solver, contrary to the 2D one, has to be applied to a more simplified geometry not containing any holes. Therefore, the scaling of the 3D solver should exceed the scaling of the 2D solver. This allows the overall scalability of the BIT3 code to be well above 16 000 cores.

Another part of the project is represented by the implementation of the biased PFC structures into the BIT2 code. Recently the 2D Poisson solver developed by the HLST has been successfully implemented into the BIT2 code. The BIT2 code is very flexible and allows the simulation of different SOL geometries and different atomic/molecular and PSI processes, but it cannot simulate the biasing of the PFC structures. On the other hand, recent experiments indicate that PFC biasing strongly affects net erosion rates of high-*Z* materials, which in turn might influence the overall content of high-*Z* impurities during the discharge in present day and future tokamaks. The behavior of the erosion during the PFC biasing is unexplained and requires corresponding kinetic modelling. Another application of PFC biasing is the interpretation of Langmuir probe measurements at JET and AUG. There is still not a good understanding of these measurements during the transient events like ELMs, which require corresponding modelling.

The implementation of the biased boundaries into the BIT2 Poisson solver will significantly increase the predictive capabilities of the BIT2 code, enabling it to tackle the problems discussed above.

We will first implement and then test the parallel multigrid method for the BIT3 code and then modify it for the BIT2 code.

5.2. ***Parallel implementation of the multigrid method***

In this section, we present an implementation of the parallel multigrid algorithm with a merging step which solves discretized system on massively parallel computers. This implementation can be used for 2D and 3D problems.

Parallel Multigrid Algorithm

Recursive multigrid: $u_h^{(k+1)} = V_h\left(u_h^{(k)}, A_h, f_h, \nu_h^1, \nu_h^2, \mu\right)$

1: **if** coarsest level **then**

2: solve $A_h u_h^{(k+1)} = f_h$ by a parallel direct solver or Krylov iteration solver

3: **else if** merging level **then**

4: Merge vector f_h to serial f_h^s

5: Perform serial multigrid algorithm $u_h^s = V_h\left(0, A_h^s, f_h^s, \nu_h^1, \nu_h^2, \mu\right)$

6: Distribute serial u_h^s to vector $u_h^{(k+1)}$

7: **else**

8: $\bar{u}_h^{(k)} = \mathcal{S}^{\nu_h^1}\left(u_h^{(k)}, A_h, f_h\right)$ {presmoothing}

9: $r_H = Rr_h = R(f_h - A_h\bar{u}_h^{(k)})$ {restrict computed residual}

10: $e_H^i = e_H^{i-1} + V_H\left(0, A_H, r_H - A_H e_H^{i-1}, \nu_H^1, \nu_H^2, \mu\right)$ for $i = 1, \ldots, \mu$ {recursion}

11: $\tilde{u}_h^{(k+1)} = \bar{u}_h^{(k)} + Pe_H^\mu$ {prolongate coarse grid error correction}

12: $u_h^{(k+1)} = \mathcal{S}^{\nu_h^2}\left(\tilde{u}_h^{(k+1)}, A_h, f_h\right)$ {postsmoothing}

13: **end if**

As shown in the **Parallel Multigrid Algorithm**, the algorithm needs the coarse grid operators A_H on each grid level. A_H can be obtained on each grid by direct computations, but as a matter of fact we would then need the value of the coefficient $\varepsilon(x)$ on each grid as well. To avoid computing the coefficient on each coarser grid level, we may use a numerically efficient algebraic relation of the coarse and fine grid to calculate A_H in an approximate way

$$A_H = cI_h^H A_h I_H^h$$

where c can be used to improve performance. Only for the simplest case of $\varepsilon(x)=const$ one can chose the value of c in such a way that A_H is identical with the results from its conventional computation.

Usually Krylov subspace methods (CGM or GMRES) as well as direct methods can be used as coarsest level solvers. These solvers are well known and fast for small problems but are not scalable with respect to the number of degrees of freedom (DoF).

The smoothing method has to be simple and should reduce the high frequency error component. The damped Jacobi and Gauss-Seidel methods are well-known smoothers. The damped Jacobi method is less effective than the Gauss-Seidel method, but its performance doesn't depend on the number of MPI tasks. Therefore, we can use the damped Jacobi method for debugging parallel codes. However in real computing, we use the Gauss-Seidel method as a smoother. The multigrid method can be used as a solver and as a preconditioner for the Krylov subspace method. Using the multigrid method as a solver doesn't guarantee convergence in some cases, so we also consider a Krylov subspace method.

In the **Parallel Multigrid Algorithm**, each core has to handle more than one DoF. From this follows that the number of levels has to be limited. This limitation causes a severe problem with the coarsest level as it needs a lot of time to be solved. To overcome this limitation, we use a merging level algorithm which uses a serial multigrid algorithm starting from a certain level.

Modern computer architectures have highly hierarchical system designs, i.e. multi-socket multi-core shared-memory nodes which are connected via high-speed interconnects. This trend will continue into the foreseeable future, broadening the available range of hardware designs for high-end systems. Consequently, it seems natural to employ a hybrid programming model which uses OpenMP for parallelization inside the node and MPI for message passing between nodes. OpenMP is an API specification for multi-threading, a method of parallelization whereby a master thread forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the run time environment allocating threads to different processors. This approach can be used to reduce the number of MPI tasks and therefore increase the flexibility of the program.

5.3. *Discretization in 3D*

To develop a 3D multigrid solver for second order partial differential equations (PDEs), we consider the domain Ω of a parallelepiped given by L_x, L_y, and L_z shown in Fig. 40. In particular, we will focus on the second order PDE

$$-\left[\frac{\partial}{\partial x}\epsilon(\mathbf{x})\frac{\partial}{\partial x}+\frac{\partial}{\partial y}\epsilon(\mathbf{x})\frac{\partial}{\partial y}+\frac{\partial}{\partial z}\epsilon(\mathbf{x})\frac{\partial}{\partial z}\right]\phi(\mathbf{x})=\rho(\mathbf{x})$$

where **x** = *(x, y, z)* with a large aspect ratio L_z/L_x and $L_x = L_y$. ϕ is the electrostatic potential, ρ is the charge density and ε is the dielectric constant. As boundary conditions we choose a Dirichlet boundary condition in the *x*-direction and Dirichlet or Neumann boundary conditions in the *y*- and *z*-directions.

The parallelepiped domain is discretized with a mesh size of $h_x = L_x/n_x$, $h_y = L_y/n_y$, and $h_z = L_z/n_z$ as shown in Fig. 41. The discretized function $\phi_{i,j,k}$ is defined on all points $P(x_{i,j,k}) = P(x_i, y_j, z_k)$ for $x_i = ih_x$, $y_j = jh_y$, $z_k = kh_z$, $0 \le i \le n_x$, $0 \le j \le n_y$, and $0 \le k \le n_z$ as shown in Fig. 41.

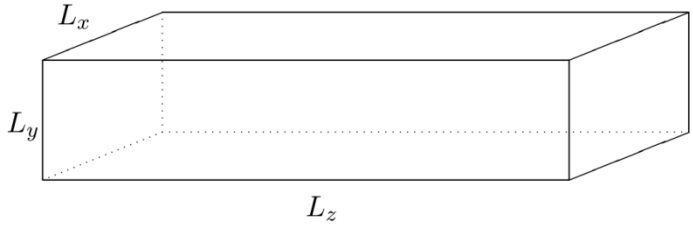


Fig. 40 Simulation domain of the BIT3 code.

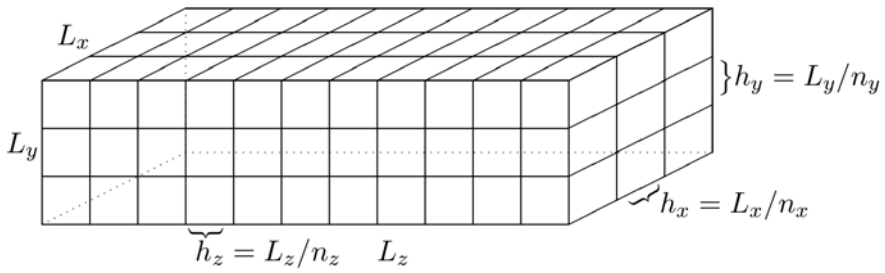


Fig. 41 Discretized domain of the BIT3 code.

We assume that $\varepsilon(x, y, z)$ is defined on each cell (x_i, y_j, z_k) which has its center point at $((i+\frac{1}{2})h_x, (j+\frac{1}{2})h_y, (k+\frac{1}{2})h_z)$ for $0 \le i \le n_x-1$, $0 \le j \le n_y-1$, and $0 \le k \le n_z-1$, in a similar

way to the previous 2D case. This shall be denoted by $\epsilon_{i+\frac{1}{2},j+\frac{1}{2},k+\frac{1}{2}} = \epsilon((i+\frac{1}{2})h_x, (j+\frac{1}{2})h_y, (k+\frac{1}{2})h_z)$.

For the 3D multigrid solver we consider two discretization methods. One is a finite difference method (FDM) and the other is a finite element method (FEM). The respective intergrid transfer operators have to be shaped differently for each discretization method.

5.3.1. **Finite difference discretization**

We consider a typical second order finite difference scheme for a 3D domain. We get the following finite difference formulations of the first derivatives of $\phi_{i,j,k} = \phi(\mathbf{x}_{i,j,k}) = \phi(ih_x, jh_y, kh_z)$:

$$\begin{aligned}
\frac{\partial}{\partial x}\phi_{i-\frac{1}{2},j,k} &= \frac{\phi_{i,j,k}-\phi_{i-1,j,k}}{h_x}, \quad \frac{\partial}{\partial x}\phi_{i+\frac{1}{2},j,k} = \frac{\phi_{i+1,j,k}-\phi_{i,j,k}}{h_x} \\
\frac{\partial}{\partial y}\phi_{i,j-\frac{1}{2},k} &= \frac{\phi_{i,j,k}-\phi_{i,j-1,k}}{h_y}, \quad \frac{\partial}{\partial y}\phi_{i,j+\frac{1}{2},k} = \frac{\phi_{i,j+1,k}-\phi_{i,j,k}}{h_y} \\
\frac{\partial}{\partial z}\phi_{i,j,k-\frac{1}{2}} &= \frac{\phi_{i,j,k}-\phi_{i,j,k-1}}{h_z}, \quad \frac{\partial}{\partial z}\phi_{i,j,k+\frac{1}{2}} = \frac{\phi_{i,j,k+1}-\phi_{i,j,k}}{h_z}
\end{aligned}$$

Accordingly the finite difference formulations of the second order derivatives are given by

$$\begin{aligned}
\frac{\partial}{\partial x}\epsilon(\mathbf{x})\frac{\partial}{\partial x}\phi_{i,j,k} &= \left\{\epsilon_{i+\frac{1}{2},j,k}\frac{\partial}{\partial x}\phi_{i+\frac{1}{2},j,k} - \epsilon_{i-\frac{1}{2},j,k}\frac{\partial}{\partial x}\phi_{i-\frac{1}{2},j,k}\right\}/h_x \\
&= \frac{(\phi_{i+1,j,k}-\phi_{i,j,k})\epsilon_{i+\frac{1}{2},j,k} - (\phi_{i,j,k}-\phi_{i-1,j,k})\epsilon_{i-\frac{1}{2},j,k}}{h_x^2} \\
&= \frac{\phi_{i+1,j,k}\epsilon_{i+\frac{1}{2},j,k} - \phi_{i,j,k}(\epsilon_{i+\frac{1}{2},j,k}+\epsilon_{i-\frac{1}{2},j,k}) + \phi_{i-1,j,k}\epsilon_{i-\frac{1}{2},j,k}}{h_x^2},
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial y}\epsilon(\mathbf{x})\frac{\partial}{\partial y}\phi_{i,j,k} &= \left\{\epsilon_{i,j+\frac{1}{2},k}\frac{\partial}{\partial x}\phi_{i,j+\frac{1}{2},k} - \epsilon_{i,j-\frac{1}{2},k}\frac{\partial}{\partial x}\phi_{i,j-\frac{1}{2},k}\right\}/h_y \\
&= \frac{(\phi_{i,j+1,k}-\phi_{i,j,k})\epsilon_{i,j+\frac{1}{2},k} - (\phi_{i,j,k}-\phi_{i,j-1,k})\epsilon_{i,j-\frac{1}{2},k}}{h_y^2} \\
&= \frac{\phi_{i,j+1,k}\epsilon_{i,j+\frac{1}{2},k} - \phi_{i,j,k}(\epsilon_{i,j+\frac{1}{2},k}+\epsilon_{i,j-\frac{1}{2},k}) + \phi_{i,j-1,k}\epsilon_{i,j-\frac{1}{2},k}}{h_y^2},
\end{aligned}$$

$$\begin{aligned}
\frac{\partial}{\partial z}\epsilon(\mathbf{x})\frac{\partial}{\partial z}\phi_{i,j,k} &= \left\{\epsilon_{i,j,k+\frac{1}{2}}\frac{\partial}{\partial z}\phi_{i,j,k+\frac{1}{2}} - \epsilon_{i,j,k-\frac{1}{2}}\frac{\partial}{\partial z}\phi_{i,j,k-\frac{1}{2}}\right\}/h_z \\
&= \frac{(\phi_{i,j,k+1}-\phi_{i,j,k})\epsilon_{i,j,k+\frac{1}{2}} - (\phi_{i,j,k}-\phi_{i,j,k-1})\epsilon_{i,j,k-\frac{1}{2}}}{h_z^2} \\
&= \frac{\phi_{i,j,k+1}\epsilon_{i,j,k+\frac{1}{2}} - \phi_{i,j,k}(\epsilon_{i,j,k+\frac{1}{2}}+\epsilon_{i,j,k-\frac{1}{2}}) + \phi_{i,j,k-1}\epsilon_{i,j,k-\frac{1}{2}}}{h_z^2},
\end{aligned}$$

where

$$\begin{aligned}
\epsilon_{i,j,k\pm\frac{1}{2}} &= \frac{1}{4}(\epsilon_{i+\frac{1}{2},j+\frac{1}{2},k\pm\frac{1}{2}} + \epsilon_{i-\frac{1}{2},j+\frac{1}{2},k\pm\frac{1}{2}} + \epsilon_{i+\frac{1}{2},j-\frac{1}{2},k\pm\frac{1}{2}} + \epsilon_{i-\frac{1}{2},j-\frac{1}{2},k\pm\frac{1}{2}}), \\
\epsilon_{i,j\pm\frac{1}{2},k} &= \frac{1}{4}(\epsilon_{i+\frac{1}{2},j\pm\frac{1}{2},k+\frac{1}{2}} + \epsilon_{i-\frac{1}{2},j\pm\frac{1}{2},k+\frac{1}{2}} + \epsilon_{i+\frac{1}{2},j\pm\frac{1}{2},k-\frac{1}{2}} + \epsilon_{i-\frac{1}{2},j\pm\frac{1}{2},k-\frac{1}{2}}), \\
\epsilon_{i\pm\frac{1}{2},j,k} &= \frac{1}{4}(\epsilon_{i\pm\frac{1}{2},j+\frac{1}{2},k+\frac{1}{2}} + \epsilon_{i\pm\frac{1}{2},j-\frac{1}{2},k+\frac{1}{2}} + \epsilon_{i\pm\frac{1}{2},j+\frac{1}{2},k-\frac{1}{2}} + \epsilon_{i\pm\frac{1}{2},j-\frac{1}{2},k-\frac{1}{2}})
\end{aligned}$$

and $\varepsilon_{i\pm\frac{1}{2},\, j\pm\frac{1}{2},\, k\pm\frac{1}{2}}$ are the dielectric constants defined at the cell center as shown in Fig. 42.

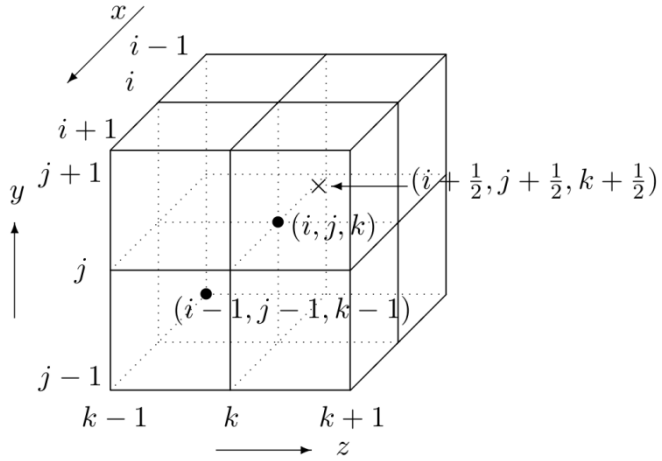


Fig. 42 Notation for the discretization of the 3D domain.

By summing these equations, we get

$$
\begin{aligned}
&-\frac{\epsilon_{i+\frac{1}{2},j,k}}{h_x^2}\phi_{i+1,j,k} - \frac{\epsilon_{i-\frac{1}{2},j,k}}{h_x^2}\phi_{i-1,j,k} - \frac{\epsilon_{i,j+\frac{1}{2},k}}{h_y^2}\phi_{i,j+1,k} \\
&-\frac{\epsilon_{i,j-\frac{1}{2},k}}{h_y^2}\phi_{i,j-1,k} - \frac{\epsilon_{i,j,k+\frac{1}{2}}}{h_z^2}\phi_{i,j,k+1} - \frac{\epsilon_{i,j,k-\frac{1}{2}}}{h_z^2}\phi_{i,j,k-1} \\
&+\left\{\frac{\epsilon_{i+\frac{1}{2},j,k}+\epsilon_{i-\frac{1}{2},j,k}}{h_x^2} + \frac{\epsilon_{i,j+\frac{1}{2},k}+\epsilon_{i,j-\frac{1}{2},k}}{h_y^2} + \frac{\epsilon_{i,j,k+\frac{1}{2}}+\epsilon_{i,j,k-\frac{1}{2}}}{h_z^2}\right\}\phi_{i,j,k} = \rho_{i,j,k}.
\end{aligned}
$$

Concerning the Dirichlet boundary condition, on the boundaries it holds that

$$\phi(\mathbf{x}) = \phi_c(\mathbf{x}) \quad .$$

The internal conductor area is included in the boundaries. The values of the function $\phi_c(\mathbf{x})$ are constants, i.e.,

$$\phi_{i,j,k} = \phi_c.$$

For the Neumann boundary conditions, we use the central difference scheme (see Fig. 43) with ghost points outside the domain.

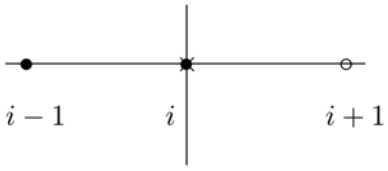


Fig. 43 Notation for the finite difference scheme for the Neumann boundary condition at the surface with $x = L_x$ (• : real nodal points, × : boundary point, ○ : ghost point).

For example, on the boundary with the Neumann boundary condition, along normal direction **n** in holds that

$$\frac{\partial\phi}{\partial\mathbf{n}} = \frac{\partial\phi}{\partial x} = g_x$$

And we get

$$\frac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{2\Delta x} = g_{i,j,k},$$

i.e.,

$$\phi_{i+1,j,k} = \phi_{i-1,j,k} + 2\Delta x g_{i,j,k}.$$

Finally, we get the discretized system

$$Ax = b,$$

where *A* has the following form

$$A = \begin{bmatrix} C & D & 0 & 0 & \cdots & \cdots & 0 \\ D & C & D & 0 & \ddots & \cdots & 0 \\ 0 & D & C & D & 0 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & 0 & D & C & D & 0 \\ 0 & \ddots & \ddots & 0 & D & C & D \\ 0 & \cdots & \cdots & \cdots & 0 & D & C \end{bmatrix} \text{ with } C = \begin{bmatrix} T & D & 0 & 0 & \cdots & 0 \\ D & T & D & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & D & T & D & 0 \\ 0 & \ddots & 0 & D & T & D \\ 0 & \cdots & 0 & 0 & D & T \end{bmatrix}$$

and *D* is a diagonal matrix and *T* is a tridiagonal matrix.

With *2i = I, 2j = J, 2k = K*, the linear restriction operator $P_{l\text{-}1}$ is defined by

$$\begin{aligned}(P_{l-1}\phi)_{I,J,K} = {} & \frac{1}{8}\phi_{2i,2j,2k} + \frac{1}{16}\Big\{\phi_{2i-1,2j,2k} + \phi_{2i+1,2j,2k} + \phi_{2i,2j-1,2k} + \phi_{2i,2j+1,2k} \\ & + \phi_{2i,2j,2k-1} + \phi_{2i,2j,2k+1}\Big\} + \frac{1}{32}\Big\{\phi_{2i-1,2j-1,2k} + \phi_{2i+1,2j+1,2k} + \phi_{2i+1,2j-1,2k} \\ & + \phi_{2i-1,2j+1,2k} + \phi_{2i-1,2j,2k-1} + \phi_{2i+1,2j,2k+1} + \phi_{2i+1,2j,2k-1} + \phi_{2i-1,2j,2k+1} \\ & + \phi_{2i,2j-1,2k-1} + \phi_{2i,2j+1,2k+1} + \phi_{2i,2j-1,2k+1} + \phi_{2i,2j+1,2k-1}\Big\} \\ & + \frac{1}{64}\Big\{\phi_{2i-1,2j-1,2k-1} + \phi_{2i+1,2j+1,2k+1} + \phi_{2i+1,2j-1,2k-1} + \phi_{2i-1,2j+1,2k-1} \\ & + \phi_{2i-1,2j-1,2k+1} + \phi_{2i+1,2j+1,2k-1} + \phi_{2i+1,2j-1,2k+1} + \phi_{2i-1,2j+1,2k+1}\Big\}\end{aligned}$$

and the linear prolongation operator is defined by

$$\begin{aligned}(I_l\phi)_{2i,2j,2k} &= \phi_{I,J,K} \\ (I_l\phi)_{2i\pm1,2j,2k} &= \frac{1}{2}\{\phi_{I\pm1,J,K} + \phi_{I,J,K}\} \\ (I_l\phi)_{2i,2j\pm1,2k} &= \frac{1}{2}\{\phi_{I,J\pm1,K} + \phi_{I,J,K}\} \\ (I_l\phi)_{2i,2j,2k\pm1} &= \frac{1}{2}\{\phi_{I,J,K\pm1}) + \phi_{I,J,K}\} \\ (I_l\phi)_{2i\pm1,2j\pm1,2k} &= \frac{1}{4}\{\phi_{I\pm1,J\pm1,K} + \phi_{I\pm1,J,K} + \phi_{I,J\pm1,K}) + \phi_{I,J,K}\} \\ (I_l\phi)_{2i\pm1,2j,2k\pm1} &= \frac{1}{4}\{\phi_{I\pm1,J,K\pm1} + \phi_{I\pm1,J,K} + \phi_{I,J,K\pm1} + \phi_{I,J,K}\} \\ (I_l\phi)_{2i,2j\pm1,2k\pm1} &= \frac{1}{4}\{\phi_{I,J\pm1,K\pm1} + \phi_{I,J,K\pm1}) + \phi_{I,J\pm1,K} + \phi_{I,J,K}\} \\ (I_l\phi)_{2i\pm1,2j\pm1,2k\pm1} &= \frac{1}{8}\{\phi_{I\pm1,J\pm1,K\pm1} + \phi_{I\pm1,J,K\pm1} + \phi_{I,J\pm1,K\pm1} + \phi_{I,J,K\pm1} \\ &\quad + \phi_{I\pm1,J\pm1,K} + \phi_{I\pm1,J,K} + \phi_{I,J\pm1,K} + \phi_{I,J,K}\}\end{aligned}$$

using the notation shown in Fig. 44.

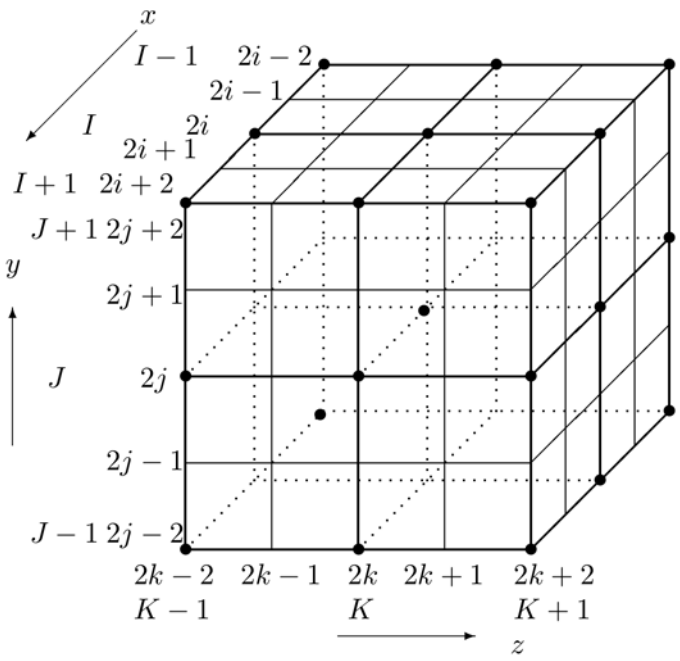


Fig. 44 Notation for the intergrid transfer operators of the 3D problem (• : coarser node points).

The intergrid transfer operators are defined on the boundary in accordance with the boundary condition. The values of the functions $P_{l-1}\phi$ and $I_l\phi$ on the Dirichlet boundaries are all zero because we solve a problem with a zero-Dirichlet boundary condition, i.e., a zero value on all Dirichlet boundary points, except on the finest level. The values of the function on the Neumann boundary are defined in the same way as in the case of the finite difference schemes with a zero-Neumann boundary condition, i.e., the values at the ghost points are the same as the reflective real points,

$$\phi_{i+1,j,k} = \phi_{i-1,j,k}$$

where $P_{i,j,k}$ is a boundary point with the Neumann boundary condition.

5.3.2. **Finite element discretization**

In order to define a finite element scheme, we consider the weak formulation of the problem described in Sec. 5.3. By multiplying with the test function $\psi(\mathbf{x})$ and integrating on both sides, we get

$$-\int_\Omega \left[\frac{\partial}{\partial x}\epsilon(\mathbf{x})\frac{\partial}{\partial x} + \frac{\partial}{\partial y}\epsilon(\mathbf{x})\frac{\partial}{\partial y} + \frac{\partial}{\partial z}\epsilon(\mathbf{x})\frac{\partial}{\partial z}\right]\phi(\mathbf{x})\psi(\mathbf{x})\,\mathrm{d}\mathbf{x} = \int_\Omega \rho(\mathbf{x})\psi(\mathbf{x})\,\mathrm{d}\mathbf{x}.$$

By using Green's theorem, we get

$$\int_\Omega \epsilon(\mathbf{x})\left[\frac{\partial\phi(\mathbf{x})}{\partial x}\frac{\partial\psi(\mathbf{x})}{\partial x} + \frac{\partial\phi(\mathbf{x})}{\partial y}\frac{\partial\psi(\mathbf{x})}{\partial y} + \frac{\partial\phi(\mathbf{x})}{\partial z}\frac{\partial\psi(\mathbf{x})}{\partial z}\right]\mathrm{d}\mathbf{x} - \int_{\partial\Omega}\epsilon(\mathbf{x})\psi(\mathbf{x})\frac{\partial\phi(\mathbf{x})}{\partial\mathbf{n}}\,\mathrm{d}s = \int_\Omega \rho(\mathbf{x})\psi(\mathbf{x})\,\mathrm{d}\mathbf{x},$$

i.e.,

$$\int_\Omega \epsilon(\mathbf{x}) \left[\frac{\partial \phi(\mathbf{x})}{\partial x} \frac{\partial \psi(\mathbf{x})}{\partial x} + \frac{\partial \phi(\mathbf{x})}{\partial y} \frac{\partial \psi(\mathbf{x})}{\partial y} + \frac{\partial \phi(\mathbf{x})}{\partial z} \frac{\partial \psi(\mathbf{x})}{\partial z} \right] \mathrm{d}\mathbf{x} = \int_\Omega \rho(\mathbf{x}) \psi(\mathbf{x}) \, \mathrm{d}\mathbf{x} + \int_{\partial\Omega} \epsilon(\mathbf{x}) \psi(\mathbf{x}) g_{\mathbf{n}}(\mathbf{x}) \, \mathrm{d}s.$$

We consider the piecewise trilinear element space which is a continuous and trilinear polynomial space on each rectangular parallelepiped (Fig. 45) with a finite element basis $\phi_{p,q} = \delta_{p,q}$.

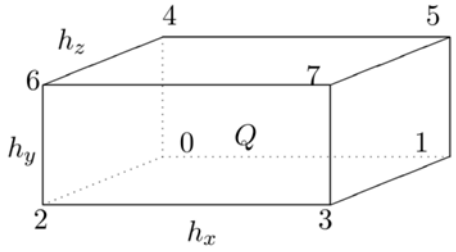


Fig. 45 A rectangular parallelepiped element Q for the FEM of a 3D problem.

First of all, we compute the integration of the rectangular parallelepiped, as shown in Fig. 45, with respect to each basis function:

$$a_{p,p} = \frac{\epsilon_Q}{9} \left[\frac{h_x h_z}{h_y} + \frac{h_x h_y}{h_z} + \frac{h_y h_z}{h_x} \right], \quad p = 0, \ldots, 7$$

$$a_{p,q} = \frac{\epsilon_Q}{18} \left[\frac{h_x h_z}{h_y} + \frac{h_x h_y}{h_z} - \frac{2 h_y h_z}{h_x} \right], \quad (p,q) = (0,1), (2,3), (4,5), (6,7)$$

$$a_{p,q} = \frac{\epsilon_Q}{18} \left[\frac{h_y h_z}{h_x} + \frac{h_x h_y}{h_z} - \frac{2 h_x h_z}{h_z} \right], \quad (p,q) = (0,2), (1,3), (4,6), (5,7)$$

$$a_{p,q} = \frac{\epsilon_Q}{18} \left[\frac{h_x h_z}{h_y} + \frac{h_y h_z}{h_x} - \frac{2 h_x h_y}{h_z} \right], \quad (p,q) = (0,4), (1,5), (2,6), (3,7)$$

$$a_{p,q} = \frac{\epsilon_Q}{36} \left[\frac{h_x h_y}{h_z} - \frac{2 h_x h_z}{h_y} - \frac{2 h_y h_z}{h_x} \right], \quad (p,q) = (0,3), (1,2), (4,7), (5,6)$$

$$a_{p,q} = \frac{\epsilon_Q}{36} \left[\frac{h_x h_z}{h_y} - \frac{2 h_x h_y}{h_z} - \frac{2 h_y h_z}{h_x} \right], \quad (p,q) = (0,5), (1,4), (2,7), (3,6)$$

$$a_{p,q} = \frac{\epsilon_Q}{36} \left[\frac{h_y h_z}{h_x} - \frac{2 h_x h_y}{h_z} - \frac{2 h_x h_z}{h_y} \right], \quad (p,q) = (0,6), (1,7), (2,4), (3,7)$$

$$a_{p,q} = \frac{\epsilon_Q}{36} \left[-\frac{h_x h_y}{h_z} - \frac{h_x h_z}{h_y} - \frac{h_y h_z}{h_x} \right], \quad (p,q) = (0,7), (1,6), (2,5), (3,4)$$

To generate a matrix equation, we assemble all computations on the rectangular elements of the parallelepiped and get a discretized linear system

$$Ax = b.$$

In comparison with the FDM, which only has seven nonzero elements, the generated matrix of the FEM has 27 nonzero elements.

The restriction operators for the FEM are the same as for the FDM. But the prolongation operator, which is the transpose of the restriction operator, is different on the boundaries for a Neumann boundary condition.

5.4. *Numerical studies*

5.4.1. **Without merging step**

For performance reasons a merging step becomes necessary in a parallel multigrid algorithm when the number of DoF on one core becomes too small at a certain level.

In such a case all data is merged on a particular core to solve the remaining levels serially with the multigrid algorithm until the coarsest level is reached. At the coarsest level the remaining number of DoF is already quite small and it becomes feasible to solve the problem by an iterative or direct solver. However, without the merging the coarsest level still has quite a large number of DoF which makes solving it by other methods than multigrid quite inefficient. As a result the performance of the full multigrid algorithm is relatively poor on a massively parallel computer.

		All Dirichlet BD				Neumann BD 1-dir			
		FDM		FEM		FDM		FEM	
Problem size	levels	Ja	GS	Ja	GS	Ja	GS	Ja	GS
$2^5 \times 2^5 \times 2^5$	4	17	9	8	6	17	9	8	5
$2^6 \times 2^6 \times 2^6$	4	18	9	8	7	17	9	8	6
	5	18	9	8	6	17	9	8	6
$2^7 \times 2^7 \times 2^7$	4	18	9	8	7	17	9	8	6
	5	18	9	8	7	17	9	8	6
	6	18	9	8	6	17	9	8	6
$2^8 \times 2^8 \times 2^8$	4	18	9	8	7	17	9	8	6
	5	18	9	8	7	17	9	8	6
	6	18	9	8	6	17	9	8	6
$2^9 \times 2^9 \times 2^9$	4	18	9	8	7	17	9	8	6
	5	18	9	8	7	17	9	8	6
	6	18	9	8	7	17	9	8	6
$2^{10} \times 2^{10} \times 2^{10}$	5	18	9	8	7	17	9	8	6
	6	18	9	8	7	17	9	8	6
$2^{11} \times 2^{11} \times 2^{11}$	6	18	9	8	7	17	9	8	6

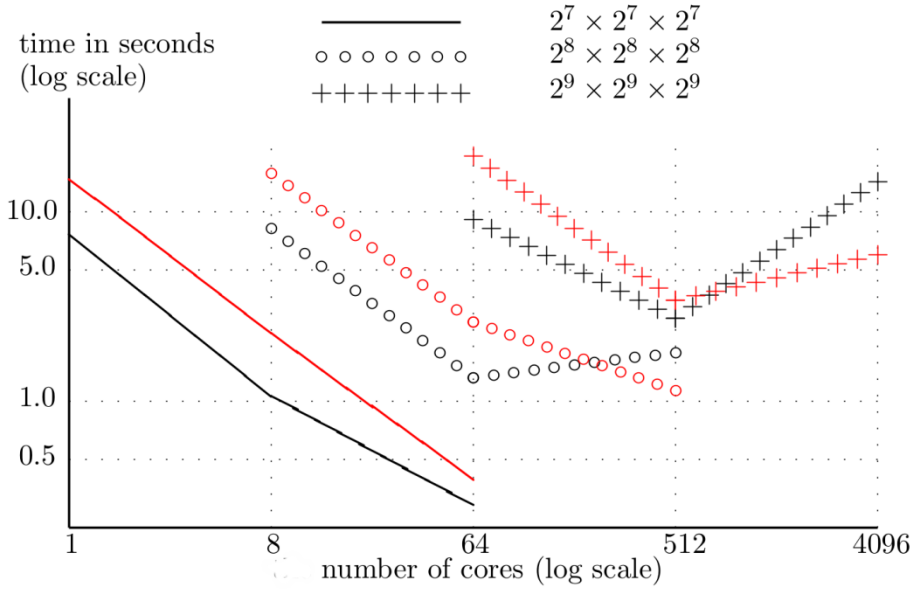
Table 12 The required number of iterations of the 3D parallel multigrid algorithm for a cube domain.

		All Dirichlet BD				Neumann BD 1-dir			
		FDM		FEM		FDM		FEM	
Problem size	levels	Ja	GS	Ja	GS	Ja	GS	Ja	GS
$2^4 \times 2^4 \times 2^7$	3	17	9	8	5	17	9	8	5
$2^5 \times 2^5 \times 2^8$	4	18	9	8	7	17	9	8	7
$2^6 \times 2^6 \times 2^9$	4	18	9	8	7	17	9	8	7
	5	18	9	8	7	18	9	8	7
$2^7 \times 2^7 \times 2^{10}$	4	18	9	8	7	17	9	8	7
	5	18	9	8	7	17	9	8	7
	6	18	9	8	7	17	9	8	7
$2^8 \times 2^8 \times 2^{11}$	4	18	9	8	7	18	9	8	7
	5	18	9	8	7	18	9	8	7
	6	18	9	8	7	18	9	8	7
$2^9 \times 2^9 \times 2^{12}$	5	18	9	8	7	18	9	8	7
	6	18	9	8	7	18	9	8	7
$2^{10} \times 2^{10} \times 2^{13}$	6	18	9	8	7	18	9	8	7

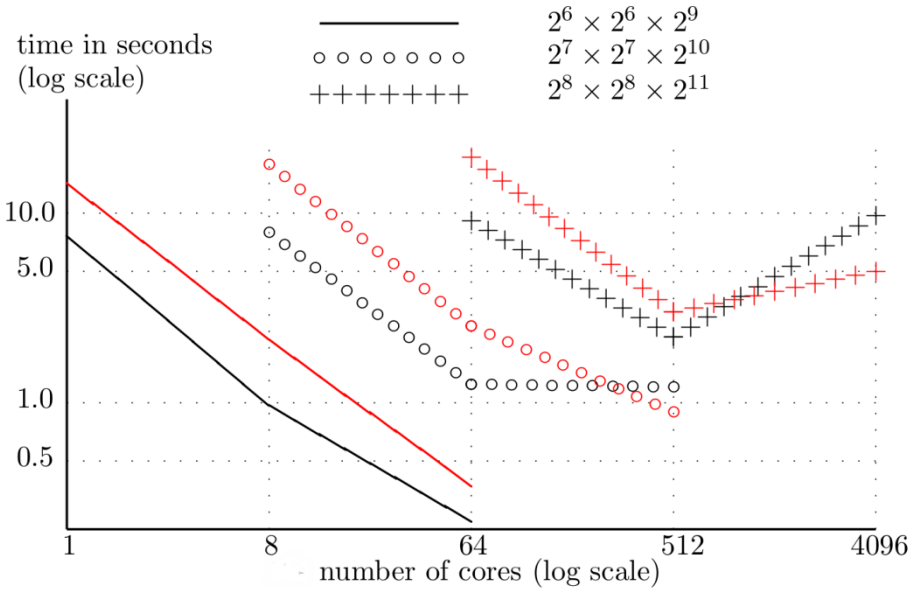
Table 13 The required number of iterations of the 3D parallel multigrid algorithm for the elongated rectangular parallelepiped domain with an axis ratio of (1 × 1 × 8).

First, we check the number of iterations of the multigrid cycle, which should be independent of the number of grid levels if the implementation is correct. As test problems we consider two domains, one is a cube domain and the other is an elongated rectangular parallelepiped domain as explained in the introduction. The maximum number of DoF per core is 2^{21}, i.e. $2^7 \times 2^7 \times 2^7$ or $2^6 \times 2^6 \times 2^9$. We tested both

the FDM and FEM with Jacobi (Ja) and Gauss-Seidel (GS) smoothing. We observed that the number of iterations is fixed, as shown in Table 12 and Table 13, for the multigrid algorithm with Jacobi and Gauss-Seidel smoothers. The number of iterations of the multigrid method for the FEM is less than for the FDM.

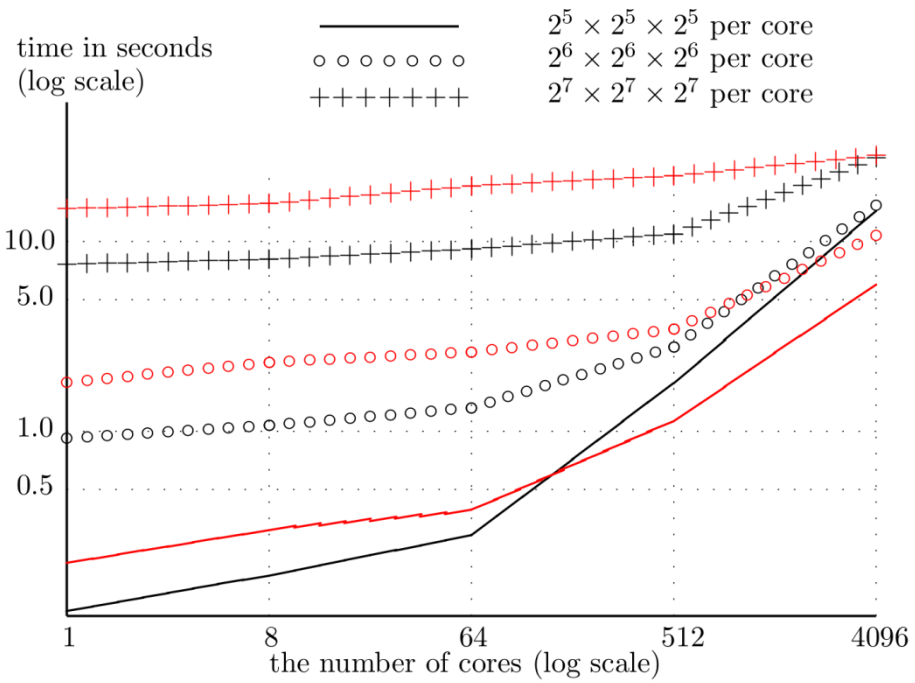


(a) On the unit cube domain (1×1×1)

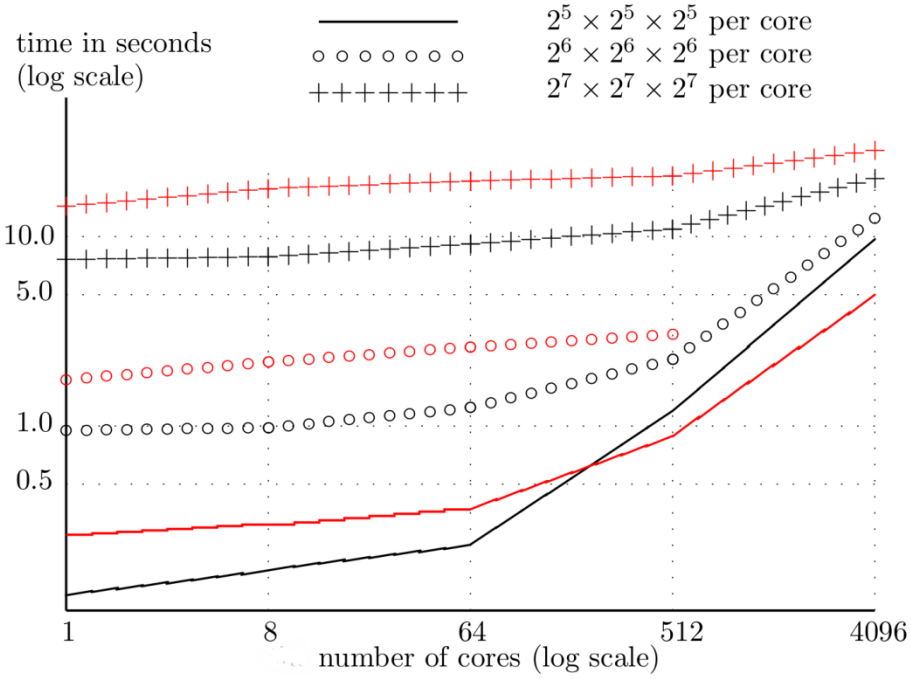


(b) On the elongated rectangular parallelepiped domain (1×1×8)

Fig. 46 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores. Three different mesh sizes have been selected in order to test strong scaling. The results of the finite difference method (FDM) are shown in black and the results of the finite element method (FEM) are shown in red.



(a) On the unit cube domain (1×1×1)



(b) On the elongated rectangular parallelepiped domain (1×1×8)

Fig. 47 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores for the case with a fixed number of DoF per core. The results of the finite difference method (FDM) are shown in black and the results of the finite element method (FEM) are shown in red.

Next we consider the scalability of the implemented algorithm. We executed the program on the A1 partition of the Marconi machine at CINECA, Italy, which is the new supercomputer for the EUROfusion consortium. The A1 partition of Marconi is based on the Intel Xeon processor E5-2600 v4 product family (Broadwell) and consists of 2×18 cores at 2.30 GHz per node. Unfortunately the number of cores per node is not a power of two which makes it difficult to match the number of assigned cores (the number of nodes ×36) with the number of MPI tasks (a power of two in

many cases. In Fig. 46 and Fig. 47 we show the solution times of the multigrid method as a solver with a Gauss-Seidel smoother. We show results for both, the FDM (in black) and the FEM (in red). Within each figure we make a further distinction between case (a) a unit cube domain (1×1×1) and case (b) an elongated rectangular parallelepiped domain (1×1×8). In Fig. 46 we focus on strong scaling with three different mesh sizes. For case (a) we use the following finest meshes: $2^7 \times 2^7 \times 2^7$, $2^8 \times 2^8 \times 2^8$ and $2^9 \times 2^9 \times 2^9$ and for case (b) we use the following finest meshes: $2^7 \times 2^7 \times 2^9$, $2^8 \times 2^8 \times 2^{10}$ and $2^9 \times 2^9 \times 2^{11}$. In Fig. 47 we consider a quasi weak scaling where we have fixed the number of DoF per core. For case (a) and (b) we use the same DoF per core: $2^5 \times 2^5 \times 2^5$, $2^6 \times 2^6 \times 2^6$ and $2^7 \times 2^7 \times 2^7$. Thus, the number of MPI tasks for the parallelepiped domain (1×1×8) is larger by a factor of eight compared to the cube domain (1×1×1).

First of all we can see that the solution time on the unit cube and the elongated rectangular parallelepiped domain are very similar for both the strong and weak scaling and cases (a) and (b). Next we can see that the solution time of the FEM is always longer than the solution time of the FDM even if the required number of iterations of the FEM is less than for the FDM. This behavior might be caused by the differing number of nonzero elements of the generated matrices, i.e., 27 for the FEM and 7 for the FDM. In Fig. 46 we can see good strong scaling properties only for small numbers of MPI tasks. The larger the number of MPI tasks becomes the more the scaling degrades which clearly points in the direction of an increasing overhead in communication. In Fig. 47 we can see good weak scaling properties for the largest numbers of DoF per core ($2^7 \times 2^7 \times 2^7$) with a degradation of the scaling for smaller numbers of DoF per core.

5.4.2. **Including the merging step and an MPI/OpenMP hybrid model**

We have completed the parallel multigrid method by implementing a merging step and a parallel MPI/OpenMP hybrid model. From the previous numerical tests, we have learned that the numerical behavior of the multigrid method on the unit cube domain and on the elongated rectangular parallelepiped domain is very similar. Therefore, from now on we consider only tests for the case of a parallel multigrid solver with a Gauss-Seidel smoother on the unit cube domain.

First, we consider the MPI/OpenMP hybrid model on Broadwell (Marconi partition A1). We solve a problem with a mesh size of $2^8 \times 2^8 \times 2^8$ with a Dirichlet boundary condition on the whole boundary by using the FDM and the FEM as a test problem. We run four MPI tasks with different numbers of threads each, up to nine on Broadwell with 36 cores, i.e., on a single node. solution times (—) and the speedup (•••) in black for the FDM and in red for the FEM. For comparison, we show the solution times in seconds (×××) and the speedup (○○○) for the same number of cores of a pure MPI run with 4 × (number of threads) MPI tasks. From the experimental results in Fig. 48, we can see that the solution times for the FEM are slower than for the FDM. In addition, the solution times for both cases are gradually decreasing up to a total factor of one third as the number of threads approaches nine. For the pure MPI case, the solution times for both cases decrease rapidly by a factor of one third when reaching four threads, i.e., a total of 16 MPI tasks. For even larger numbers of threads the solution time only slightly decreases. For all cases on a single node the pure MPI parallelization has a better performance than the hybrid parallelization.

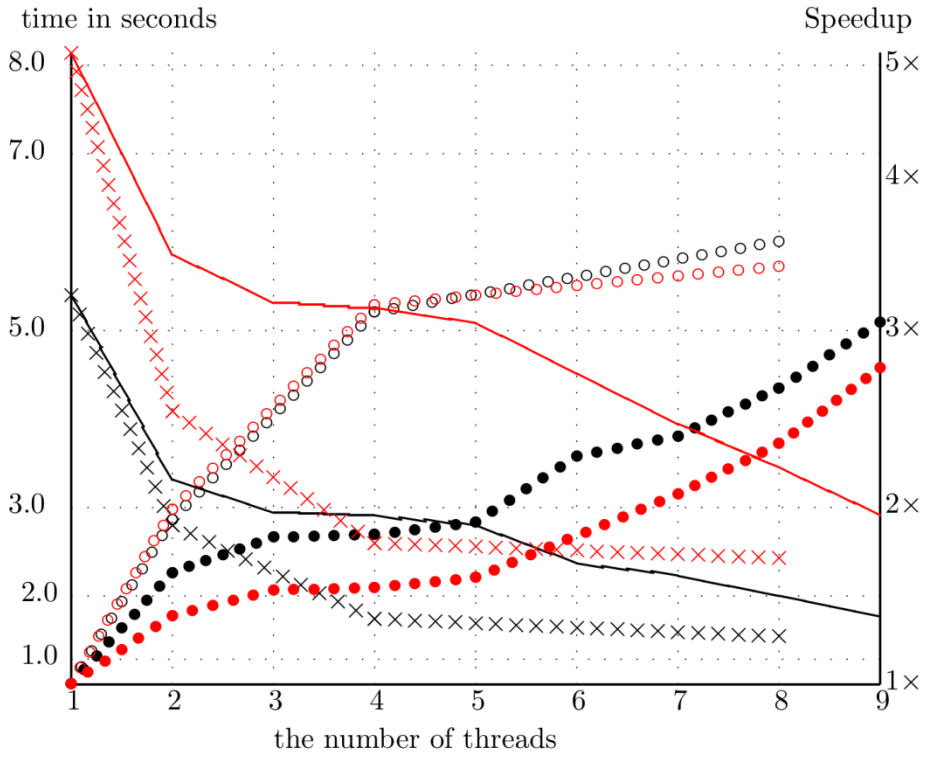
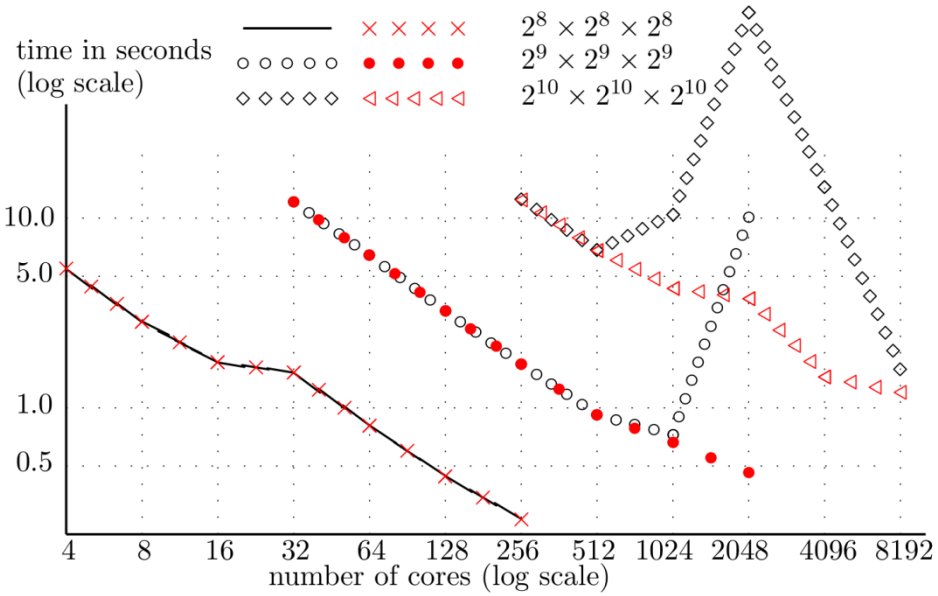


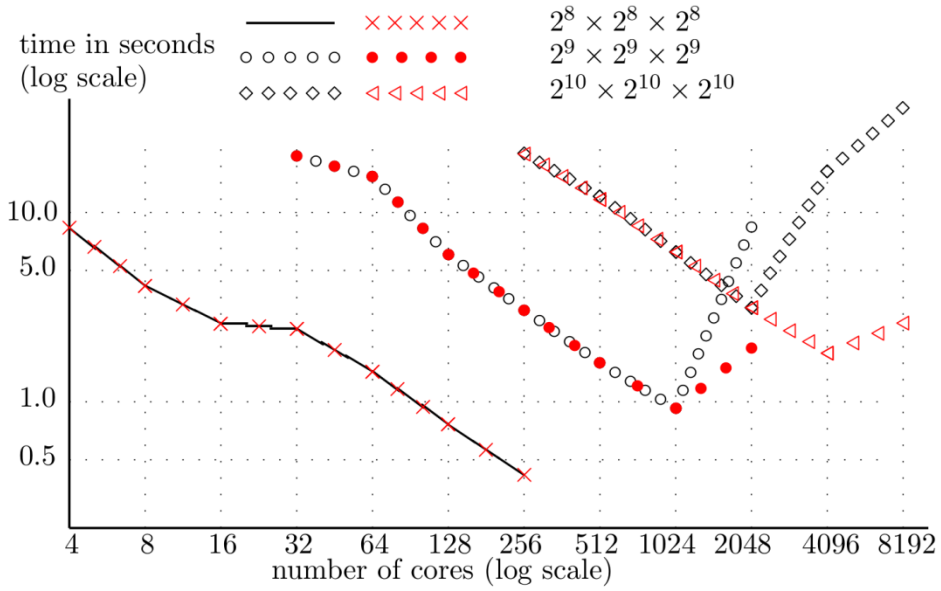
Fig. 48 The solution time in seconds (—) and the speedup (•••) of the multigrid method with a Gauss-Seidel smoother as a function of the number of threads per MPI task with four MPI tasks for the FDM (in black) and for the FEM (in red). The problem size is a $2^8 \times 2^8 \times 2^8$ mesh. For comparison, we also report on solution time in seconds (×××) and the speedup (○○○) of a pure MPI run with the same number of cores (corresponding to 4 × number of threads).

Now, we consider a large number of MPI tasks, i.e., more than 1000 MPI tasks, for which we expect to have a higher overhead for MPI communication. We report on strong and weak scaling behavior for several cases in the next figures. Due to the memory size, there is a limitation of the problem size which can be stored within a node. We found out that the biggest problem size on a Broadwell node, which has 118 GB of available memory, is a problem with a $2^9 \times 2^9 \times 2^9$ mesh. Due to this limitation, we test three different problem sizes, i.e., a $2^8 \times 2^8 \times 2^8$ mesh executed on 4 cores, a $2^9 \times 2^9 \times 2^9$ mesh executed on 32 cores, and a $2^{10} \times 2^{10} \times 2^{10}$ mesh executed on 256 cores. We use 32 cores per node except for the case of 8192 cores. In this case we use 36 cores per node on 228 nodes to be able to run the test on Marconi. For investigating the hybrid parallel model, we choose the best solution time among one, two, four, and eight threads per MPI task with an identical number of cores (plot in red in Fig. 49 and Fig. 50). We use up to 4 threads for the case of 8192 cores, i.e., 8192 MPI tasks with one thread, 4096 MPI tasks with two threads, and 2048 MPI tasks with four threads. We report on strong scaling property in Fig. 49 and the weak scaling property in Fig. 50.

Comparing the pure MPI case in Fig. 49 and Fig. 50 with pure MPi case with Fig. 46 (a) and Fig. 47 (a), we now achieve faster solution times due to the implementation of the merging step. The numerical results in Fig. 49 and Fig. 50 show that the hybrid model has a better performance than the pure MPI parallelization when the number of cores is larger than 1024 cores. The larger the number of cores the smaller the number of DoF per core becomes. The results also show that the parallel multigrid solver has a very good strong and weak scaling behavior up to 8000 cores.

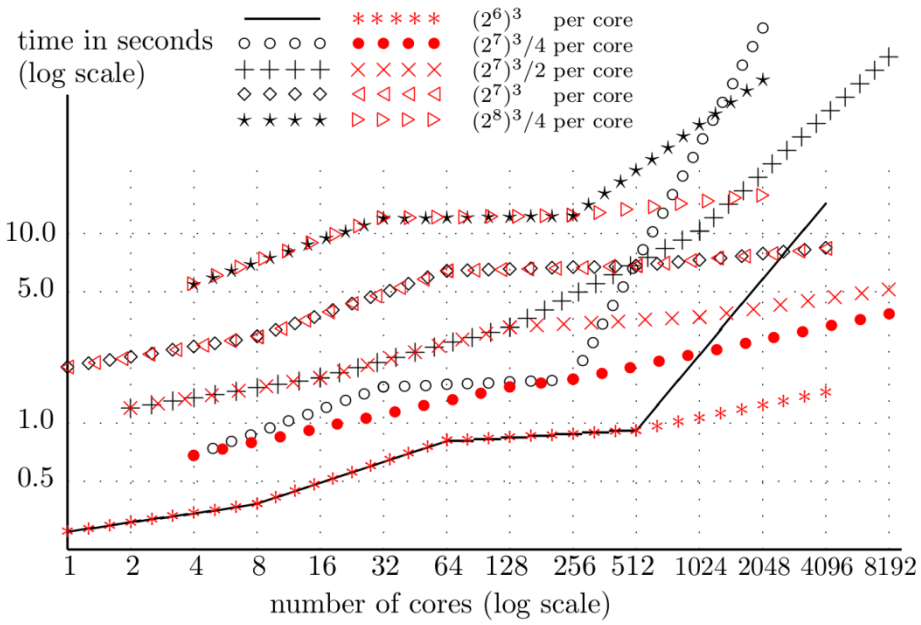


(a) FDM

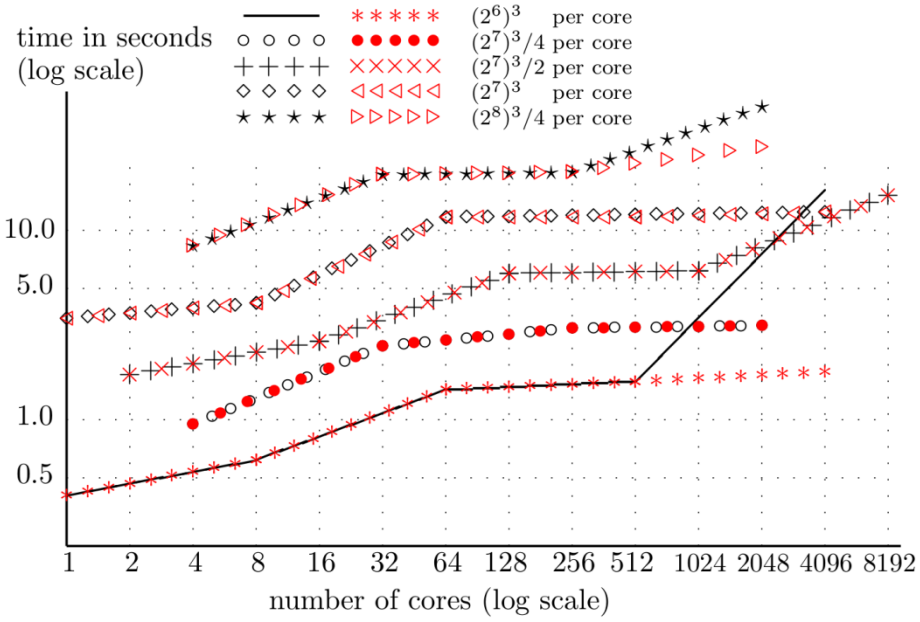


(b) FEM

Fig. 49 The solution time on the unit cube domain in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores. Pure MPI results in black and hybrid MPI/OpenMP results in red.



(a) FDM



(b) FEM

Fig. 50 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores for the case of an identical number of DoF per core. Pure MPI results in black and hybrid MPI/OpenMP results in red.

Next, we consider the performance on Knights Landing (KNL, Marconi partition A2). KNL refers to the Intel Xeon Phi processor which has a Many Integrated Core (MIC) architecture. It is available as a standalone processor and has 68 Silvermont-based cores at 1.4 GHz per node which sums up to more than three TFlops per node. It offers 16 GB MCDRAM with more than 400 GB/s of bandwidth (BW) on package and 96 GB DDR memory with 90 GB/s of BW system memory. It can use up to four hyper-threads per core and is binary compatible with the Intel Xeon processor. A KNL node can run in cache mode where the MCDRAM is considered as another cache level or in flat mode. As a result, the available memory consists of 96 GB (effectively 86 GB) in cache mode and 112 GB (effectively 101 GB) in flat mode.

We solve the same problem as we did on Broadwell but focus on using the FDM. We run with four MPI tasks and different numbers of threads, up to seventeen on KNL (which has 68 cores), i.e., on a single node. We report on solution times (—) and speedup (•••) in Fig. 51 in black in cache mode in red in flat mode.

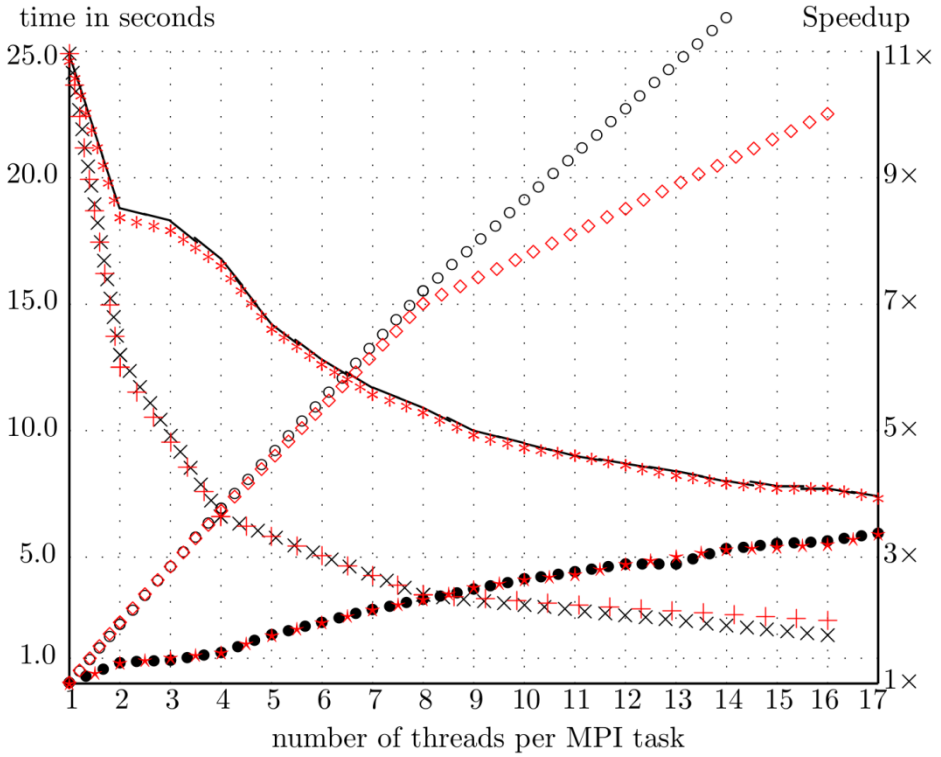
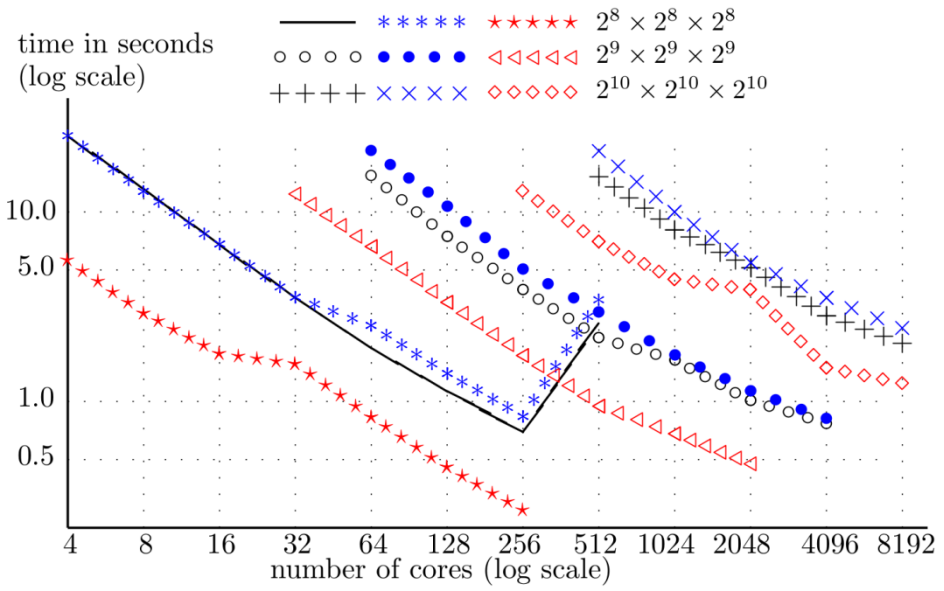


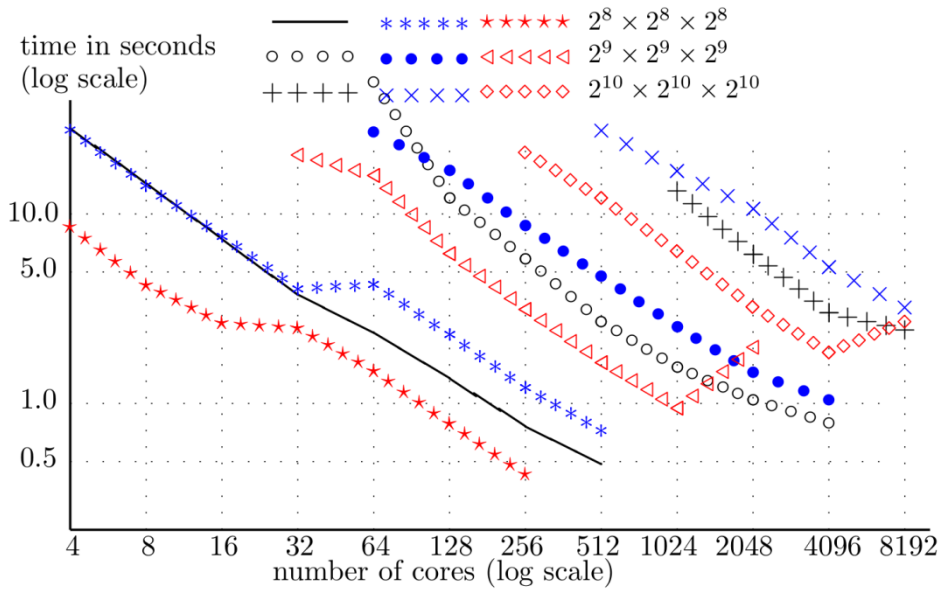
Fig. 51 The solution time in seconds (—, ×××) and the speedup (•••, ***) of the multigrid method with a Gauss-Seidel smoother as a function of the number of threads with four MPI tasks for the FDM in black in cache mode and in red in flat mode. The problem size is a $2^8 \times 2^8 \times 2^8$ mesh. For comparison, we report on solution time in seconds (×××, +++) and the speedup (○○○, ◊◊◊) of a pure MPI run with the same number of cores (corresponding to 4 × number of threads).

The results show that the solution time on KNL with four MPI tasks is five times slower than on Broadwell (shown in Fig. 48). The numerical behavior of the two memory modes on KNL is very similar, i.e., there is no significant difference of the two modes on one node. The speedup with 17 threads which uses all cores of the KNL node is approximately 3.3 and is similar to the multithreaded version on Broadwell. In comparison to a run using pure MPI tasks, the hybrid MPI/OpenMP version is relatively slow on KNL. This is caused by the large speedup of the pure MPI parallelization on KNL (10 to 12 times with sixteen threads).

Next, we investigate the strong and weak scaling properties on KNL, comparing both memory modes using the same problems as on Broadwell. We may expect similar numerical results compared to the numerical results on Broadwell when we use the same number of nodes because the clock speed of KNL cores is almost half (1.4 GHz vs. 2.3 GHz) the clock speed of Broadwell cores and the number of cores per node is almost doubled (68 vs. 36). In addition, the amount of memory is comparable as well. We solve the same problem sizes on KNL for both memory modes. Unfortunately we cannot solve the problem with a $2^8 \times 2^8 \times 2^8$ mesh per node using FEM because this problem needs too much memory to be able to store the discretized matrix.



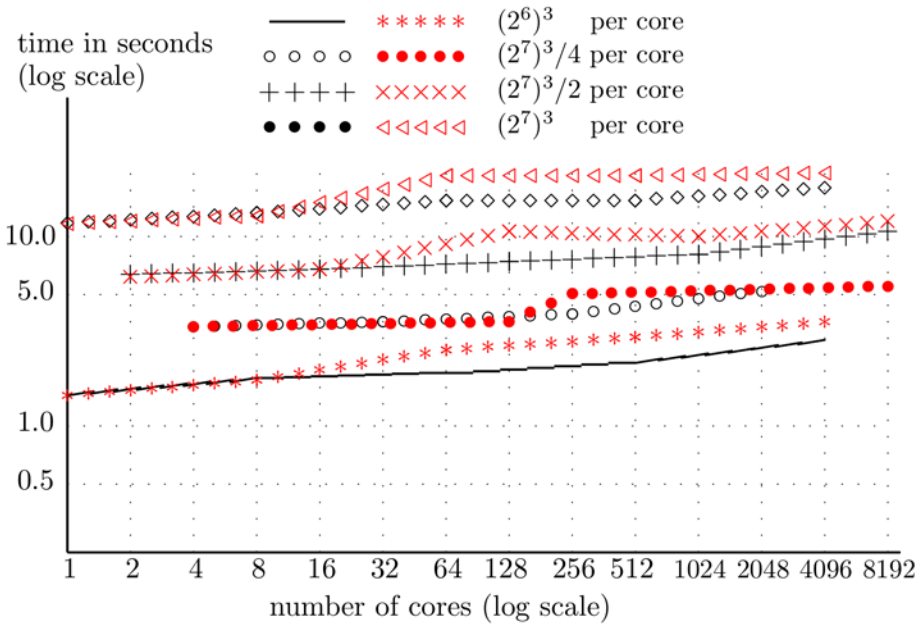
(a) FDM



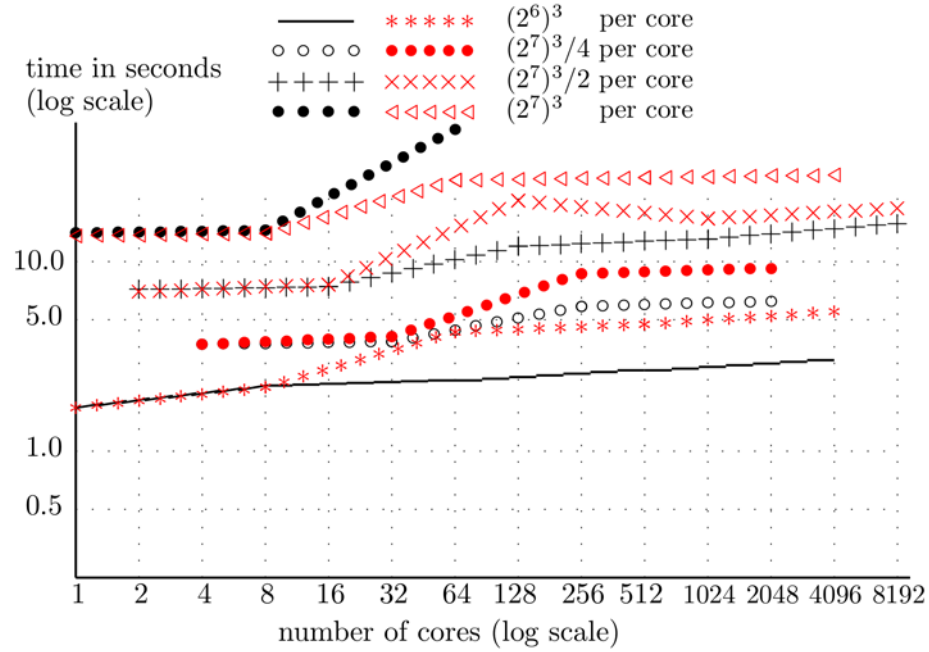
(b) FEM

Fig. 52 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores on KNL with cache mode in black and with flat mode in blue, on Broadwell in red as a reference case.

We report on solution times for three problem sizes, a problem with a $2^8 \times 2^8 \times 2^8$ mesh executed on 4 cores, with a $2^9 \times 2^9 \times 2^9$ mesh executed on 64 cores, and with a $2^{10} \times 2^{10} \times 2^{10}$ mesh executed on 512 cores, on KNL with cache mode in black and with flat mode in blue (see Fig. 52). We notice that there is no improvement of the hybrid MPI/OpenMP model up to 8192 cores (128 nodes). So we only report on case of a pure MPI parallelization. For comparison, we plot the solution time on Broadwell in red. From the results in Fig. 52, we see a similar behavior for both modes. The solution time in cache mode is faster than in flat mode when we use more than one node. The only case when the solution time in flat mode is faster than in cache mode is the case with a $2^9 \times 2^9 \times 2^9$ mesh on 64 cores (one node) using the FEM. The solution times on KNL with cache mode are similar to the ones on Broadwell using the same number of nodes. This matches our expectation. From these results, we conclude that the solver on KNL has a good strong scaling property up to 8192 cores.



(a) FDM



(b) FEM

Fig. 53 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores for the case with the same number of DoF per core on KNL in cache mode (in black) and in flat mode (in red).

Finally, in Fig. 53, we report on solution times as a function of the number of cores for the case with an identical number of DoF per core for four different cases. From the results we notice that the solution times in cache mode for more than one node (64 cores) are faster than in flat mode. However, we can say that the parallel multigrid solver on KNL for both memory modes has a good weak scaling property up to 8192 cores.

5.4.3. **Benchmark on the Marconi A3 partition**

Last August the third partition of Marconi called A3, which hosts 1512 nodes each equipped with two Intel Xeon 8160 processor (Skylake, SKL), was taken into operation. Each 24-core Intel Xeon 8160 CPU has a base clock speed of 2.1 GHz

(capable of 32 double-precision FLOPs/cycle), providing a theoretical peak performance of about 2.23 TFLOPs per node. Each node has 192 GB of DDR4 RAM and is connected via a 100 GB/s Intel OmniPath 2:1 interconnect. The processor supports vectorisation instructions such as SSE and AVX up to AVX-512.

During the preproduction phase of the Marconi A3 partition, we performed a benchmark with the current multigrid solver and compared the results with the ones from the other partitions. We used the FDM with a $2^{10}\times2^{10}\times2^{10}$ mesh and ran it on the A1 partition using 32 cores (out of 36 cores) per node and a hybrid MPI/OpenMP model, on the A2 partition using 64 cores (out of 68 cores) per node, and on the A3 partition using 44 cores (out of 48 cores) per node.

We report on solution time in Table 14 and Fig. 54 (in relation to the number of cores in black and to the number of nodes in red). For the same number of cores, the numerical results show that the solution time on KNL is the slowest which is expected because of the clock frequency of the CPU. The numerical results also show that the solution time on SKL is faster than on Broadwell even though the clock frequency of the Broadwell CPU is slightly higher than the clock frequency of the SKL CPU.

	A1		A2		A3	
# of cores	time [s]	# of nodes	time [s]	# of nodes	time [s]	# of nodes
128					19.704	3
256	12.4489	8			9.7631	6
512	6.6276	16	15.4459	8	5.1863	12
1024	4.1733	32	7.9523	16	2.6648	24
2048	3.665	64	4.9853	32	1.5371	48
4096	1.4254	128	2.8518	64	1.7287	96
8192	1.1763	228	1.9965	128		

Table 14 The solution time of the 3D parallel multigrid algorithm with a Gauss-Seidel smoother on a unit cube domain with a $2^{10}\times2^{10}\times2^{10}$ mesh.

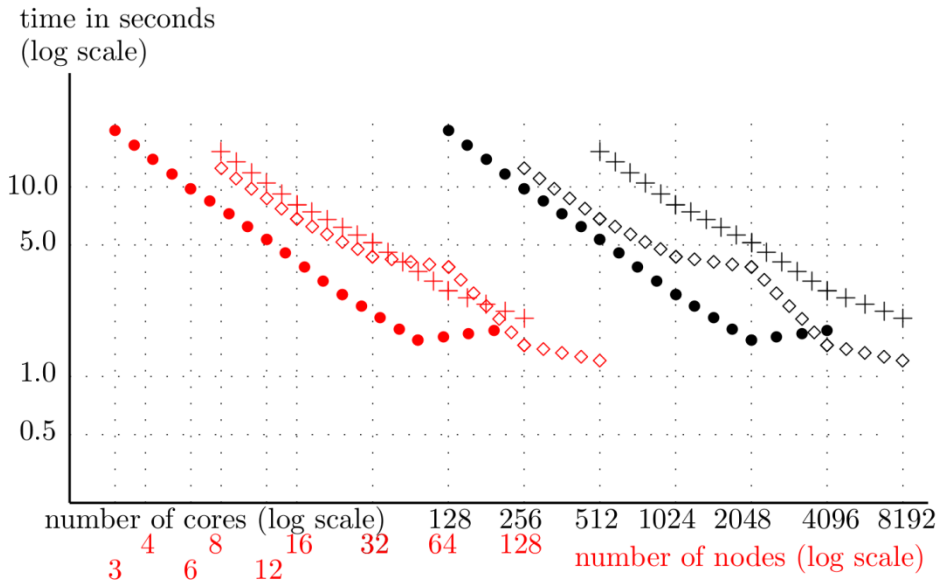


Fig. 54 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a function of the number of cores (in black) and the number of nodes (in red) for the FDM with a $2^{10}\times2^{10}\times2^{10}$ mesh. We show results of KNL (A2 partition) in cache mode (+++), of SKL (A3 partition) (•••), and of Broadwell (A1 partition) (◊◊◊).

The purpose of this benchmark is to show the performance improvement on SKL in a node to node comparison with KNL and Broadwell. Previously we have achieved almost the same node to node performance on KNL and Broadwell. We get a slight

	A1	A2		A3 *	
# of nodes	time[s]	time[s]	A2/A1	time[s]	A3*/A1
8	12.4489	15.4459	1.24	8.2375	0.66
16	6.6276	7.9523	1.20	4.3458	0.66
32	4.1733	4.9853	1.19	2.2889	0.55
64	3.665	2.8518	0.78	1.6010	0.44
128	1.4254	1.9965	1.40		

Table 15 The relative solution time of the 3D parallel multigrid algorithm with a Gauss-Seidel smoother on a unit cube domain with a $2^{10}\times2^{10}\times2^{10}$ mesh. The values achieved on the A3 partition are interpolated from the two nearest numbers of cores.

5.5. *Current status and future work*

A parallel 3D multigrid solver for the Poisson problem, using either the finite difference (FDM) or the finite element method (FEM), has been implemented. We have completed the parallel multigrid method with a merging step using a hybrid MPI/OpenMP parallelization scheme. The numerical study of the current implementation shows that the 3D multigrid solver needs, as expected, a fixed number of iterations. This is clear evidence that the implementation is correct.

We tested the solver on two domains with different shapes, one domain was a cube domain and the other one was an elongated rectangular parallelepiped domain, with cube cell elements. We found that the required number of iterations is identical for both cases. We also found that the required number of iterations for problems with different types of boundary conditions, i.e., a problem with a Dirichlet boundary condition everywhere and a problem with Neumann boundary conditions along one direction and Dirichlet boundary conditions along all other directions, is the same.

The required number of iterations for the solver for the FDM is larger than for the FEM. However the solution time for the FDM is faster than for the FEM because the generated matrices for the FDM have only seven nonzero elements, but the generated matrices for the FEM have 27 nonzero elements. Using the merging step, we are also able to reduce the solution times for simulations with a large number of MPI tasks. So we concluded that the merging step is an essential component in the parallel multigrid method.

On a single Broadwell node, the hybrid MPI/OpenMP parallelization shows no benefit in comparison with the pure MPI parallelization. However, we see a significant performance improvement on more than 1000 cores. On a single KNL node, the pure MPI parallelization has outstanding performance in comparison to the hybrid MPI/OpenMP parallelization. This phenomenon continues up to 8000 cores where we still don't measure any benefit from the hybrid MPI/OpenMP parallelization on a KNL system. The solution times using cache mode are identical to the ones using flat mode on a single KNL node, but on more than two nodes (more than 128 cores) cache mode is slightly faster than flat mode. We showed that in a node to node comparison, the performance on Broadwell and on KNL is similar and we get good scaling properties up to 8000 cores on both systems. In addition, we tested the multigrid solver on the Marconi A3 partition (SKL) and achieved a performance improvement of about 33% for the same number of nodes compared to the Broadwell (A1) and the KNL (A2) partitions.

The first part of the project has been finished by developing a 3D parallel multigrid solver. In the remaining part of the project we will continue the work. This includes a flexible boundary condition for the 2D and 3D parallel solvers of the BIT2 and the BIT3 codes.

6. **Final report on HLST project CINCOMP2 – Part 1**

The CINCOMP2 project is a continuation of the CINCOMP project [1] and dedicated to provide support for European scientists who use the Marconi machine located at CINECA. The Marconi supercomputer was launched in July 2016. The official production phase started in mid-October 2016. The first partition has the code name A1 and is based on the Intel Xeon processor E5-2600 v4 product family (*Broadwell*). An additional partition was added at the beginning of 2017 named A2, equipped with the next-generation of the Intel Xeon Phi product family (*Knights Landing*). In the framework of this project, both, the hardware and the software of the A1 and A2 partitions were tested using a variety of benchmarks. As expected for a new supercomputer, many issues were found and subsequently reported to the Marconi support team via the ticket system.

6.1. *The Marconi supercomputer architecture*

The Marconi supercomputer is located in Bologna in the largest Italian computing centre named CINECA. The machine is planned to be gradually constructed during 16 months of operation, between April 2016 and July 2017, implementing three upgrades, named A1, A2 and A3. The A1 partition, which is based on the Intel Xeon processor E5-2600 v4 product family (*Broadwell*), went into its official production phase in mid-October 2016. A total of 1512 computing nodes were assembled providing a computational power of 2 Pflop/s. This system was tested in our previous project CINCOMP [1]. The A1 partition is in full production now. The second upgrade was made in mid-January 2017, during which a new section, equipped with the latest generation processors of the Intel Xeon Phi product family (*Knights Landing*), which is based on the many integrated core architecture (68 cores), was added. The new section (A2) with about 250,000 cores provides additional computational power of about 11 Pflop/s. The tests in this report will focus mainly on this partition. The last upgrade is planned for July 2017. It will incorporate Intel Xeon *Skylake* processors. This update (A3) should allow to reach an overall computing power of about 20 Pflop/s. The european fusion community will only have access to the so-called Marconi-Fusion part of Marconi.

6.2. *Intel Broadwell vs Knights Landing architecture*

Knights Landing (KNL) is the code name for the second generation MIC architecture product from Intel. In comparison to an Intel *Broadwell* (A1) computing node which is equipped with 2 CPUs, a KNL node (A2) has only one processor. Table 16 shows a comparison of the main hardware characteristics between an Intel *Broadwell* and an Intel Knights Landing node which are installed on Marconi. In spite of the lower CPU frequency used for the Intel KNL architecture the peak performance of KNL is about 2.27 times higher in comparison to Intel *Broadwell* due to a larger number of cores and twice larger vector registers. A KNL node consists of two types of main memory: (i) MCDRAM - providing more than 400 GB/s bandwidth and (ii) DDR4 delivering 115.2 GB/s bandwidth. The bandwidth of the MCDRAM memory is much higher (2.6 times) than the bandwidth of an Intel *Broadwell* CPU. However, the capacity of the MCDRAM memory is relative low (16 GB). Therefore, if one uses data from DDR4 memory (96 GB) the bandwidth is lower in KNL in comparison to *Broadwell*.

Processor	Intel KNL	Intel Broadwell
Number of cores	68	2×18
Memory	96 GB DDR4 + 16 GB on chip MCDRAM	2×64 GB
AVX base frequency	1.2 GHz	2.0 GHz
FMA units	2	2
Vector register	8 doubles	4 doubles
Peak performance	2610 GFlop/s	1150 GFlop/s
Memory bandwidth	115.2 GB/s–DDR4 445 GB/s–MCDRAM	153.6 GB/s

Table 16 Hardware characteristics comparison between Intel *Knights Landing* and Intel *Broadwell*.

6.3. *Memory bandwidth test*

6.3.1. Knights Landing vs Broadwell

The theoretical memory bandwidths according to the vendor's specification are 115.2 GB/s and >445 GB/s for the Intel KNL DDR4 and MCDRAM memory, respectively. The theoretical memory bandwidth for the Intel *Broadwell* is 153.6 GB/s. However, all these values are not reachable in practice, thus we have to measure the real memory bandwidth by means of different benchmarks. The Stream benchmark [2] is one of the most popular benchmark in the high performance computing community for doing so.

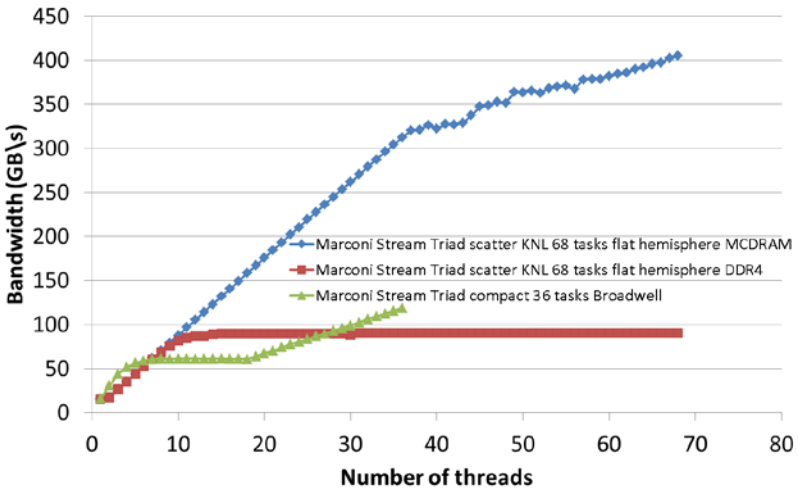


Fig. 55 Memory bandwidth of a KNL and Broadwell node on Marconi.

Fig. 55 presents the results obtained with the Stream benchmark on a single Marconi *Broadwell* and KNL node. The maximum bandwidth of the DDR4 memory on the KNL node is about 90.3 GB/s (red line) which is lower than the maximum bandwidth that can be obtained from a *Broadwell* node – 118.8 GB/s (green line). However, using data fitting into the MCDRAM memory (16 GB) can significantly increase the bandwidth. The MCDRAM memory from the KNL node delivers a maximum bandwidth of 405.1 GB/s. Therefore, if a code is memory bound and its memory consumption is larger than 16 GB, the Intel *Broadwell* node should provide faster results. In all other cases the Intel KNL node should work better.

6.3.2. KNL Stream benchmark

The Stream benchmark discussed above provides four tests for measuring the memory bandwidth named Copy, Scale, Add and Triad [2]. Fig. 56 shows the results of these four tests obtained when the Stream benchmark was launched on a KNL

node. We found that three numerical benchmarks (Copy, Add and Triad) have a similar behavior and produce a high bandwidth between 393 and 443 GB/s. However, the Scale test delivers a much smaller bandwidth than the others (309 GB/s). This behavior is not well understood because the Scale test is very similar to the Add test. Therefore, the issue was reported to the Marconi support team and it is still under investigation.

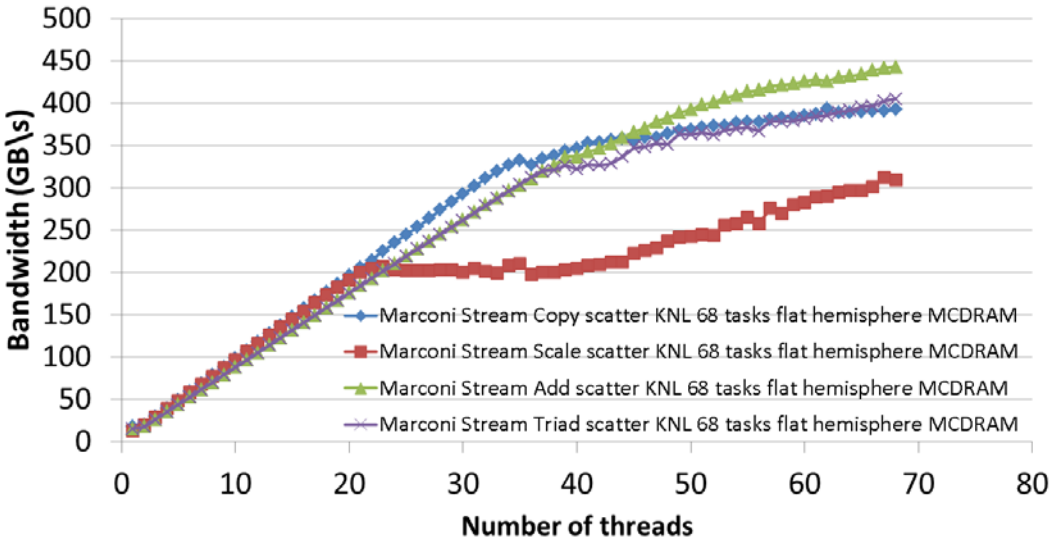


Fig. 56 Result of the Stream benchmark on a single Marconi KNL node.

6.3.3. **KNL different memory modes check**

The KNL nodes on Marconi can be used in two different memory modes (flat and cache) and three different cluster modes (quadrant, hemisphere and snc2). The memory modes define the way in which the DDR4 and MCDRAM memory is visible to the software. In cache mode MCDRAM behaves as a memory-side direct mapped cache above the DDR4. In flat mode, the MCDRAM is visible to the operation system as a separate NUMA node. The cluster modes define different ways of how the memory requests from the cores will be treated by the memory controllers. For more details we refer to [3].

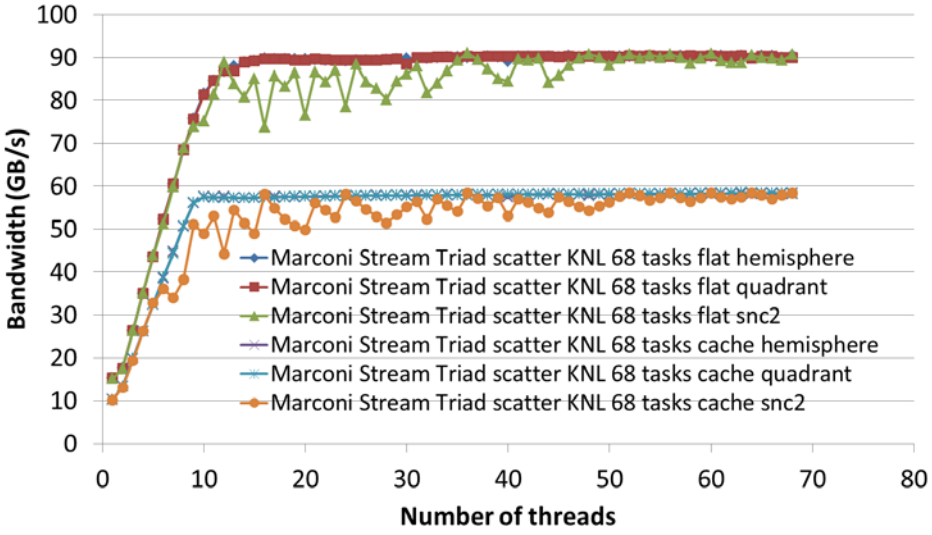


Fig. 57 Result of the Stream Triad benchmark on a single Marconi KNL node for different memory and cluster modes.

Fig. 57 shows the memory bandwidth obtained from the Stream benchmark executed on the Marconi KNL node for two different memory and three different cluster modes. The total size of the Stream arrays was 72 GB for these tests. One can see that in

flat memory mode the bandwidth is higher by a factor 1.5 in comparison to cache memory mode. This means, that, if a users data size is larger than the MCDRAM memory (16 GB), it is better to use flat memory mode. We found no large difference between the clustering modes. The snc2 mode is fluctuating a bit for low numbers of threads, however when a complete node is used (68 threads) the bandwidth is identical to the other cluster modes.

6.4. *Marconi login node test*

The possibility to login to the Marconi supercomputer via a secure shell (ssh) was continuously tested starting from the end of January 2017. This test provides information about the times the machine is in operation. For this purpose a special script was developed by Jacques David which is launched by the cron job scheduler every 10 minutes. Fig. 58 presents the results from this test. One can see that in the period from January 27 to February 16 the machine was out of production four times. Two periods were short but the other two took about an hour. We continue to monitor the status of the machine and will update the results accordingly.

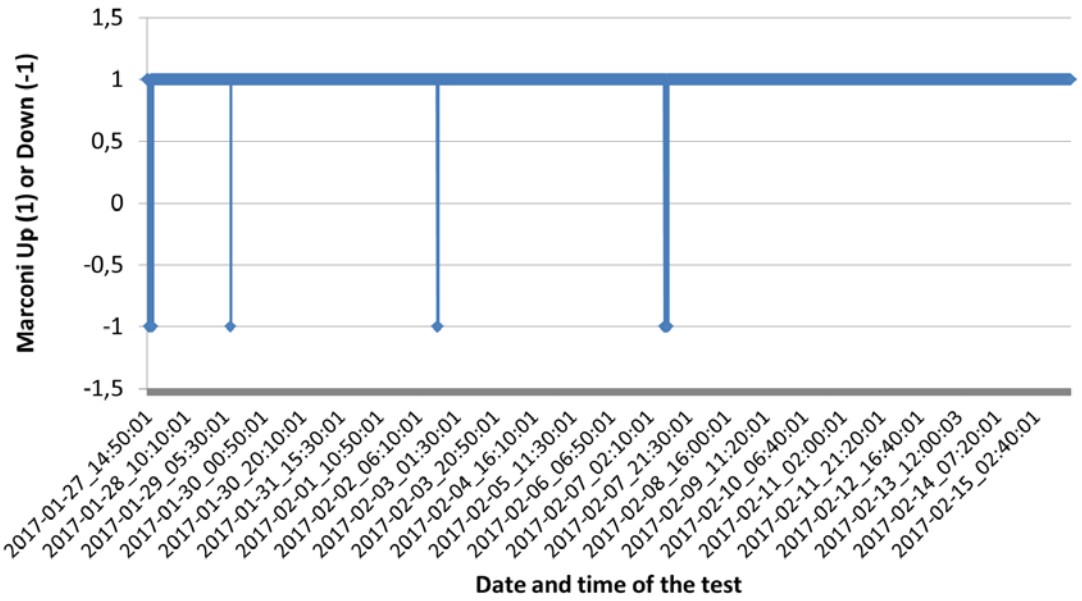


Fig. 58 Marconi login test: 1 corresponds to a working machine; -1 stands for a state when the machine did not work.

6.5. *Check for ailing computing nodes on Marconi*

During the transfer of the GYGLES code on Marconi it was detected that some nodes provided results much faster than others. Afterwards we figured out that these nodes did not work properly. Therefore, the following test was developed in order to identify ailing nodes and exclude them from the system.

The test includes two benchmarks: Intel SMP Linpack [4] and Stream [2]. The first benchmark solves a dense system of linear equations and measures the execution time. Afterwards the benchmark converts the measured time into performance (Flops/s) and checks the results for accuracy. The Stream benchmark discussed in detail in the previous report [1] measures the memory bandwidth (GB/s). Both benchmarks are dedicated to test a single computing node. Hence, each benchmark was launched simultaneously on many nodes (> 200) and the results were compared. If the deviation of the obtained results was less than a few percent it was assumed that all nodes were working properly. If some nodes produced results which were significantly different than the results of the others we assumed that these nodes were ailing. The test was launched once a week on both A1 (Intel *Broadwell*) and A2 (Intel KNL) partitions of Marconi.

6.5.1. **Check for ailing computing nodes on the Intel Broadwell partition of Marconi**

Fig. 59 shows the performance obtained from the Intel SMP Linpack benchmark launched on the A1 partition. The obtained results are quite similar with only a deviation of less than two percent between the nodes. Moreover, all nodes provide a very high performance of 1115±12 GFlops/s which is about 96 percent of the peak performance (Table 16). The Stream benchmark also provides impressive results with a bandwidth of ~117 GB/s which is about 76 percent of the maximum bandwidth specified by the vendor. Therefore we conclude that all tested nodes were working well when we took the measurements.

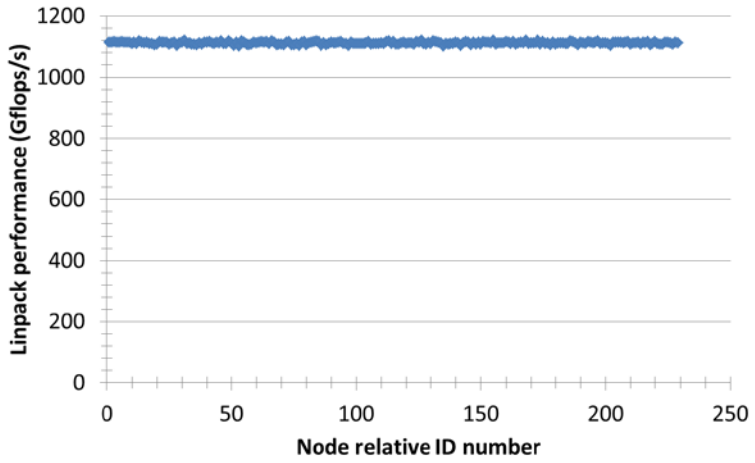


Fig. 59 Performance obtained from the Intel SMP Linpack benchmark launched on the Marconi *Broadwell* partition (February 3, 2017).

6.5.2. **Check for ailing computing nodes on the Intel KNL partition of Marconi**

The test discussed above was also applied to identify ailing computing nodes on the Marconi KNL partition. Fig. 60 presents the performance obtained from the Intel SMP Linpack benchmark launched on the Marconi A2 partition (February 3, 2017). We can clearly see from the figure that five nodes provided a much smaller performance than the others. We reported this issue to the Marconi support team. They performed additional tests and confirmed the problem. Finally, these five nodes were excluded from the system.

The average performance of the healthy nodes was measured to be about 1850 GFlops/s. This value is faster by a factor of ~1.6 than the performance measured on the Intel *Broadwell* partition. However, it is only 70 percent of the peak performance.

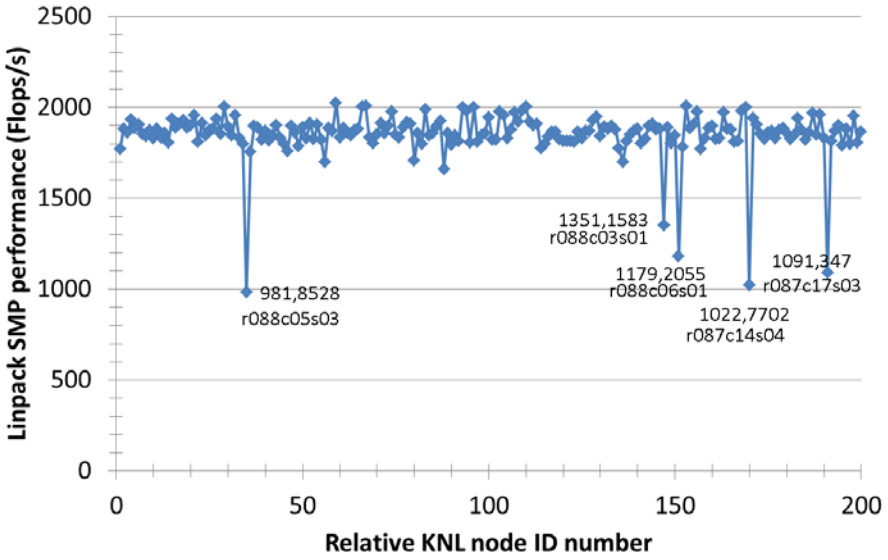


Fig. 60 Performance obtained from the Intel SMP Linpack benchmark launched on the Marconi KNL partition (February 10, 2017).

6.6. *Marconi network performance*

The PingPong test from the Intel MPI Benchmark suite [5] was used to test the network performance on both the Intel *Broadwell* (A1) and KNL (A2) partitions of Marconi. In this test an *N* byte message is sent from process one to process two by means of the *MPI_Send* and *MPI_Recv* functions. When the message is received, process two sends the same data back to process one. The communication time is calculated afterwards as $time = \Delta t/2$, where Δt is the time necessary for the transaction from one process to another and back. The test is repeated 100 times and the values are averaged. The latency is defined as the time spent to send a zero byte message.

First, the intra node network performance was tested for both partitions A1 and A2. The obtained results are compared with the ones measured on the HELIOS supercomputer that was based on the Intel *Sandy Bridge* CPU. All results are shown in Table 17. The latency between two processes running on the same *Sandy Bridge* CPU was 0.25 µs which is more than two times faster in comparison to the Intel *Broadwell* CPU (0.61 µs) and more than three times faster in comparison to the Intel KNL processor. The latency between two processes running on the same *Sandy Bridge* node (two different CPUs) is also faster by almost a factor of two (0.64 µs) in comparison to the Intel *Broadwell* node (1.09 µs).

Sandy Bridge CPU0	0.25
Sandy Bridge CPU1	0.64
Latency (µs)	Sandy Bridge CPU0

Broadwell CPU0	0.61
Broadwell CPU1	1.09
Latency (µs)	Broadwell CPU0

KNL CPU0	0.85
Latency (µs)	KNL CPU0

Table 17. The intra node network performance of: (i) the Intel Sandy Bridge partition on the HELIOS supercomputer; (ii) the Intel Broadwell partition on Marconi and (iii) the Intel KNL partition on Marconi.

The inter node network performance was tested next. For this test two MPI tasks were pinned on two different computing nodes. As in the intra node test the lowest latency was achieved between two Intel *Sandy Bridge* CPUs (1.13 µs). The latency slightly increases for the Intel *Broadwell* CPUs, up to 1.49 µs. Finally, the highest latency was measured between the Intel KNL CPUs: 3.99 µs.

The Intel MPI benchmark (IMB) also provides a measurement of the memory bandwidth which is shown in Fig. 61 for intra and inter node tests performed on both Marconi partitions (*Broadwell* and KNL). One can see that the problem being detected in the previous CINCOMP project [1] for the Intel *Broadwell* node is still not resolved. The memory bandwidth in the intra node test (green line) is smaller in comparison to the inter node test (violet line) for large messages (> 1MB). A similar problem was detected for the Intel KNL node. The inter node bandwidth grows with an increasing message size (red line). However, the bandwidth for the intra node test (blue line) significantly drops at a message size of about 3 kB and stays relatively low for all larger messages. This issue was reported to the Marconi support team and is still under investigation.

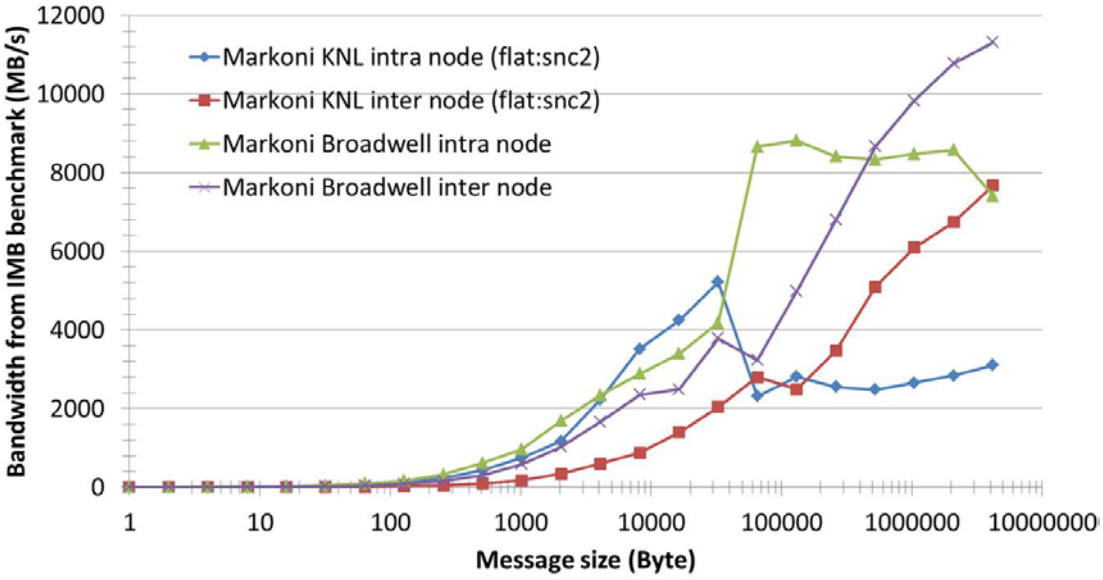


Fig. 61 Comparison of the inter and intra node memory bandwidth for the Marconi Broadwell and KNL computing nodes.

6.7. *Conclusions*

Marconi went into operation in July 2016 as the new supercomputer for the fusion community. It replaced the HELIOS machine at IFERC-CSC which was decommissioned at the end of 2016.

Different benchmarks and tests were made to determine the performance of both partitions available on Marconi named A1 (*Broadwell*) and A2 (KNL). Issues were found that significantly limited their use. Some of them were resolved by the Marconi support team, others however are still under investigation.

The Stream benchmark shows a very high memory bandwidth for the MCDRAM memory of the KNL node in comparison to Intel *Broadwell*. However, the memory bandwidth for DDR4 is slightly lower on KNL in comparison to Intel *Broadwell*.

The Intel MPI PingPong test for the intra node benchmarks on the A2 partition has revealed a drastic bandwidth drop for message sizes larger than 3 kB.

A combination of two benchmarks was applied to find ailing nodes on both partitions. Five of such nodes were detected on the Intel KNL partition. They are found to deliver much lower performance than the other nodes. Finally, these nodes were excluded from the production.

6.8. *References*

[1] S. Mochalskyy, HLST annual report 2016.
[2] https://www.cs.virginia.edu/stream/
[3]https://software.intel.com/en-us/articles/intel-xeon-phi-x200-processor-memory-modes-and-cluster-modes-configuration-and-use-cases#Quadrant

[4] https://software.intel.com/en-us/node/528615
[5] https://software.intel.com/en-us/articles/intel-mpi-benchmarks

7. Final report on the CINCOMP2 project – Part 2

7.1. *Introduction*

The CINCOMP2 project is the continuation of the CINCOMP project which was finished in late 2016. These projects are dedicated to provide support for specific issues on the supercomputers of the EUROfusion consortium.

7.2. *Performance of different common timing facilities on HPC resources*

When the new Skylake partition (A3) on the Marconi supercomputer came online, an unexpected behavior of the timing facilities was discovered. It resulted in degraded performance and unreliable results for certain code benchmarks. This made it important to investigate the possible timing facilities of HPC systems and decide on an optimal one for our use case.

7.2.1. Commonly available timing facilities

Modern HPC resources offer a variety of systems to accurately measure time. The new Skylake partition on Marconi is configured to use the High Precision Event Timer (HPET), while most other HPC resources in the world are configured to employ the Time Stamp Counter (TSC). This makes an investigation into these two systems necessary and important in order to find the best solution for the EUROfusion community.

The two timing facilities differ in various key properties. They sit at different locations on a server board. From its placement on the north or south bridge it is obvious that the HPET is a shared resource between all cores in a specific node. The TSC is core local and is not shared at all.

The TSC was used to count the cycles of the respective core. This has changed in modern Intel CPUs (like Skylake) in the regard that it now counts the wall time. This became necessary due to the different clock modes of a modern CPU, which had implications on the return values of the TSC. The new configuration (it now counts wall time) makes the TSC worse as a method to count the number of cycles a specific instruction takes but better in gauging the overall performance of a code. The value of the TSC may be accessed using the assembly instruction RDTSC (read time stamp counter).

The HPET was introduced as a system to accurately measure time across cores in a stable manner. It can also provide very accurate, configurable interrupts. This is necessary for many applications. Mainly multi-media applications, for example, video and music playing software, needs its capabilities. If the OS, for example, reschedules the participating processes it could severely interfere with the time-accurate playback of the respective media. It was not introduced as a timer until the Linux 2.6 kernel which hints at its usefulness in HPC systems.

All kernel functions which are related to timers will use a preconfigured timing facility. A restart of the system is necessary to switch between different facilities. The currently selected timer configuration is available via

```
cat /sys/devices/system/clocksource/*/current_clocksource.
```

7.2.2. Evaluation of the performance of the TSC and HPET timing facilities

In order to evaluate the performance of the timers and decide on a preference it is important to investigate the time a single read-out of a timer takes. This time should be at least one order of magnitude smaller than the evaluation time of any functions or algorithms which are being measured by the timer. Only then the resulting data can be trusted to be a valid representation of the performance of that function or

algorithm. A second very important property of timers is the amount of resources it needs and the scalability of a timer call. If the timer function is too expensive or does not scale properly, it may falsify the resulting timing information as the algorithm, including the timers, performs very differently from their timer-less versions.

For all measurements in this section the process skew elimination algorithm is employed. This algorithm is explained in detail in section 7.3.

For measuring the read-out time the code in Listing 1 was used.

```
for (int i = 0; i < iterations; i++) {
  MPI_Barrier(comm);
  struct timespec tv_start, tv_end;
  clock_gettime(CLOCK_MONOTONIC, &tv_start);
  clock_gettime(CLOCK_MONOTONIC, &tv_end);
}
```

Listing 1 Algorithm used to measure read-out time of the clock_gettime() kernel function.

The value of iterations was set to 10^7. An appropriate amount of warm-up cycles was employed. The final measurement value is given by $\max_{\forall \text{ranks}} \text{tv\_end} - \text{tv\_start}$. The general idea of this code snippet is illustrated in Fig. 62.

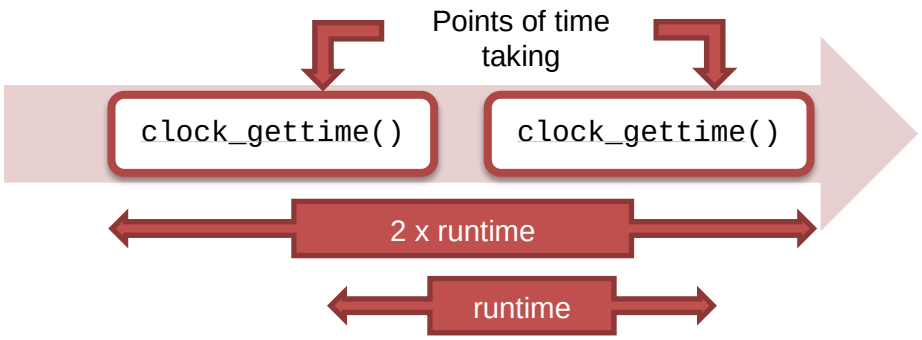


Fig. 62 Measurement scheme for measuring the read-out time of a timing facility. The resulting value $\max_{\forall \text{ranks}} \text{tv\_end} - \text{tv\_start}$ gives the total time one timer call takes. The actual location inside the clock_gettime function, where the timer read-out is taking place, is marked with “Points of time taking”. The scheme ensures that the clock_gettime internal position of this operation does not affect the results. The difference between the two reported times therefore equals the run time of the clock_gettime function.

The results of running this test with TSC and HPET can be found in Fig. 63. This benchmark is sensitive to the scalability as it calls the function on all cores in parallel. This is a common use case for applications as most scientific codes are SPMD (single program multiple data).

The results show that the mean performance of HPET is nearly three orders of magnitude worse than the performance of TSC. The total wall time for the TSC test was 1 minute and 41 seconds while it was 37 minutes and 6 seconds for the HPET test.

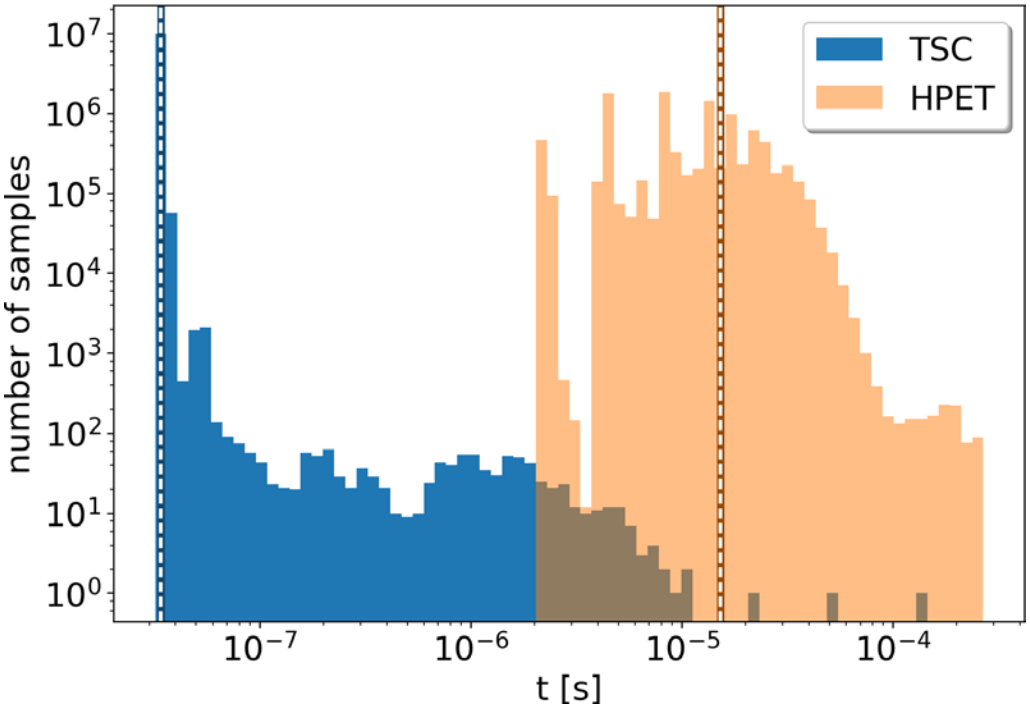


Fig. 63 Histogram of the time difference between two clock_gettime calls (global maximum) on one SKL node for 10^7 function calls for the two different timers TSC and HPET. 48 threads were used. The mean value of all samples is shown by the vertical dashed lines.

In the following tests we will measure the scalability directly. We also compare the general timer performance to the performance of two other functions, MPI_Wtime and empty_read. MPI_Wtime is the timing facility provided by the MPI library. It is very likely that it will call clock_gettime internally. It is nonetheless interesting to measure the overhead introduced by MPI. The second function, empty_read, is a 0 byte read call from STDIN. This function should switch to kernel space and almost immediately return to user space. A properly configured linux system will use so-called virtual kernel calls for the timer functions. This makes it possible to have them avoid the switch to kernel space altogether by perusing mappings of the timer related memory portions into user space. This is the reason that virtual kernel calls can be much faster than other kernel functions. We can test this by comparing the timer function calls to the empty_read calls. For the following measurements the code in Listing 2 will be used.

```
struct timespec tv_start, tv_end;
MPI_Barrier(comm);
clock_gettime(CLOCK_MONOTONIC, &tv_start);
for (int i = 0; i < iterations; i++) {
  func();
}
clock_gettime(CLOCK_MONOTONIC, &tv_end);
```

Listing 2 Algorithm used to benchmark different functions func.

The value of iterations is set to 10^6 and we again use appropriate warm-up cycles. The resulting value is given by (tv_end-tv_start) / iterations. This benchmark algorithm produces the mean run time value of func, but is not able to produce the run time spread. In exchange for this drawback, the clock impact is reduced to a minimal amount. The result of this benchmark is shown in Fig. 64.

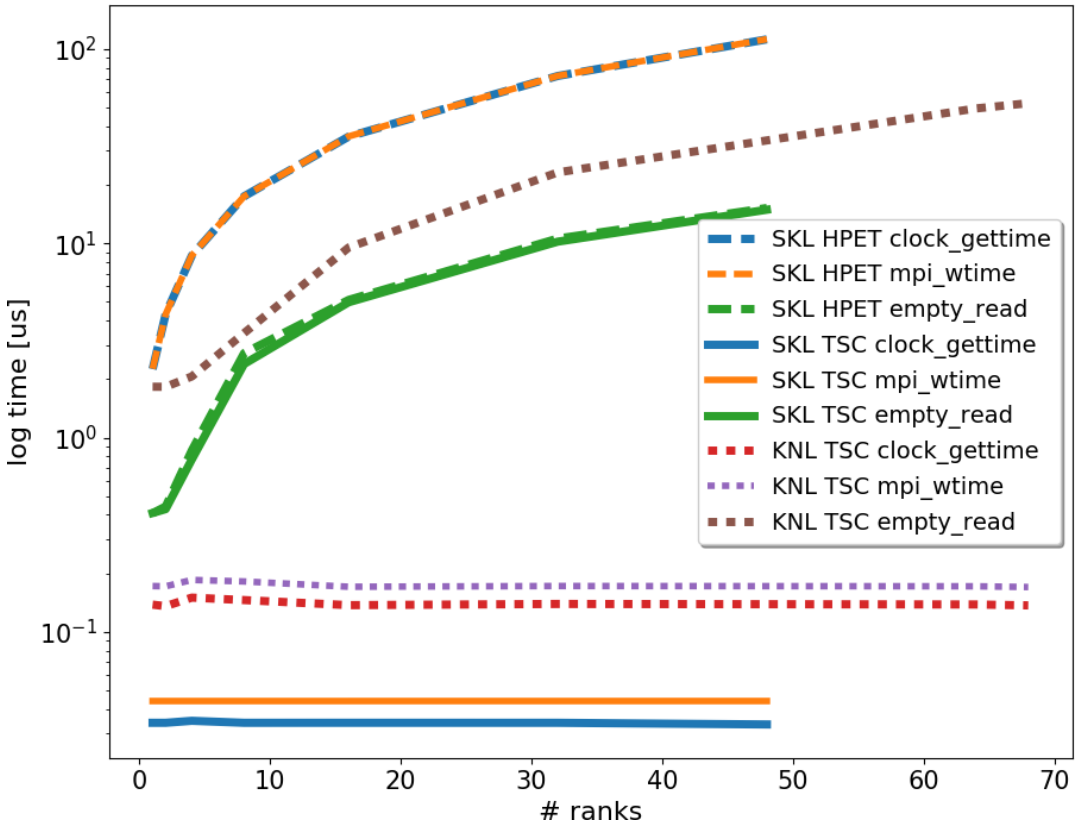


Fig. 64 Scaling behavior of three different functions for different timing facilities. The dashed lines refer to HPET, while the solid line refers to the TSC timer. The measurements where performed on a single SKL node each. The results of a KNL node are given for reference as well. They are shown as dotted lines.

The most obvious conclusion from these results is the fact that the HPET timer does not scale at all. In stark contrast to that are the results for the clock_gettime and MPI_Wtime in the case of the TSC timer. Their scaling is absolutely perfect. They also perform much better than their respective empty_read measurement which confirms the correct application of virtual kernel calls. In the case of the HPET timer, even the empty_read performs better than the two timer read-outs. This means, that the kernel call virtualization is not able to remedy the atrocious performance of the HPET timing facility. It is also interesting to see that the empty_read does not scale. This suggests that the kernel space switch is a collective operation over all processes on a node.

A comparison between the different clock_gettime and MPI_Wtime measurements shows that the overhead, introduced by the MPI library, is negligible.

The KNL node has the same systematic behavior but generally performs one order of magnitude worse than the SKL node.

7.2.3. **Summary**

We can very decisively draw the conclusion that, in the case of an HPC application, the TSC timing facility performs much better than the HPET timing facility. It should be the first choice in any HPC setup. We could not identify a single redeeming quality of the HPET facility and did not receive any indication of its usefulness even after reaching out to other colleagues. Its latency as well as its scaling behavior are both very bad and severely restrict the usability of time measurements on an HPC system.

7.3. ***Accurately measuring time differences***

Many tasks require accurate measurements of run times. From measuring algorithmic efficiency, to accurately evaluating bugs in implementations, to monitoring hardware performance, time measurements are the basis for proper evaluations.

Accurately measuring the run-time of different functions in a cluster environment is not a simple task. In order to solve this problem an independent library of small functions has been implemented. It contains around 600 lines of code and is specialized to accurately measure time differences and bandwidth.

```
If (rank == 0) {
      t1 = take_time();
      module->send(1, &t1, 8);
      module->recv(1, &t2, 8);
      t3 = take_time();
} else {
      module->recv(0, &t1, 8);
      t2 = take_time();
      module->send(0, &t2, 8);
}
```

Listing 3 Copy of Listing 4 in (Hoefler & Lumsdaine, 2008). Clock-difference is calculated as $\boldsymbol{t_{\text{diff}} = |t_1 - t_2|}$.

Special care has to be taken when designing the start timer and end timer functionality. POSIX offers the clock_gettime family of functions. These offer nanosecond resolution on most systems but are still fairly high-level and therefore easy to use. For more accurate measurements, there are several CPU instructions like RDTSC which return the number of CPU cycles from the start of the system. In light of automatic adaption of the CPU frequency to the current processor load it is easy to see that these may be harder to use. On modern processors the CPU instructions are often corrected for this, but it may be necessary to include special code for certain models. Furthermore in most multi-core systems, each socket has its own clock chip which makes synchronizations necessary. Most systems perform synchronization when booting, but the clocks may still drift during runtime. This could make resynchronization necessary. A good overview of these problems can be found in (Hoefler & Lumsdaine, 2008). After careful consideration and testing, we chose to use the clock_gettime family of functions.

Another very important part and difficulty of accurately measuring time differences in multi-process binaries are pipeline effects and process skew. You can find very nice illustrations of these problems in Fig. 1 and Fig. 2 of (Hoefler & Lumsdaine, 2008). In order to counteract these two effects comparable time measurements for the different ranks are needed. This is commonly not possible because of the above mentioned reasons, i.e. the different clocks on different ranks.

Essentially, since the clocks of the different ranks can not be trusted to be properly synchronized, it would be necessary to calculate the time offsets (and their time dependence) in order to get rid of process skew. Since the velocity of messages and the distance between ranks is unknown, the only way of measuring this offset is through synchronization. Unfortunately, synchronization, via a call to MPI_Barrier, will not suffice, as the different processes will not exit the function at the same time. The time resolution of an MPI_Barrier is too coarse as well (an MPI_Barrier call is two orders of magnitude slower than an timer read-out).

Listing 3 contains the code which is used to calculate the clock difference in the above mentioned reference. We identified two drawbacks to this approach:

1. When rank 1 is delayed with regards to rank 0 the measured time difference will be wrong.
2. The two measured values t_1 and t_2, used in the computation of t_{diff}, are not actually at the same point in time. There is a systematic difference added to the value, in the form of the time it takes for one message to be delivered.

Therefore a different scheme was implemented. You can find an outline of this scheme in Fig. 65. Since no time measurements are performed before a call to a communication function, process skew can not disrupt it. Due to the averaging

process, the returned clock difference is actually the real clock difference without any systematic offset.

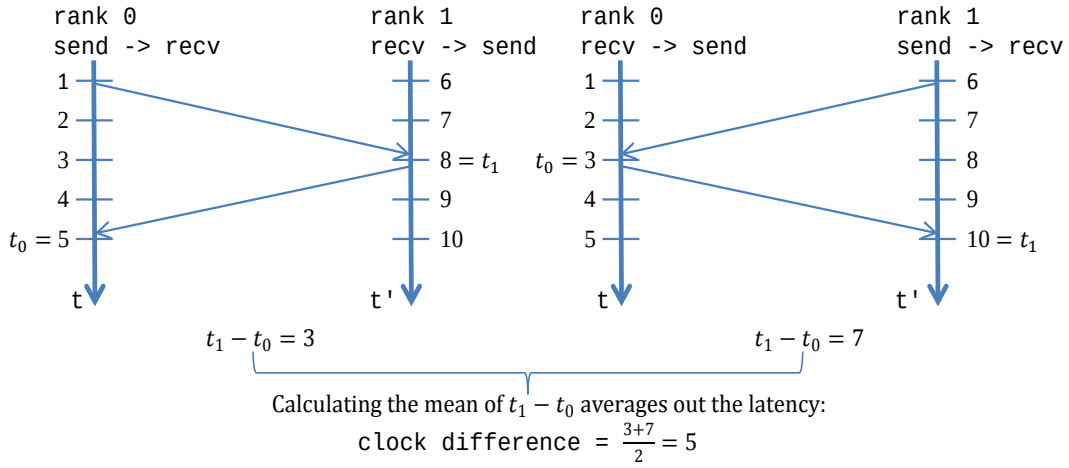


Fig. 65 Time measurements happen at the marked positions. The downward arrows signify the time as measured on the respective rank. It is easy to see that the clock difference in this case is 5. The diagonal arrows represent communication events. An arbitrary amount of $\boldsymbol{t_1 - t_0}$ values are computed using either the left or the right process. Which specific process is chosen is determined randomly. The average of all these values then equals the clock difference between rank 0 and rank 1.

The process outlined above is used to calculate the time offset to rank 0 for every rank. Each rank is therefore supplied with a value $t_{r,\text{diff}}$ which can be used to transform rank local times to global (rank 0) times and back via the two simple formulas $t_{\text{global}} = t_{\text{local}} - t_{r,\text{diff}}$ and $t_{\text{local}} = t_{\text{global}} + t_{r,\text{diff}}$.

In order to measure the time a certain function took on each rank, two values are measured: $t_{r,\text{start}}$ and $t_{r,\text{end}}$, where r is the rank number. Each rank now computes its $t_{r,\text{start,global}}$ value from its $t_{r,\text{start}}$ value using $t_{r,\text{diff}}$:

$$t_{r,\text{start,global}} = t_{r,\text{start}} - t_{r,\text{diff}}.$$

Combining the values of all ranks we get

$$t_{\text{start,global,max}} = \max_r t_{r,\text{start,global}},$$

which is the time of latest entry into the measured function. Up to this point in global time all ranks, except the slowest one, have been waiting. This value needs to be transformed to local time using

$$t_{r,\text{start,max}} = t_{\text{start,global,max}} + t_{r,\text{diff}}.$$

Finally, the time the examined function took on each rank is given by

$$t_r = t_{r,\text{end}} - t_{r,\text{start,max}}.$$

This process is then repeated for n iterations in order to increase the confidence in the result. The different results are denoted by $t_{r,i}$. The result of the first iteration is commonly discarded as there are a couple of startup effects which disrupt its validity.

The time measurement library will print several values to stdout or to a json file when prompted:

- $\min_{r,i} t_{r,i}$
- $\max_{r,i} t_{r,i}$
- $\text{mean}_{r,i} t_{r,i}$
- $\sum_{r,i} t_{r,i}$
- $\min_r \text{sum}_i t_{r,i}$
- $\max_r \text{sum}_i t_{r,i}$
- $\text{mean}_r \text{sum}_i t_{r,i}$

The values in the second column are useful to determine differences between ranks. They are collected under the name “ranksum” in the output.

These values are also used when bandwidth measurements are requested. In this case the number of bytes per measurement has to be supplied to the library. The bandwidth is then calculated in a straightforward manner. Special care has to be taken for the “ranksum” values. In their case it is very important to keep in mind which processes happen in serial and which processes happen in parallel. For serial processes the times which are used in the calculation have to be added, for parallel processes the amount of bytes has to be multiplied by the number of processes.

A nice example of the advantages of this approach to time measurements can be found in chapter 7.4.1. In order to further improve this library clock drift could be taken into account. This has not been done yet, because the timescales for clock drift are much higher than our current use cases (refer to Fig. 5 in (Hoefler & Lumsdaine, 2008)). Furthermore, the suggestions in 3.1 of (Hoefler & Lumsdaine, 2008) could be beneficial and may be added in the future.

7.4. *Performance implications of Hardware interrupts*

The first use case for the new time measurement library is an investigation into the implications of hardware interrupts on the performance of computing nodes. Hardware interrupts may be triggered by different parts of a computer architecture. They enable a very important functionality in computer systems: the ability to force a CPU to react to a certain request immediately. Real time functionality, like the movement of the mouse pointer, is controlled using interrupts. Interconnect controllers on HPC systems use interrupts for the message handling as well.

If interrupts are handled by only one fixed specific core of a multi-core system, the performance of collective operations could decrease as this specific core will lag behind the other cores because of its additional duties. Knowledge about this possible behavior could lead to updated configuration advice to our users. Specifically, it could be advantageous to leave the first (or any other specific) core free during an MPI or OpenMP computation in order to not have it affect the performance of collective operations. Since the number of cores on the KNL and the Skylake machine is not a power of 2, some jobs already do not use all available cores. In this case it could be very helpful to be able to choose the right core to leave out of a computation.

In order to investigate this, we measure the time an MPI_Barrier call takes for different configurations of cores left out of the MPI environment. We use a custom made program, using the library which was developed in the previous chapter. It makes 10^7 calls to the MPI_Barrier function and measures the time each core takes to complete it. It then computes the maximum time over all cores and uses it to identify the time each call took to finish globally. These 10^7 times are then binned and plotted in a histogram using a double logarithmic plot.

We always use a single node and run the test twice, once with a process pinning which utilizes the cores from 0 upwards and once with a process pinning specifying the cores from 1 upwards. In each case we use one core less than is available on the node. The results of these two measurements are plotted on top of each other in the following figures.

7.4.1. **Process skew elimination**

In order to gauge the effect of the process skew elimination algorithm, detailed in the previous chapter, simulations where made with and without it. The results can be found in Fig. 66.

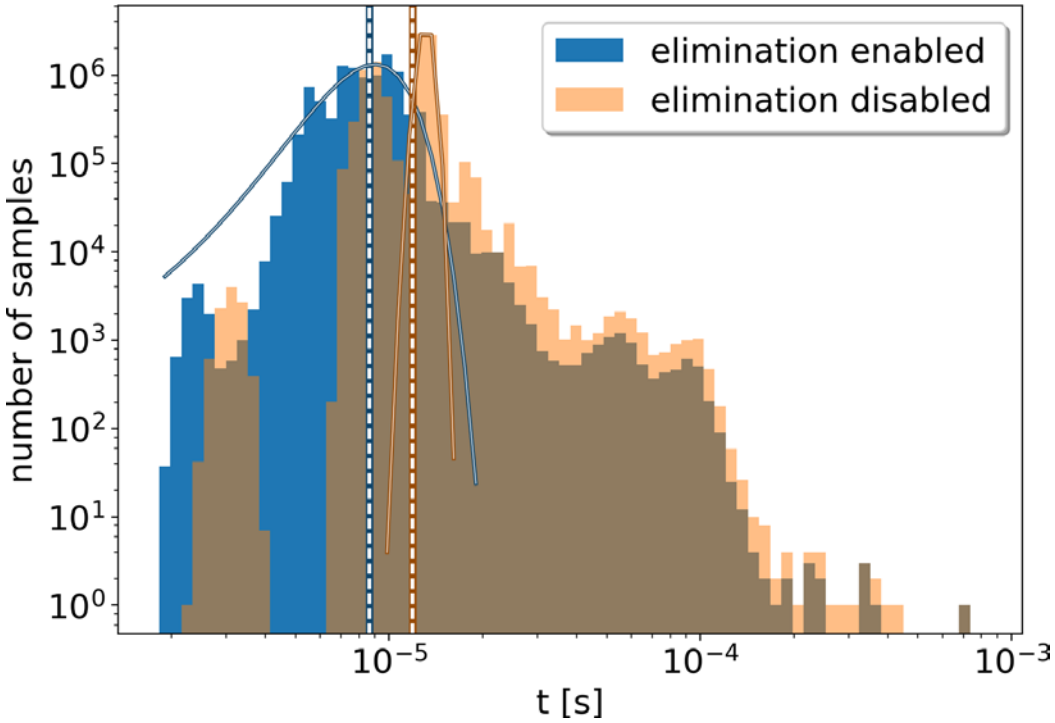


Fig. 66 Time per MPI_Barrier call (global maximum) on one KNL cache node for 10^7 functions calls. 68 threads are used (all cores, no hyperthreading). The mean value of all samples is shown by the vertical dashed lines. The same samples are used for both results, only the post-processing differs for the two cases. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines.

It is important to note that the process skew elimination algorithm will always make individual samples quicker, that is to say, their execution time shorter. This is guaranteed by assertions in the running binary which make sure that $t_{r,\text{start,max}} \geq t_{r,\text{start}}$ always holds.

Therefore, the process skew elimination algorithm results in a general shift of all samples to the left. It makes the execution time of all samples either smaller or leaves them at their initial time. This coincides with an expected left shift of the mean.

It can be assumed that all effects which are responsible for differences in the completion time of an MPI_Barrier should, in theory, be of a statistical nature. This would necessitate that the slopes of the measured peaks follow a Gaussian function. For this reason a least-squares fit to a Gaussian function is plotted in all graphs. Any values lying outside of this fitting function must be the result of additional statistical or systematic processes. They can not be explained by a single normally distributed random process.

The first peak of the corrected version loosely fits the Gaussian structure. The slower events do not follow the fitting function, which means that they probably are the result of different systematic mechanisms, which are slowing down the barrier.

For the uncorrected version the fit is much worse. This is due to a split of the main peak into two peaks and an additional very distinct first peak. These peaks make the least squares method converge to one of the two main peaks, which hints at the fact that the whole data series does not have a Gaussian shape. These fundamental changes in the shape of the results show a systematic error introduced by the process skew. It seems to favor certain completion times over others.

These systematic errors are resolved using the process skew elimination algorithm. All subsequent results therefore employ it.

7.4.2. **KNL partition**

The results for the KNL cache partition can be found in Fig. 67 and Fig. 68, and the results for the KNL flat partition can be found in Fig. 69 and Fig. 70.

The additional plots (Fig. 68 and Fig. 70) provide a different view on the same data. For this different version, the results are binned and the amount of samples in each bin is multiplied by the bin value (taken at the center of the bin). This product is then used for the heights of the bars. These plots are helpful to determine the importance of the slower function calls in terms of the used wall time. The additional plots also show the summed time of all samples (T in the legend) and the summed time of all samples under the fitting function (F in the legend). This makes it easy to recognize the amount of total wall time which can be attributed to a single normally distributed random process.

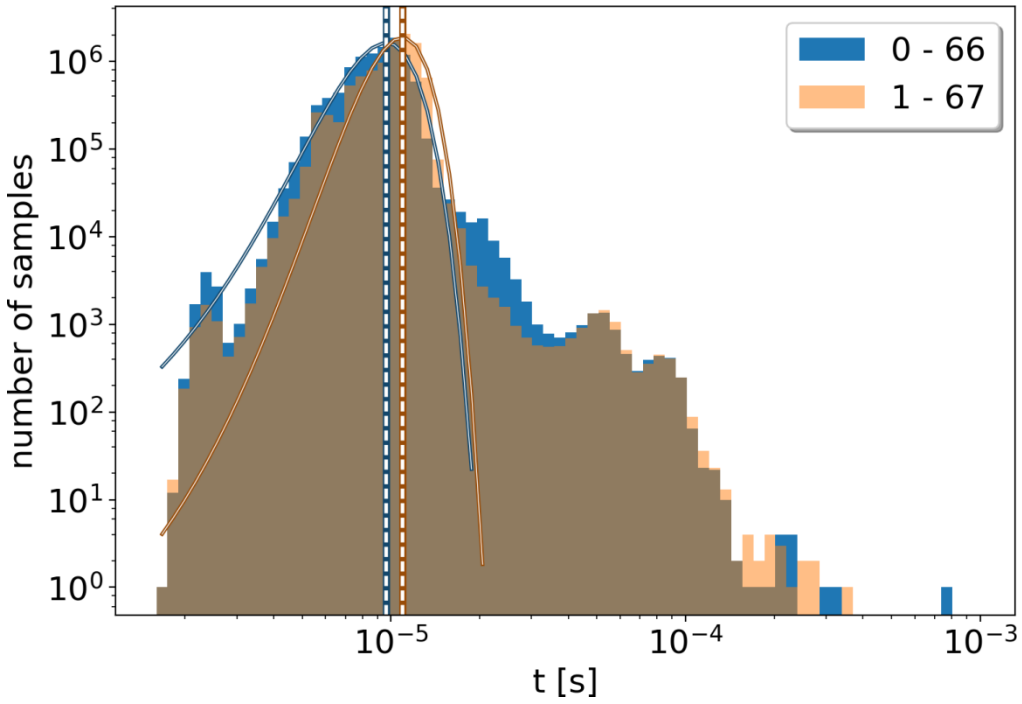


Fig. 67 Time per MPI_Barrier call (global maximum) on one KNL cache node for 10^7 function calls. 67 out of the 68 cores are used. Core numbering starts at 0. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines.

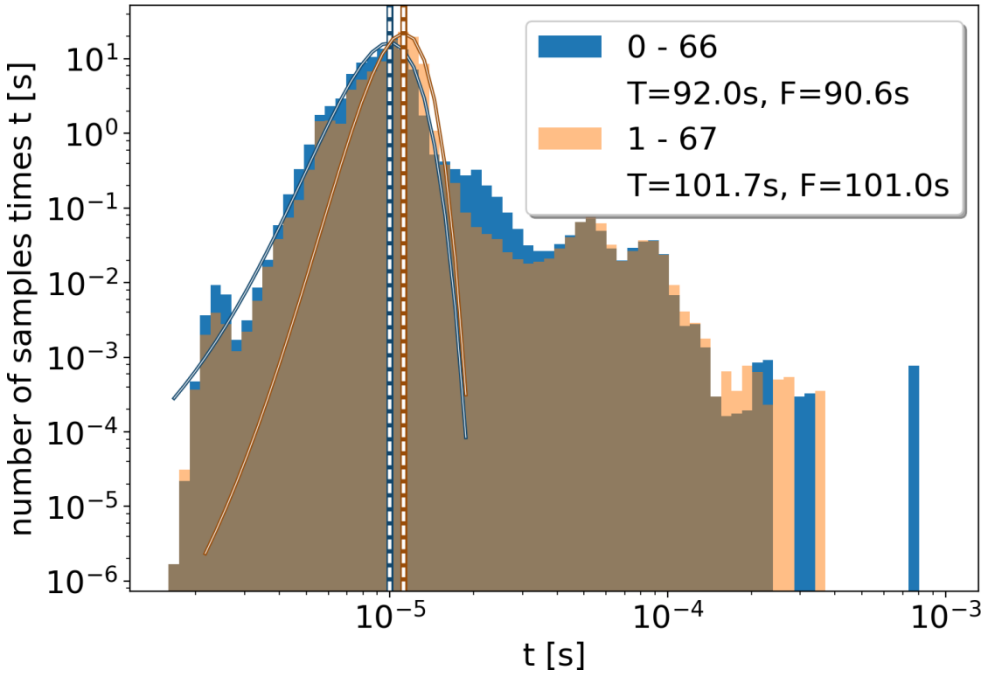


Fig. 68 Time per MPI_Barrier call (global maximum) on one KNL cache node for 10^7 function calls. 67 out of the 68 cores are used. Core numbering starts at 0. Bar height equals number of samples times t. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines. T gives the summed time of all samples, while F is restricted to samples under the fit function.

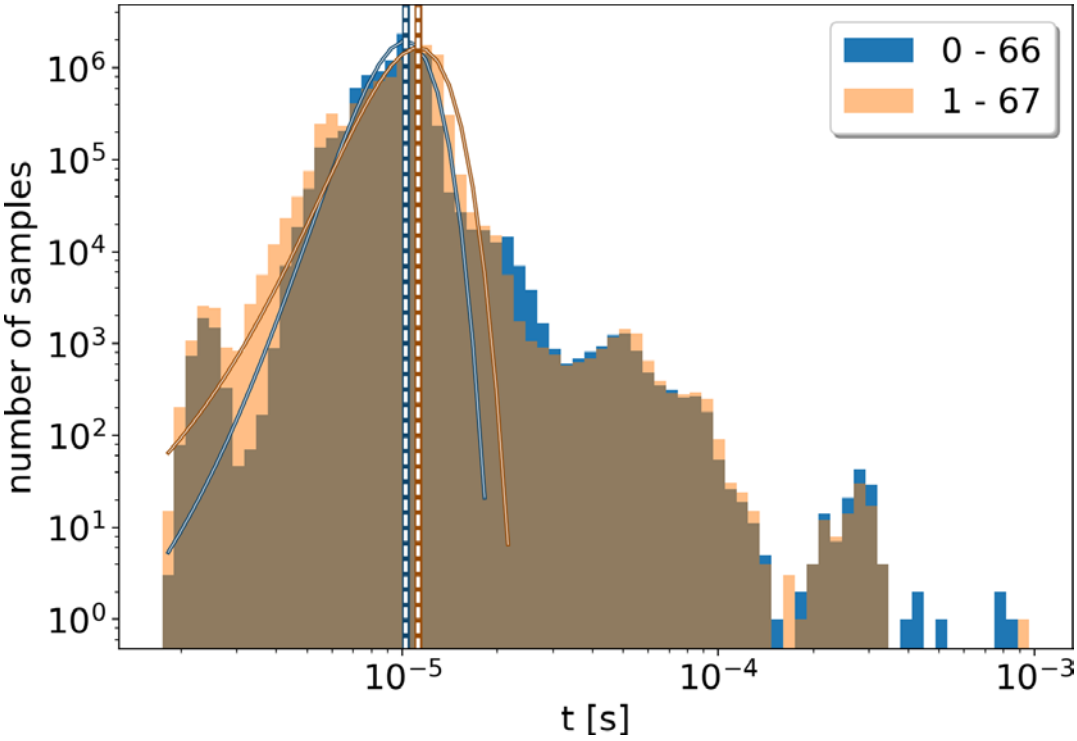


Fig. 69 Time per MPI_Barrier call (global maximum) on one KNL flat node for 10^7 function calls. 67 out of the 68 cores are used. Core numbering starts at 0. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines.

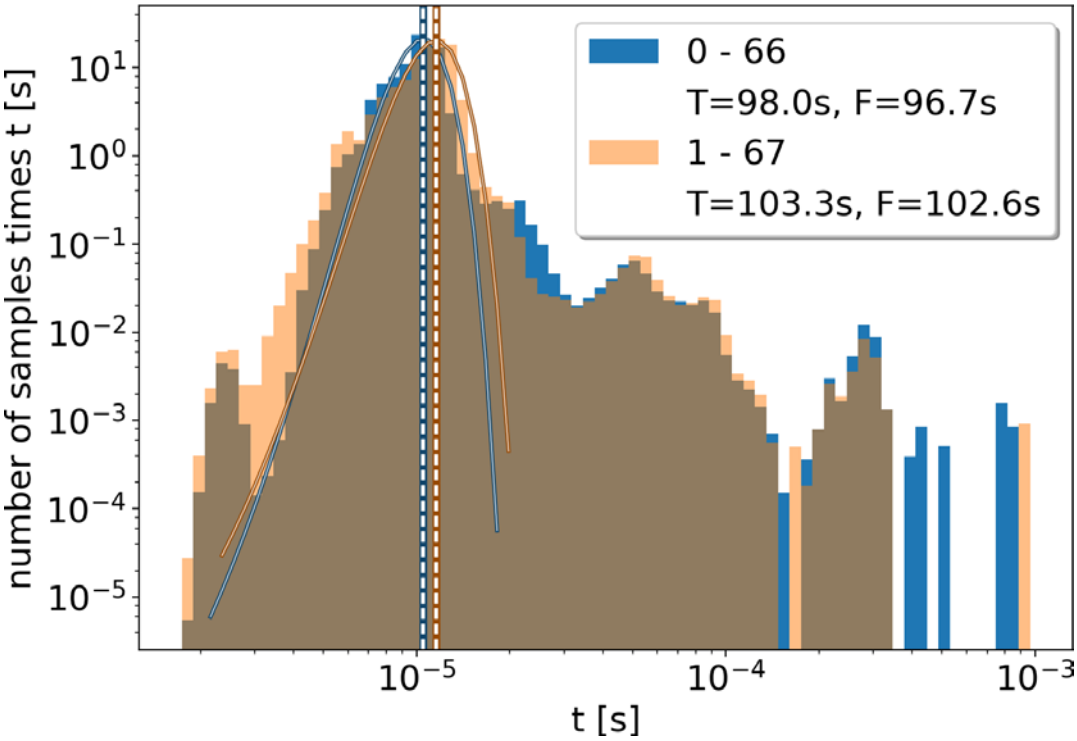


Fig. 70 Time per MPI_Barrier call (global maximum) on one KNL flat node for 10^7 function calls. 67 out of the 68 cores are used. Core numbering starts at 0. Bar height equals number of samples times t. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines. T gives the summed time of all samples, while F is restricted to samples under the fit function.

Comparing the different pinning cases, we can not identify any systematic difference between the two. For KNL cache as well as for KNL flat, it seems slightly advantageous to leave the last core free, as the mean time to completion is slightly faster. The difference is very miniscule though. No usage suggestion can be derived from this data. Since there is no systematic difference, the OS processes are probably allowed to float on a KNL system. This floating is explained in further detail in the summary.

The comparison to the Gaussian fitting function shows a lot of samples which are much to slow to be part of a randomly distributed process. When considering their effect on the wall time (Fig. 68 and Fig. 70), it is clear that they are not imposing a huge run time disadvantage on the system. They only take about one percent of the total wall time (given by $1 - F/T$).

The differences between the two memory modes are very indistinct as well. One visible difference lies in the data left of the mean, where, in the case of the KNL flat partition the dip at $3 \cdot 10^{-6}$s is considerably deeper for the blue configuration in comparison with the results of the KNL cache partition. This difference is too small to be reflected in the fitting function. KNL flat seems to be slightly more sensitive to an occupied first core. The other difference is an additional, very small peak at $3 \cdot 10^{-4}$s. No additional conclusions can be derived.

7.4.3. **SKL partition**

You can find the results for the Intel Skylake (SKL) partition (A3) using a socket agnostic configuration in Fig. 71 and Fig. 72.

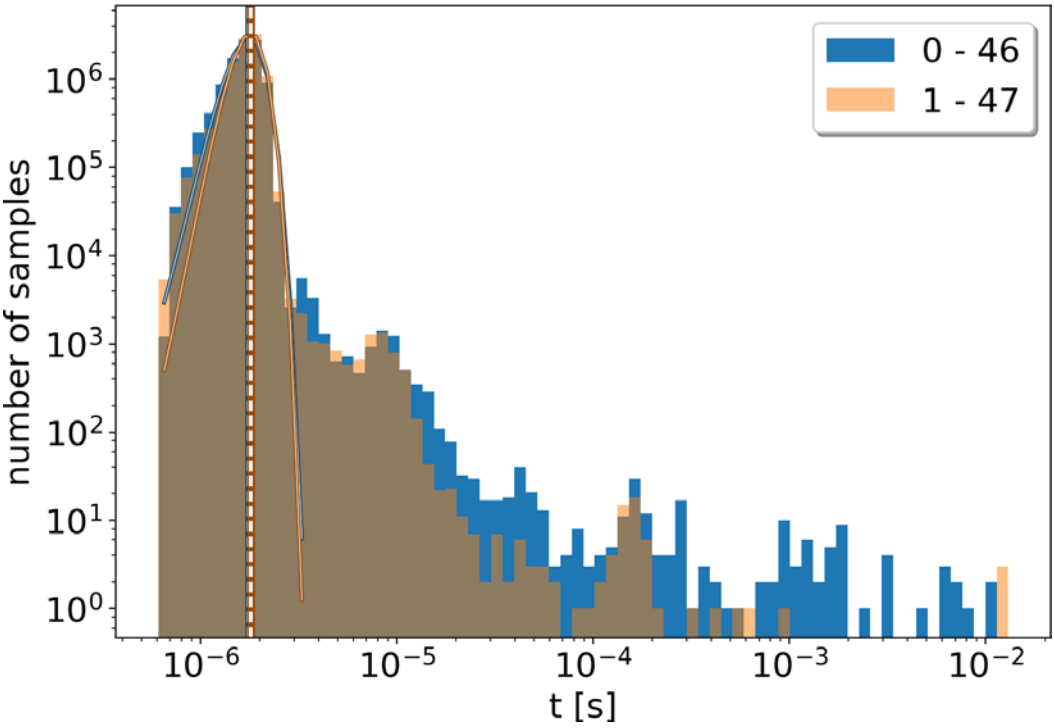


Fig. 71 Time per MPI_Barrier call (global maximum) on one SKL node for 10^7 function calls. 47 out of the 48 cores are used. Core numbering starts at 0. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines.

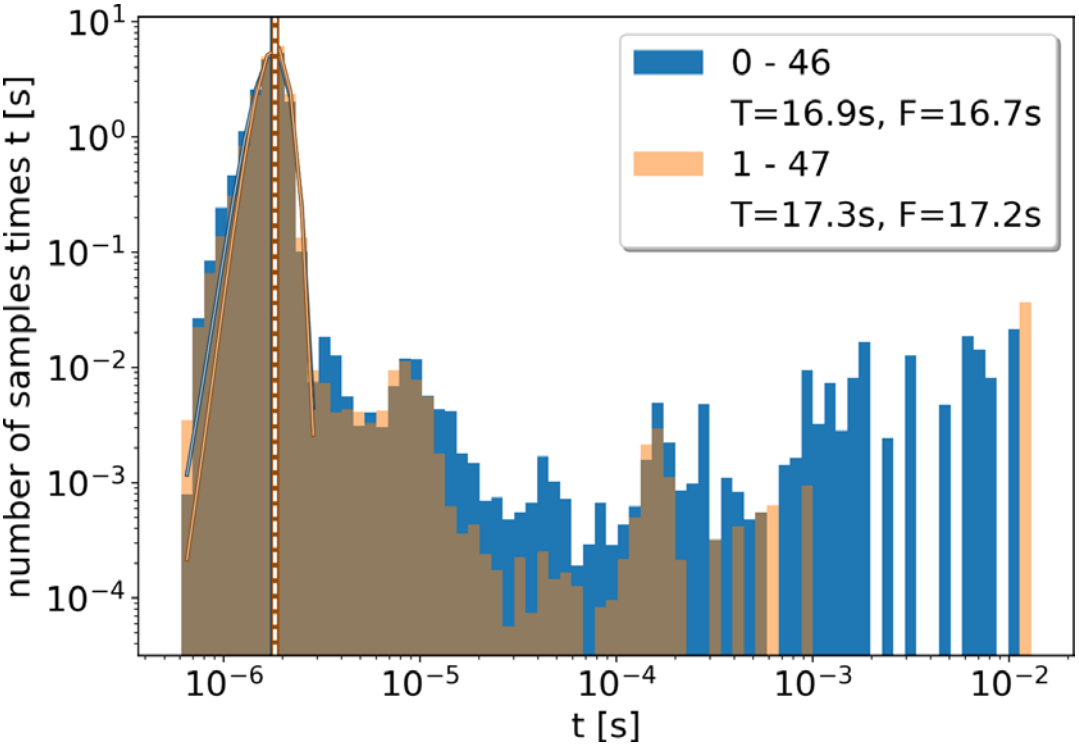


Fig. 72 Time per MPI_Barrier call (global maximum) on one SKL node for 10^7 function calls. 47 out of the 48 cores are used. Core numbering starts at 0. Bar height equals number of samples times t. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines. T gives the summed time of all samples, while F is restricted to samples under the fit function.

The first major difference to the KNL results is the fact that the mean run time for a barrier call is about one order of magnitude faster on SKL. The first peak is also much more closely shaped like the Gaussian fitting function. This means that the bulk of the samples are not encumbered by any operating system noise. This is made very obvious when examining T and F. Almost all of the wall time is spend in samples contributing to the first peak. The much faster mean run time may partially be explained by the fewer numbers of cores participating on SKL. Since a proper barrier implementation will employ some kind of tree based algorithm though, it can not satisfyingly explain the difference.

The other big difference is the occurrence of a small amount of very slow events. These events only amount to about several hundreds out of the total amount of 10^7, but they are very slow. These barriers take up to tens of microseconds to complete. They still only contribute about one percent (blue case) and less than half a percent (orange case) to the total wall time due to their limited amount.

Leaving either the first or the last core free, does not change the result in a meaningful way. Again it seems to be slightly advantageous to leave the last core free, but the differences are not meaningful enough to give usage recommendations.

Following a discussion about these results, Jacques David brought up the idea of changing the measurements in a way to include information about the detailed topology of the underlying architecture. For Skylake this means taking into account the fact that the system is organized in sockets which contain multiple CPUs and then leaving one core per socket free. This change leads to the results shown in Fig. 73 and Fig. 74.

Interestingly, a lot of the very slow events vanish in this case. This suggests that each socket needs a dedicated core to perform hardware interrupts, but it is again not important which core is left idle. The difference between the topology aware and the previous example is only 1 s, so it is not exceptionally worthwhile to instruct users to change their configurations.

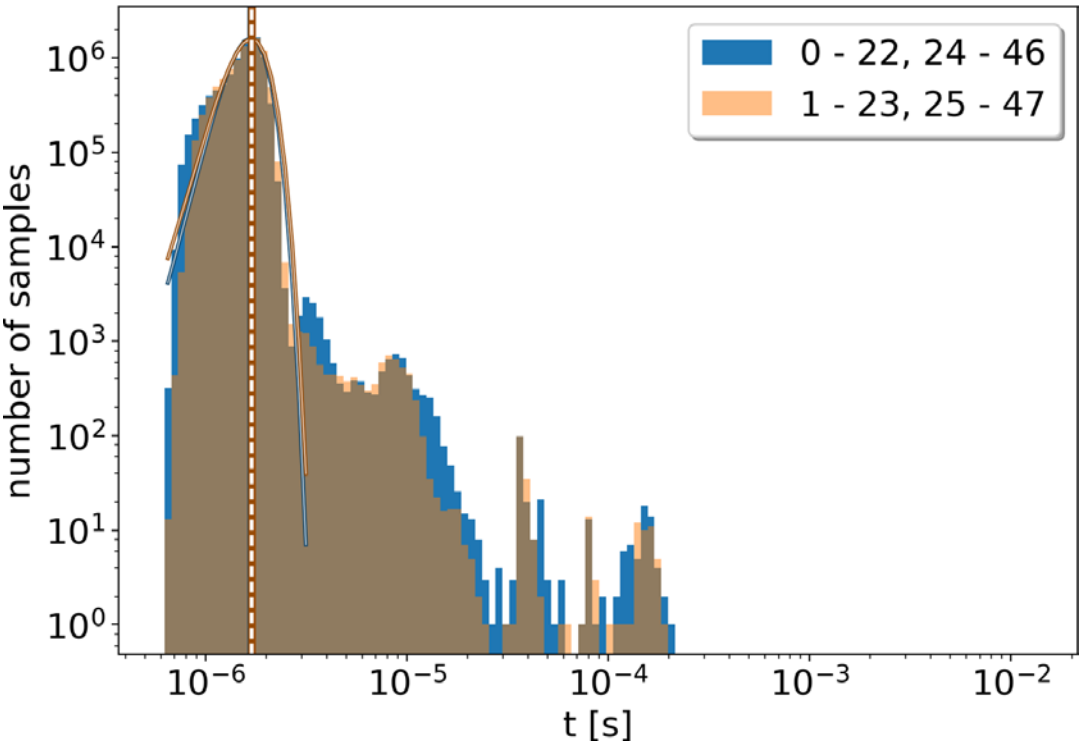


Fig. 73 Time per MPI_Barrier call (global maximum) on one SKL node for 10^7 function calls. 46 out of the 48 cores are used. Core numbering starts at 0. One core per socket is left idling. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines.

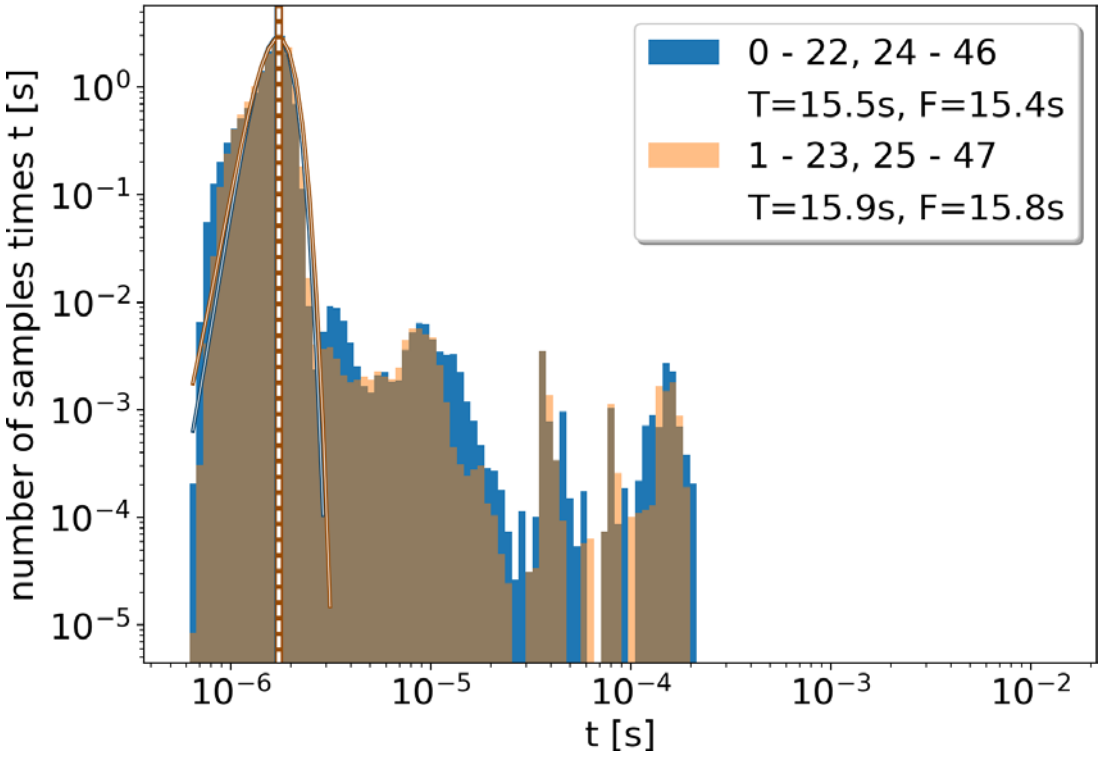


Fig. 74 Time per MPI_Barrier call (global maximum) on one SKL node for 10^7 function calls. 46 out of the 48 cores are used. Core numbering starts at 0. One core per socket is left idling. Bar height equals number of samples times t. The mean value of all samples is shown by the vertical dashed lines. A Gaussian fit, made using the least-squares algorithm, is shown in the blue-white and orange-white lines. T gives the summed time of all samples, while F is restricted to samples under the fit function.

7.4.4. **Summary**

The measurements show that the system intelligently uses free cores to perform hardware interrupt handling. It seems to need one core per socket to be able to do this effectively. The rescheduling of the interrupts of the interconnect fabric is the topic of the HFI_NO_CPUAFFINITY flag of the Performance Scaled Messaging (PSM2) messaging API. This API is part of the Intel Omni-Path Host Fabric interface used by the Marconi Cluster. The flag avoids the pinning of the PSM2 process to specific CPUs and seems to be enabled on all Marconi partitions.

In conclusion, very clear signs of hardware interrupt effects could be found. No measurements can be explained by a single normally distributed random effect. Fortunately the effects are always very small and do not interfere with the execution of the barriers in a destructive manner.

In all cases it is slightly advantageous to leave the last core on each socket free. But the difference is so small that it will not make a difference for actual simulations. The avoidance of very slow events on SKL could make it worthwhile in special cases.

7.5. *I/O performance measurements*

7.5.1. **Introduction**

Input/Output (I/O) operations are notoriously hard in a high-performance computing (HPC) environment. This can be illustrated by the following quote:

„A supercomputer is a device for turning compute-bound problems into I/O-bound problems." - Ken Batcher

The reasons for the complicated nature of I/O in HPC are mainly given by the following two points:

- It represents loads and stores on the highest latency device in the system. This makes it very important to have non-blocking operations and a lot of intermediate buffering. It is also helpful to create few very large transfers instead of many small transfers.
- Data consistency is hard to guarantee. Since access to data should be universal over different machines and for different users, data consistency checks can severely limit the effectiveness of the operations. In many cases, it can be helpful to create multiple files. This makes data consistency easier. It also makes writing multi-process programs simpler, as each rank can write its own output to disc. Unfortunately, too many files may break certain HPC file systems, and many files are very inconvenient to evaluate. A more practical solution is user guided consistency.

Because of this it is very important to thoroughly test the performance and stability of the available resources.

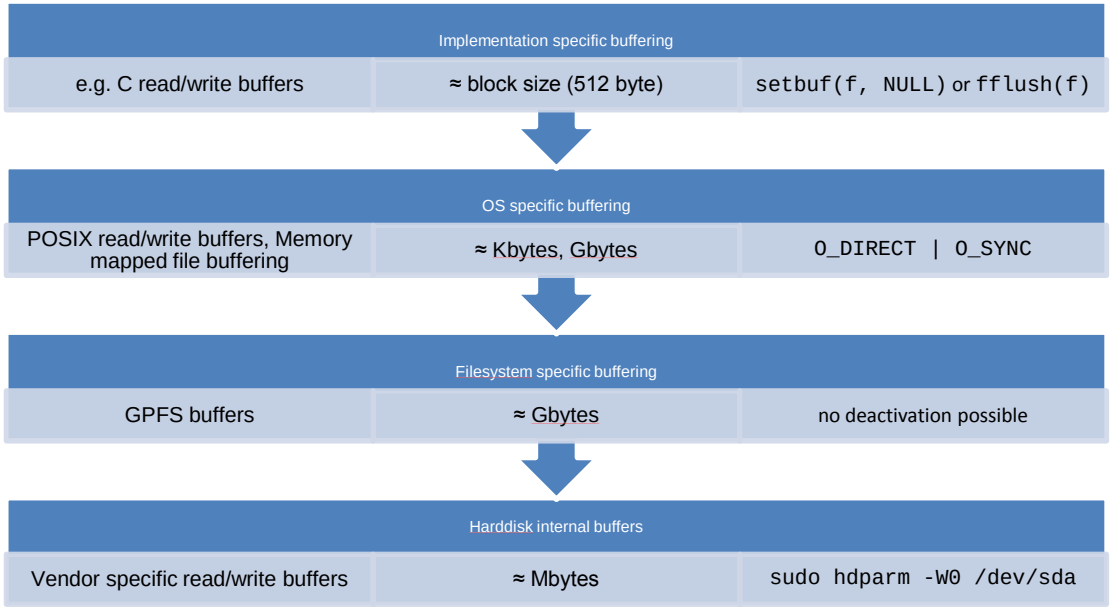


Fig. 75 The main problem when designing I/O benchmarks is the complicated caching hierarchy. This figure shows the four most common caching structures, examples of implementations for each, their approximate size and possible configuration parameters to deactivate them.

One of the main problems in the design of effective I/O benchmarks is the complicated caching hierarchy. This is shown in Fig. 75. It is important to make sure that all, or at least most, caches are circumvented when performing the measurements. The possible circumvention methods are given in the last column of Fig. 75.

7.5.2. **I/O benchmark tool**

In order to perform I/O benchmarking a tool was implemented. The current version is completely written in C and consists of about 2000 lines of code. Using a pure MPI parallelization it contains code to test POSIX I/O as well as MPI I/O. The main result is the achieved bandwidth for reading and writing blocks of data of different sizes. All parameters are easily customizable via the command line. This is very convenient when performing large parameter scans.

The resulting output data is written to files in the json format which makes it easy to evaluate the results in an efficient way.

The start/end timer function from section 7.3 is used where applicable. The tool currently offers a host of command line options which enable a broad spectrum of configurations.

7.5.3. **Bandwidth using multiple output files**

In the following we provide results of bandwidth measurements for the Broadwell and KNL partition on the Marconi supercomputer. Reading performance seems to be consistently higher than writing performance. This is propably related to remaining caches and not to actual differences in the performance of the hardware. The first batch of graphs are concerned with the Broadwell partition, using one file for each rank. Fig. 76 shows reading and writing performance of the Broadwell partition. This graph can be very helpful in anticipating and evaluating the I/O performance of a specific application. We conclude that a minimum of about 1000 ranks is needed to achieve the maximum I/O performance. According to Fig. 77 it is not sufficient to only have one rank on each node writing data. Please note that this measurement was made with a very large block size of 128 MB. A plot of the block size dependency can be seen in Fig. 78. It shows that the maximum writing bandwidth can only be expected with block sizes of about 75 MB and higher.

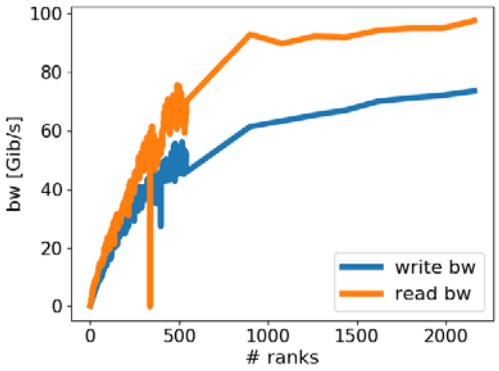


Fig. 76 I/O benchmark results on Broadwell using one file per rank.

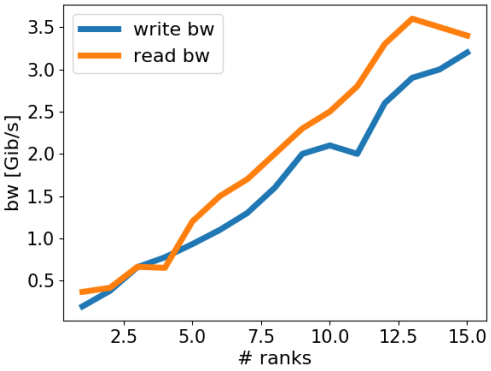


Fig. 77 I/O benchmark results on Broadwell using one file per rank and one rank per node.

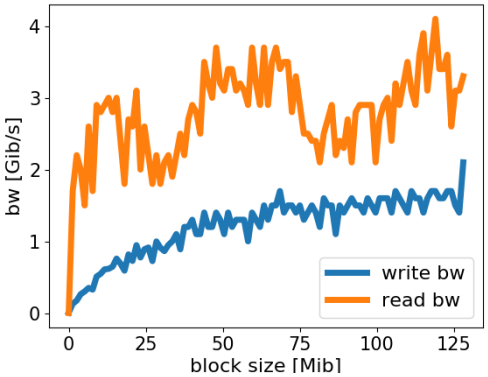


Fig. 78 I/O benchmark results on Broadwell using one file per rank and 10 nodes to perform a block size scan.

Only one result for the KNL partition is shown. In Fig. 79 it can be verified that the performance of the I/O system does not critically depend on the used compute partition, since it is a common resource shared by both partitions. This is the expected result. It is important to note that the number of ranks per node is very different between Broadwell and KNL. For this reason, less KNL nodes are needed in order to reach peak I/O performance.

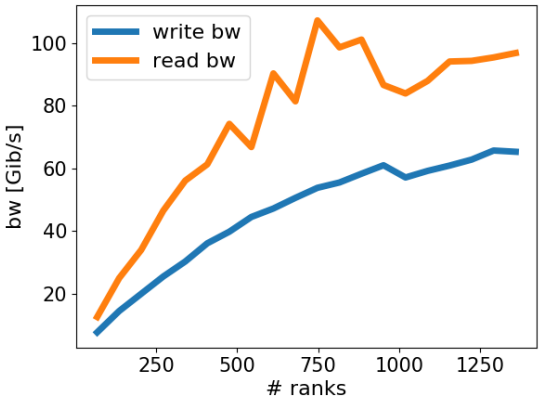


Fig. 79 I/O benchmark results on the KNL partition in cache mode using one file per rank.

7.5.4. **Bandwidth using a single file**

Optimal I/O performance can be expected when writing into one distinct file per rank. In this case the I/O is not burdened with the synchronization of the resulting data. Unfortunately this increases the user effort considerably since data collection needs to be performed after the simulation finishes. Often considerably fewer resources are available at this point. On top of that, many file systems exhibit massive performance degradation if too many files are present in a directory, necessitating complicated directory structures. Therefore, data collection to less than the total number of ranks during simulation time is often considered.

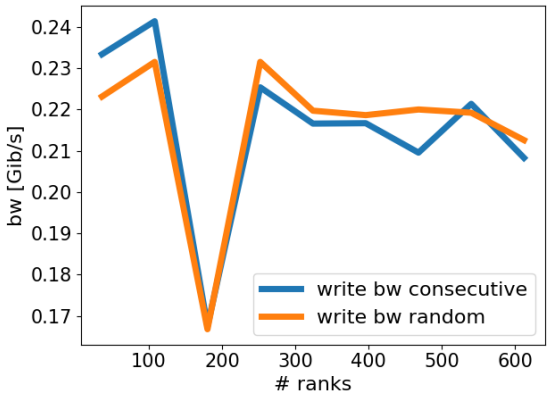


Fig. 80 I/O benchmark results on the Broadwell partition using one file for all ranks using POSIX output. The POSIX backend serializes the operations even for consecutive output.

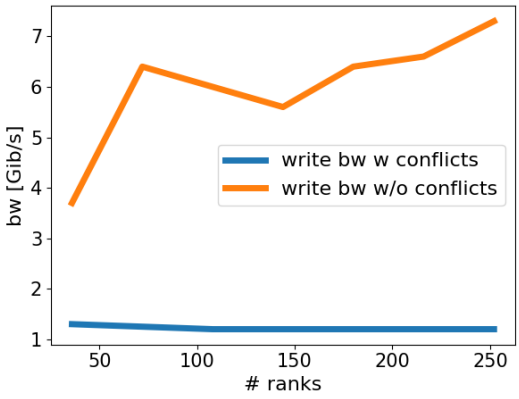


Fig. 81 I/O benchmark results on the Broadwell partition using one file for all ranks using MPI I/O. W/O conflict refers to an implementation which ensures no write conflicts. This user guided consistency provides good performance.

This can be done by implementing collective MPI routines. The collected data can then be written into fewer files or even a single file. Unfortunately, this approach can have significant drawbacks:

1. As can be easily seen when comparing Fig. 76 and Fig. 77, it is imperative to use as many writing ranks as possible in order to achieve optimal performance.
2. Using multiple ranks to write into one file (avoiding MPI communication) can lead to serialization by the file system. This is shown in Fig. 80 and Fig. 81. If the underlying software stack detects potential conflicts, it will serialize the output very aggressively. Fig. 80 shows that even when there are no conflicts in the write calls (signified by the consecutive curve) the output performance does not scale. Fig. 81 shows, that the MPI API will actively detect conflicts and will switch to serial output if necessary.

Fortunately, these problems can be circumvented by the proper usage of the MPI I/O API. The orange, no conflicts, line in Fig. 81 shows very nice performance, writing into a single file from multiple ranks.

Using MPI I/O is generally the best approach, as it defers the research and knowledge about the best way to combine and write the data to a maintained library which hopefully delivers very good performance.

7.6. *Providing support for KNL*

Scientists inside the EUROfusion consortium may apply for computational resources on the Marconi cluster. Many of these projects have acquired considerable time on the new Knights Landing (KNL) partition. Data about the used project quotas for this partition have shown some reluctance to employ the system. In order to help these projects to achieve their scientific goal, the HLST reached out to their principal investigators (PIs). The eight largest projects with the least amount of spend resources in relation to the project budget where contacted: FUA21_AUGJOR, FUA21_EST3D, FUA21_COCHLEA, FUA21_FWTOR17, FUA21_BEFUSION, FUA21_BIGDFT4F, FUA21_DIVGAS-K and FUA21_FELTOR.

Since some of these projects are already part of a different HLST project, the offer was specified to only pertain to making the code run efficiently on the KNL system. Optimizations within the code were not advertised. The following three points were therefore focused on:

1. Optimize compilation:
   Recompiling the code for KNL mandates some specific compiler flags in order to enable proper usage of all capabilities. It was made sure that this simple step is performed correctly.
2. Proper pinning:
   Since the KNL nodes offer hyper-threading (HT) it is very important to correctly pin processes to the CPUs. Hybrid codes in particular need to make sure to have a correct configuration.
3. Memory configuration:
   Since KNL offers two different kinds of memory, users need to be advised on their proper usage. Different configurations can be tested quickly and may increase performance considerably.

In the following we will report on projects which reported back with a request for help.

7.6.1. **FU21_AUGJOR specific activity**

The principal investigator, Matthias Hoelzl, was contacted with the support offer. The JOREK project contains a lot of dependencies and therefore maintains a sophisticated build structure. In order to investigate this structure, which is essential to the usage of the code and therefore to providing support, access to the source code has been granted. With the help of the PI communication with multiple JOREK users has been established. It is expected that individual meetings will take place in the near future.

7.6.2. **FUA21_EST3D specific activity**

A meeting with the PI of the GRILLIX code, Andreas Stegmeir, has taken place. In the communication previous to the meeting, the PI reported that the code has already been tested on the KNL machine, but the obtained performance has been very disappointing.

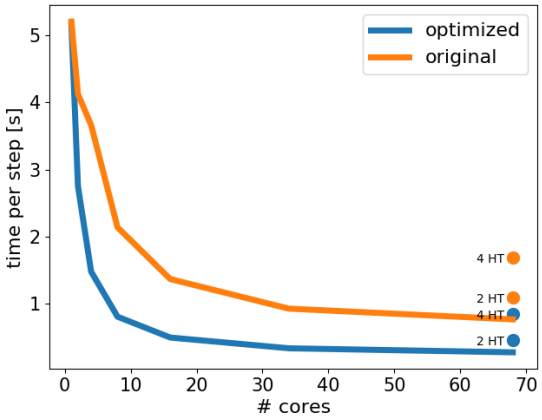


Fig. 82 Time per step in seconds for the GRILLIX code before and after the optimization effort. The strong scaling is not linear in any of the two cases, but the optimized version shows great improvements nonetheless. Hyperthreading is not beneficial to the performance of the code.

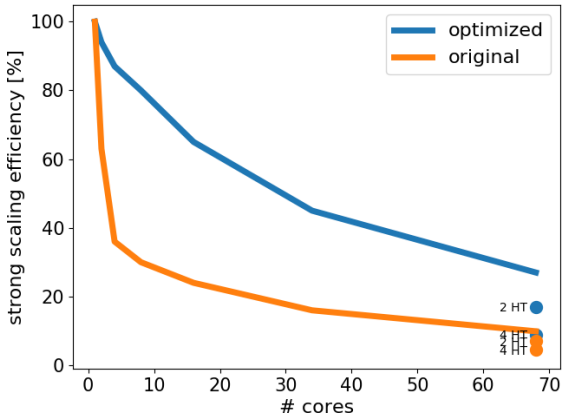


Fig. 83 The same results as in Fig. 82 expressed in terms of efficiency. Strong scaling efficiency is defined as $t_1/(N\,t_N)$ where t_1 is the time one rank needs to finish a step, N is the amount of ranks used and t_N is the time N ranks need to finish a step.

The meeting was very productive and several important configuration parameters were discussed. Simulations using these quick optimizations show a very nice improvement, as can be seen in Fig. 82 and Fig. 83. The code is still not scaling properly. Changes to the scaling are not expected from these simple configuration modifications.

7.6.3. **FUA21_COCHLEA specific activity**

The PI, Ioannis Tigelis, referred us to Dimitrios Peponis as the main user of the code on the KNL partition. Dimitris Peponis subsequently reported on its efforts of using COCHLEA on KNL and the performance problems he encountered. Following his reports, detailed instructions for his specific problems were prepared and submitted to him. The main points in these instructions were pertaining to the compilation (usage of the Intel compilers is highly recommended) and pinning (COCHLEA is a hybrid code which necessitates a very specific configuration). The results have yet to be reported back to the HLST.

7.6.4. **FUA21_BIGDFT4F specific activity**

Comprehensive assessment of the current situation of the project revealed, that it suffers from severe performance degradation on KNL. This is very surprising, as this code has very good scaling properties. By investigating the compilation options some problems where found and fixed. The pinning proved to be complex. The code was supposed to run on 64 of the available 68 cores, using 8 MPI processes with 16 threads each. This corresponds to using 2 hyperthreads per core. The original configuration looked like this:

```
#PBS select=50:ncpus=68:mpiprocs=8:mem=108GB:mcdram=flat:numa=snc2
export OMP_NUM_THREADS=16
```

This gives (disregarding the obsolete numa configuration specification) the following MPI pinning on KNL:

- Rank 0
  - 1st HT on CPU 0, ..., 8 (9 threads)
  - 2nd HT on CPU 0, ..., 8 (9 threads)
  - 3rd HT on CPU 0, ..., 7 (8 threads)
  - 4th HT on CPU 0, ..., 7 (8 threads)
- Rank 1
  - 1st HT on CPU 9, ..., 16 (8 threads)
  - 2nd HT on CPU 9, ..., 16 (8 threads)

    - 3rd HT on CPU 8, ..., 16 (9 threads)
    - 4th HT on CPU 8, ..., 16 (9 threads)
- …

Here, HT is short for “hyperthread”. This is obviously not the desired behavior. The different ranks share CPUs which is disastrous for the performance. When looking at the OpenMP thread pinning resulting from this MPI pinning, the CPU sharing is reflected there as well.

The solution we presented to the PI, Stephan Mohr, is a new configuration:

```
export KMP_AFFINITY=balanced,granularity=fine
export OMP_NUM_THREADS=16
export I_MPI_PIN=1
export I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=64-67,132-135,200-203,268-271
```

Due to a bug in Intel MPI this will only work for Intel MPI 2018. This configuration makes sure to exclude the last 4 cores and all their hyperthreads. The chosen KMP_AFFINITY=balanced results in an OpenMP pinning which uses consecutive hyperthreads for all processes (compact would use 4 hyperthreads on all necessary cores and scatter would lead to consecutive threads not residing on the same core). We also suggested different affinities to test in order to find the best performing one. It was checked that all proposed configurations give the correct pinning behavior.

As a last measure we made sure, that the MKL library, used by the project’s code, is properly configured for KNL in hybrid mode as well:

```
export MKL DYNAMIC="FALSE"
export MKL_ENABLE_INSTRUCTIONS=AVX512_MIC
```

Unfortunately all these measures did not ameliorate the situation and the performance on KNL remained lackluster. In light of this we urged the PI to make an official project proposal to the HLST which should comprise the optimization of the source code used in the project.

7.6.5. **Summary**

You can find a summary of this effort in Fig. 84.

Fig. 84 Summary of the compilation, pinning and memory configuration support provided for the KNL partition.

7.7. *Reliability monitoring of the Marconi resources*

The performance of the Marconi Fusion machines are regularly tested using the well-known Linpack (Dongarra, Bunch, Moler, & Stewart) and Stream (McCalpin) benchmark. We perform these tests on all available partitions, the Broadwell, the KNL and the SKL partition. The benchmark tests all nodes on the KNL partition (for both memory models), samples 229 nodes out of 806 for the Broadwell partition and samples 229 out of 1512 nodes for the Skylake partition. These tests have been initiated during the CINCOMP project and already lead to some very important performance optimizations. The continuous monitoring using these standardized tests ensures reliability for the users of the system. This report presents a temporal series of benchmarks which helps visualize the development of the hardware. All nodes accessible to the EUROfusion consortium are tested.

All following graphs consist of two figures, the top one shows the results of the Linpack benchmark (FLOPs in GFLOP/s), and the bottom one shows the results of

the Stream benchmark (bandwidth in MB/s). The *x*-axis of each graph denotes the respective node which produced the result. The names of the nodes are only printed if the measured value is exceptional. Values are deemed exceptional if they are bigger or smaller than two standard deviations from the mean of the whole data (over all nodes). The black line denotes one standard deviation and the light blue line denotes two standard deviations from the mean. One can find the job id at the top of each graph.

Additionally, a further partition monitoring system has been implemented. It tracks the usage of all nodes on each partition over time and classifies them into down, idle and busy nodes. The results are printed in two graphs. One graph has a high temporal resolution of 10 minutes, while the second graph sums over a whole day. White regions in the plot are times where either i) no monitoring was conducted or ii) the login node was down.

7.7.1. **Broadwell partition**

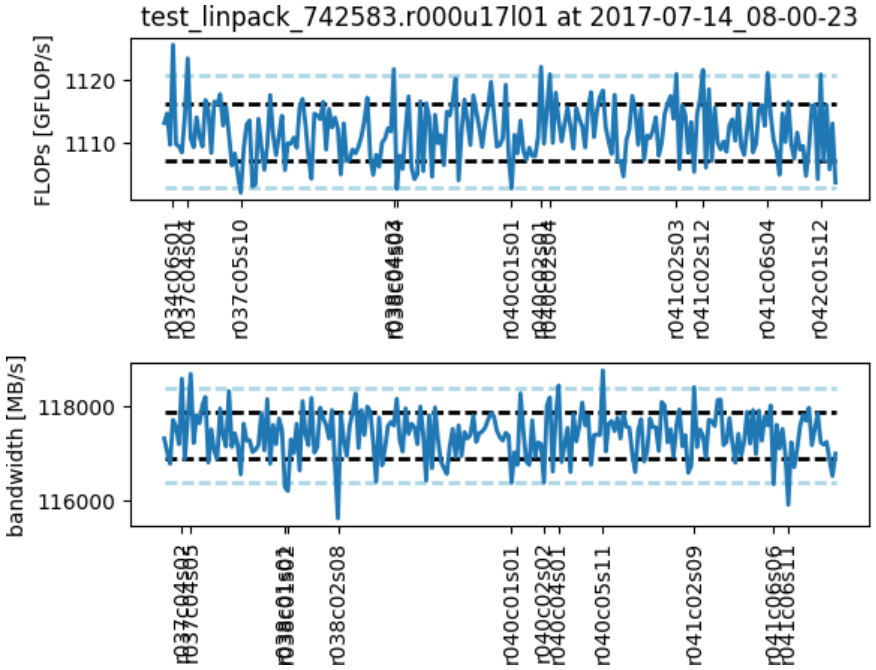


Fig. 85 June 2017 results of the Broadwell monitoring.

The results of the Broadwell partition for June 2017 can be seen in Fig. 85. This partition has been very stable for a long time, and these results agree with that assessment. We do not measure any ailing nodes, i.e. nodes with a vastly lower performance than all other nodes. The last ailing nodes were found in the measurement performed on April 13th.

Fig. 86 shows the results for October 2017. It can be seen, that the partition is very stable. This result was measured two weeks before the planned switch-off time of the system.

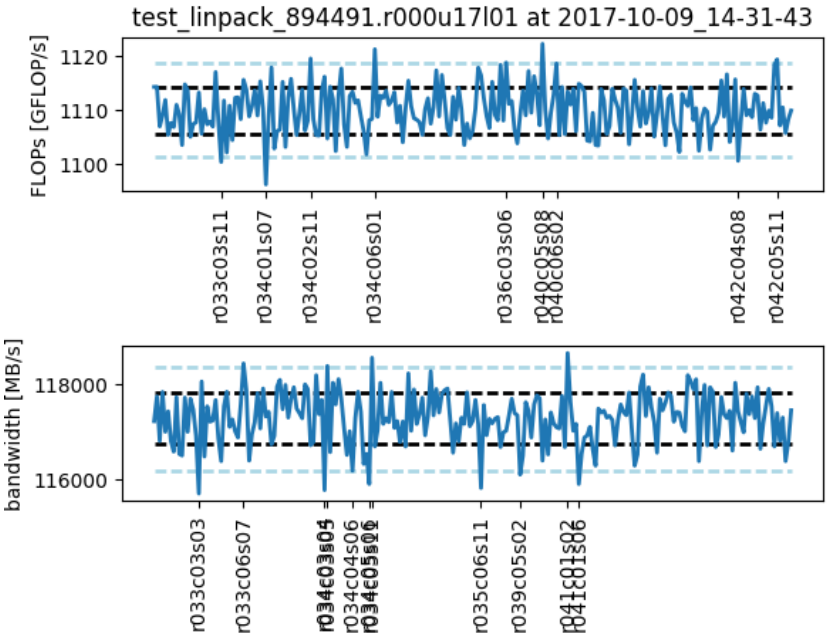


Fig. 86 October 2017 results of the Broadwell monitoring.

The node usage graph for the Broadwell partition (Fig. 87) shows a very high utilization of the system until its shutdown.

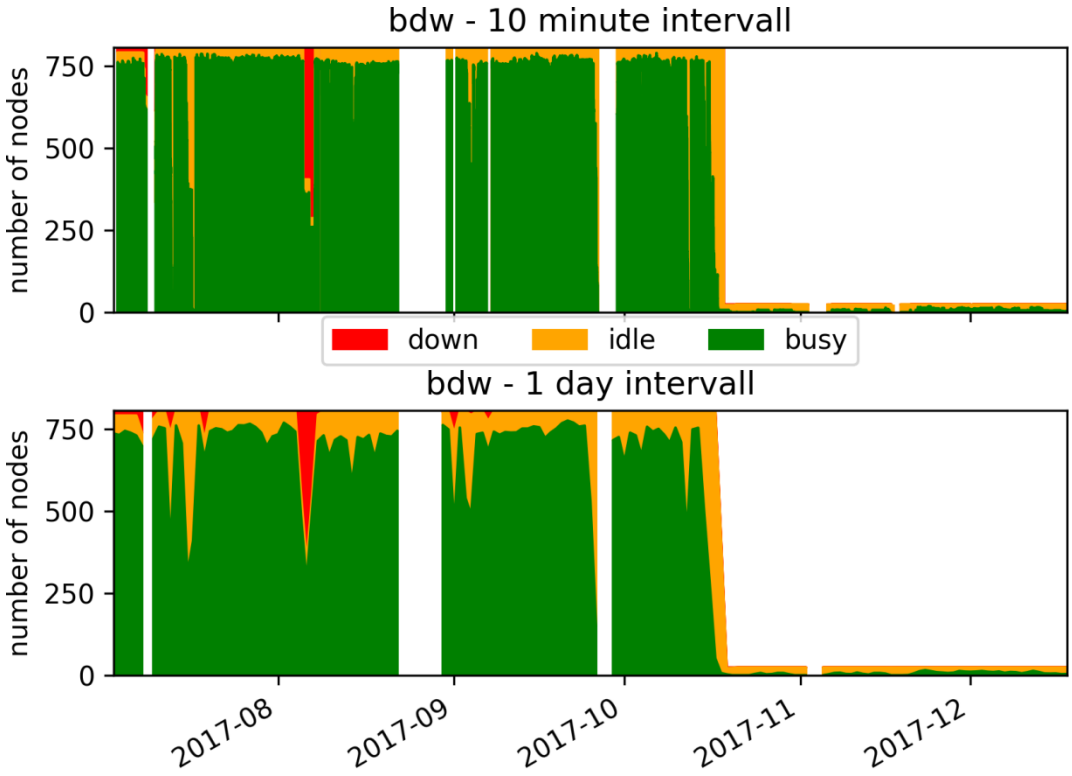


Fig. 87 Node usage for the Broadwell partition over time.

7.7.2. **KNL partition**

The results for July 2017 of the second Marconi partition, the KNL partition, are shown in Fig. 88 and Fig. 89. The first figure refers to the KNL partition which is configured to use the flat memory mode; the second figure refers to the KNL partition which is configured to use the cache memory mode.

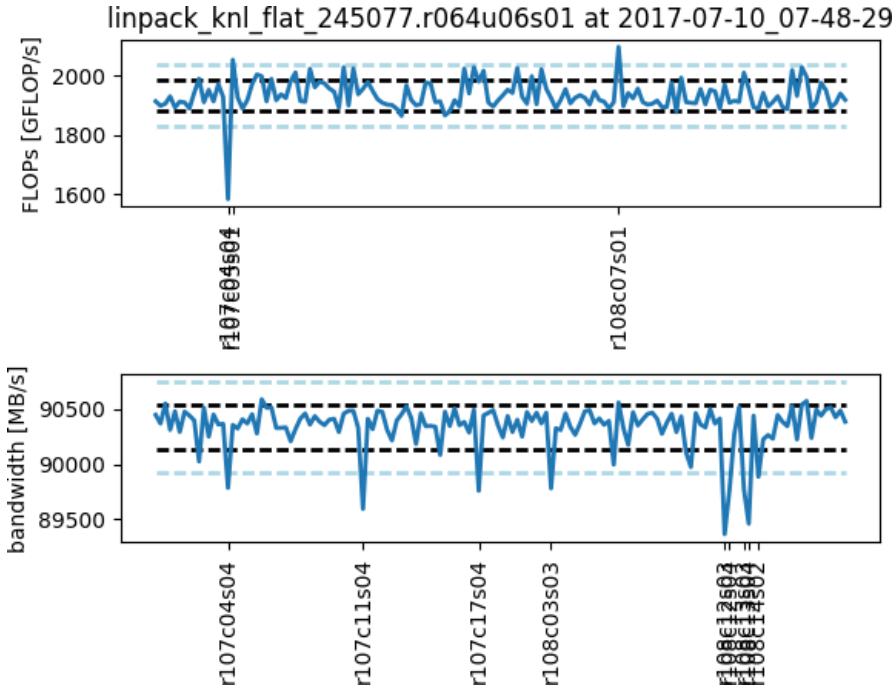


Fig. 88 July 2017 results of the KNL flat mode monitoring.

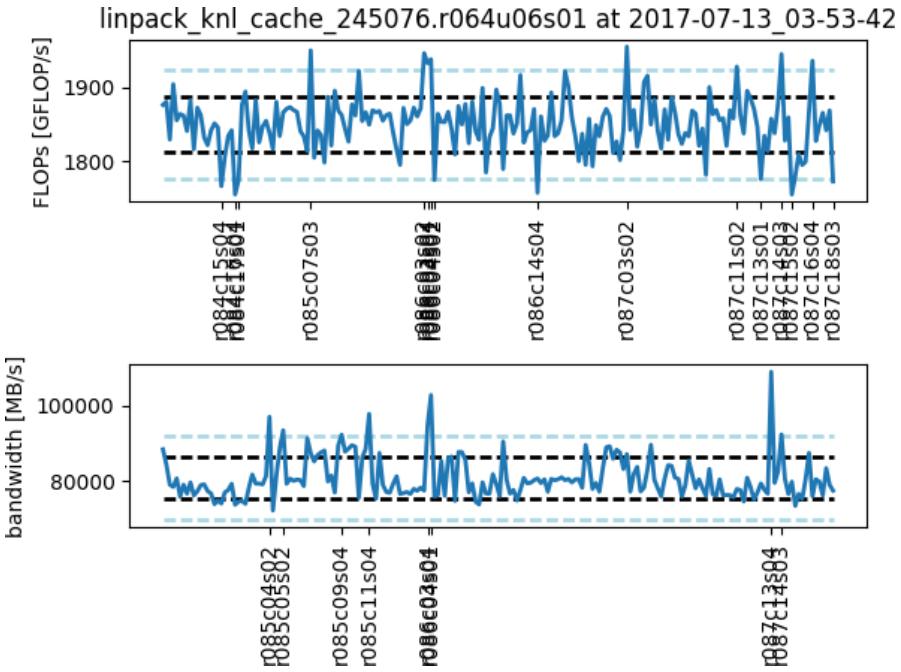


Fig. 89 July 2017 results of the KNL cache mode monitoring.

The KNL partition is much less stable in comparison to the Broadwell partition. It still exhibits ailing nodes from time to time. They do occur less often in the last months, though.

The July 2017 result for the KNL flat partition shows one ailing node, with about 16% worse performance than the others. All the other nodes are comparable in performance. Information about ailing nodes is communicated to the Marconi support team. The measurements still show a problem for the bandwidth. Several nodes have a slightly worse bandwidth than the others. This difference is not very large. It is only about 1000 MB/s less bandwidth when the mean bandwidth is about 90500 MB/s. The very suspicious look only results from the fact that the bandwidth in general is very constant for the different nodes. It is also very stable for different days of measurement. All things considered, we do not deem this very slight drop in

bandwidth a big problem. Nonetheless, it would be nice to find the source of this misbehavior.

The next batch of results concerns the KNL partition configured to use the cache memory mode. The resulting data can be found in Fig. 89. We can not identify any ailing nodes. As was already shown in previous results, the bandwidth of the cache mode nodes is not stable over time. This is reflected very well in our measurements. It fluctuated between 120000 MB/s (measurement on the 6th of April 2017) and 75000 MB/s (measurement on the 14th of April 2017), with groups of nodes having very different values. The bandwidth seems to settle at around 81000 MB/s in the following weeks and it looks almost equal for all nodes in the measurement made on the 13th of July and shown in Fig. 89. This could be related to the "cache cleaning" procedure which was implemented by the Marconi support. We will continue to monitor this development.

To summarize, the results can be considered good. The cache mode KNL nodes still show some fluctuations, but they seem to diminish over time. For the other cases (cases without very large and obvious fluctuations) it would now be a priority to reduce the standard deviation in the Linpack and Stream performances, as many codes will only run as fast as the slowest participating node. For Broadwell the standard deviation is already quite small (about 5 GFLOP/s with a mean of 1110 GFLOP/s <=> 0.5% fluctuation) but for the KNL machines it is considerably larger (about 50 GFLOP/s with a mean of 1900 GFLOP/s <=> 2.6% fluctuation). For SPMD (single program multiple data) codes this means a significant loss in the number of performed computations.

The next results are the October 2017 results for both the KNL cache and the KNL flat mode. You can find these results in Fig. 90 and Fig. 91.

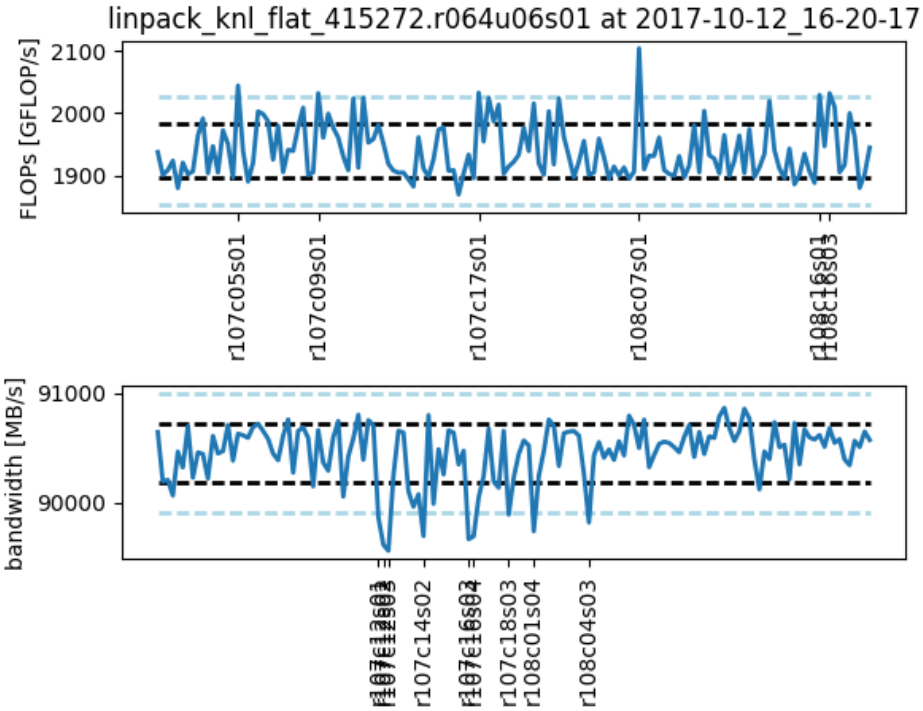


Fig. 90 October 2017 results of the KNL flat mode monitoring.

KNL flat mode monitoring (Fig. 90) show a lot of improvement compared to the previous data. As you can see, there are no ailing nodes. The computational performance as well as the bandwidth is very stable.

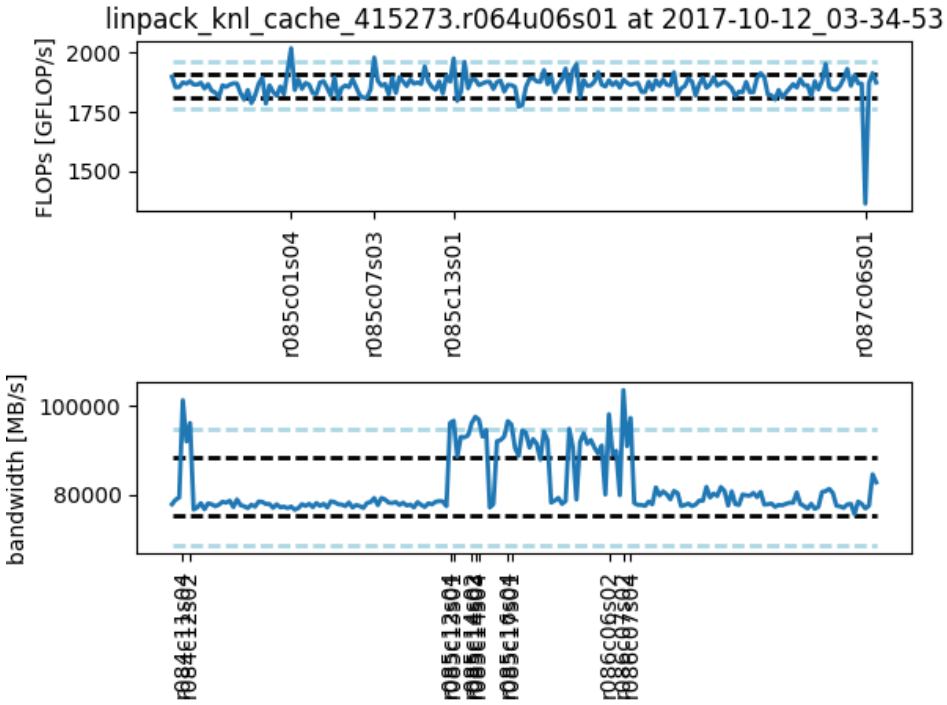


Fig. 91 October 2017 results of the KNL cache mode monitoring.

The performance of the larger part of the KNL partition, the part in cache mode, shows one ailing node (Fig. 91) which confirms the need for continuous monitoring. The bandwidth is very stable which means that the cache cleaning procedure is working.

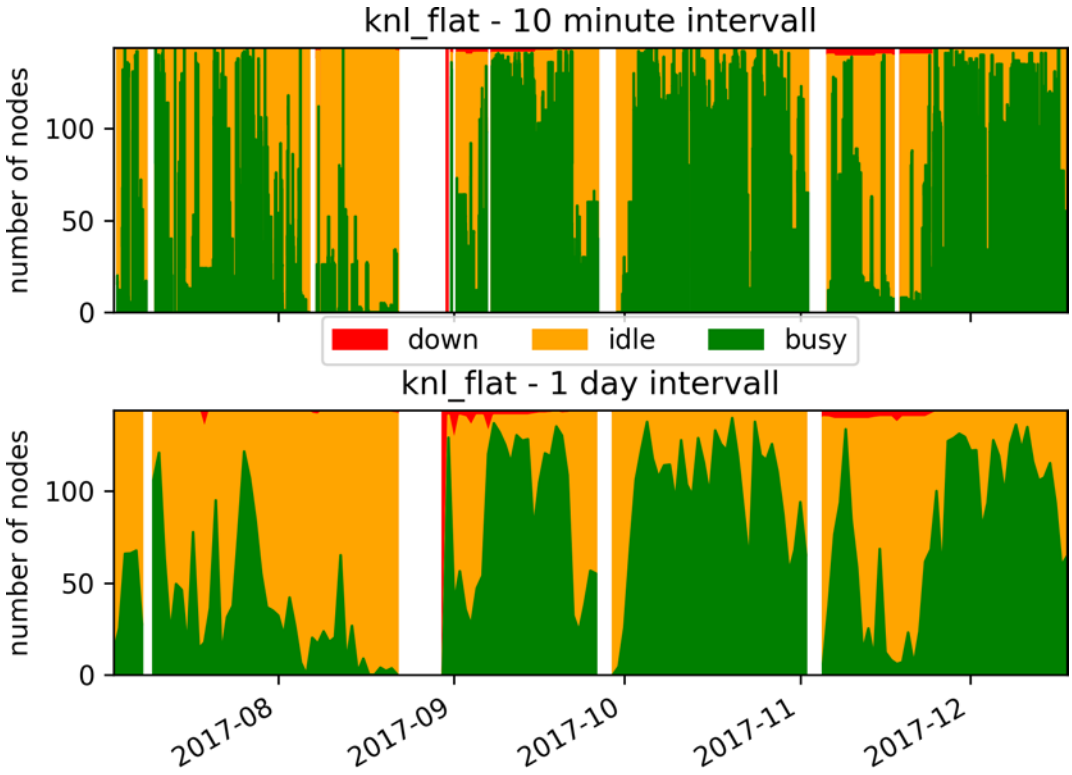


Fig. 92 Node usage for the KNL flat partition over time.

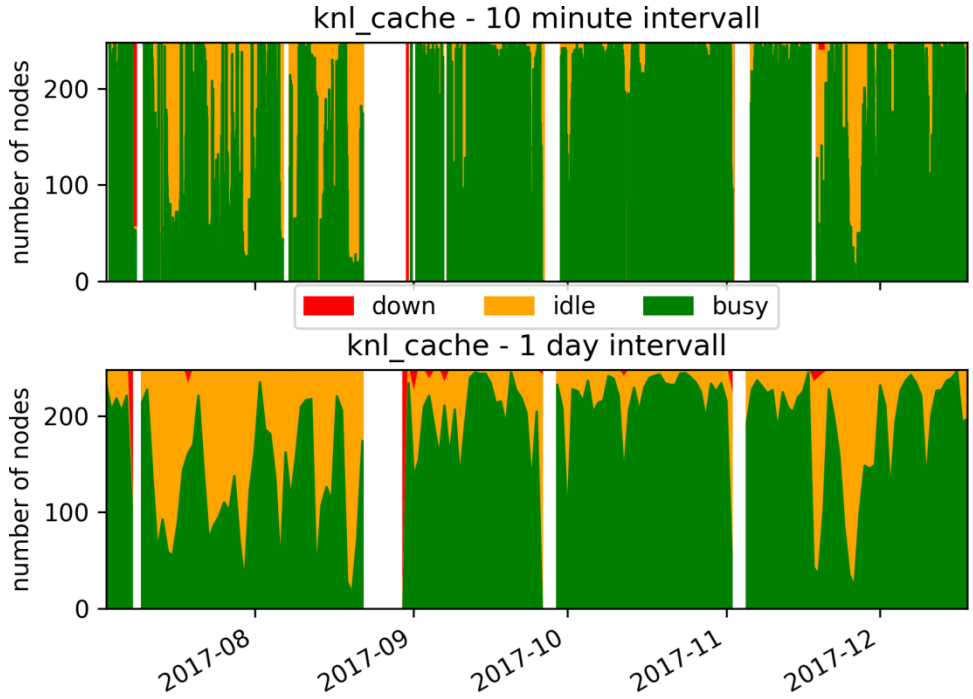


Fig. 93 Node usage for the KNL cache partition over time.

The node usage of both memory models is very different. The KNL flat partition (Fig. 92) experiences distinctly less usage in comparison to the KNL cache partition (Fig. 93). This can be explained by the fact that it was communicated to the users that the KNL cache partition is supposed to provide an easier access to more performance. Because of these statistics, the Operation Committee decided to switch all nodes to the cache memory model. Before that switch it must be made sure no users are strongly opposing it. Effective use of the KNL flat partition necessitates development effort which should not be wasted recklessly.

7.7.3. **SKL partition**

Marconi installed a new partition in August 2017. It is equipped with the new Intel Skylake architecture.

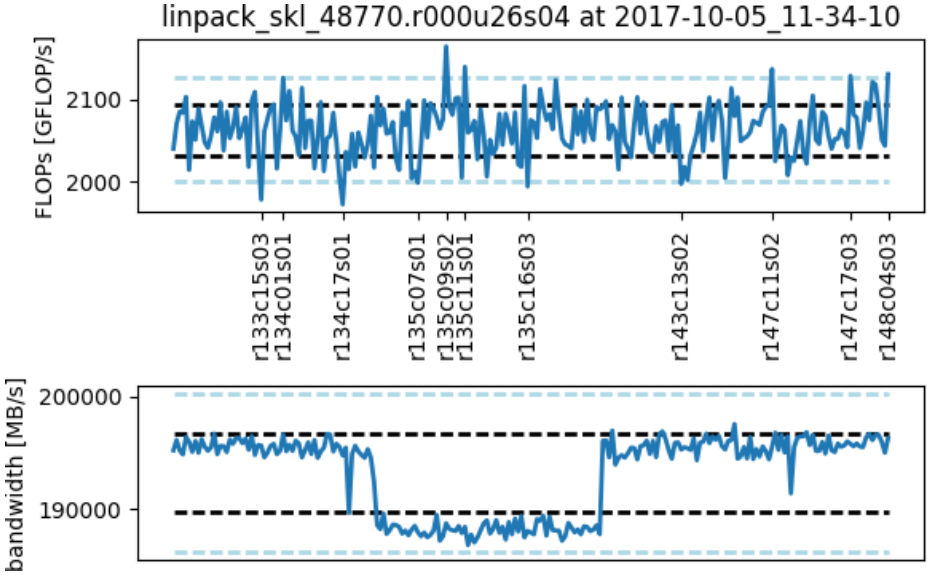


Fig. 94 October 2017 results of the SKL monitoring.

The continuous monitoring was quickly established for this new system. The results from October 2017 can be found in Fig. 94. The system is already very stable. The measured values are very close to the theoretical peak values given by 2.15 TFLOP/s for the peak computational capability and 238.36 GiB/s for the bandwidth.

One thing to note regarding the performance of the SKL partition is the emergence of the Intel Turbo Boost (ITB) technology. ITB allows increased CPU frequencies, depending on the used instructions (and the amount of active cores). An overview of this behavior is shown here (all cores active):

Mode / Instructions	Frequency		TFLOPs / node	
	Base	ITB	Base	ITB
Normal	2.1GHz	2.8GHz	3.23	4.30
AVX2	1.8GHz	2.5GHz	2.76	3.84
AVX512	1.4GHz	2.0GHz	2.15	3.07

The FLOPs amounts are calculated according to FLOPs = frequency × 32 × 48 where the factor 48 signifies the number of cores and the factor 32 stems from:

$$32 = \underbrace{512}_{\text{AVX512}} \Big/ \left(\underbrace{8}_{\text{per byte}} * \underbrace{8}_{\text{per double}} \right) * \underbrace{2}_{\text{VPUs}} * \underbrace{2}_{\text{FMAs}}$$

Employing this formula implicitly means that all the red values in the above table can never be achieved. The actual values are much lower as factors of up to 32 are lost. It is also more complicated to predict the performance of the system.

Unfortunately, ITB is not enabled on the Marconi system. It would increase the theoretical peak performance to 3.07 TFLOP/s but requires more energy and increased cooling.

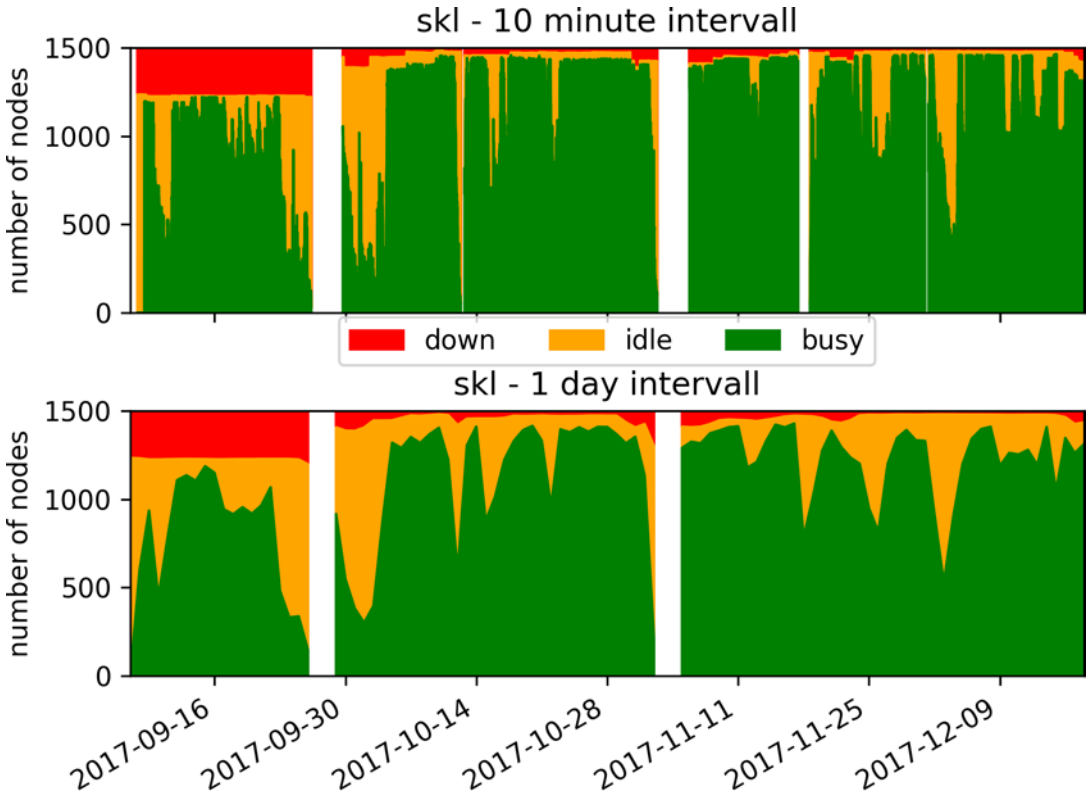


Fig. 95 Node usage for the SKL partition over time.

Node usage of the SKL partition (Fig. 95) is quickly rising. This is due to the fact that it is very easy to switch over from the Broadwell partition.

7.7.4. **Login node and file system uptime**

The uptime of the login server of Marconi is monitored as well, using a script written by our collaborator Jacques David. The resulting data can be found in Fig. 96.

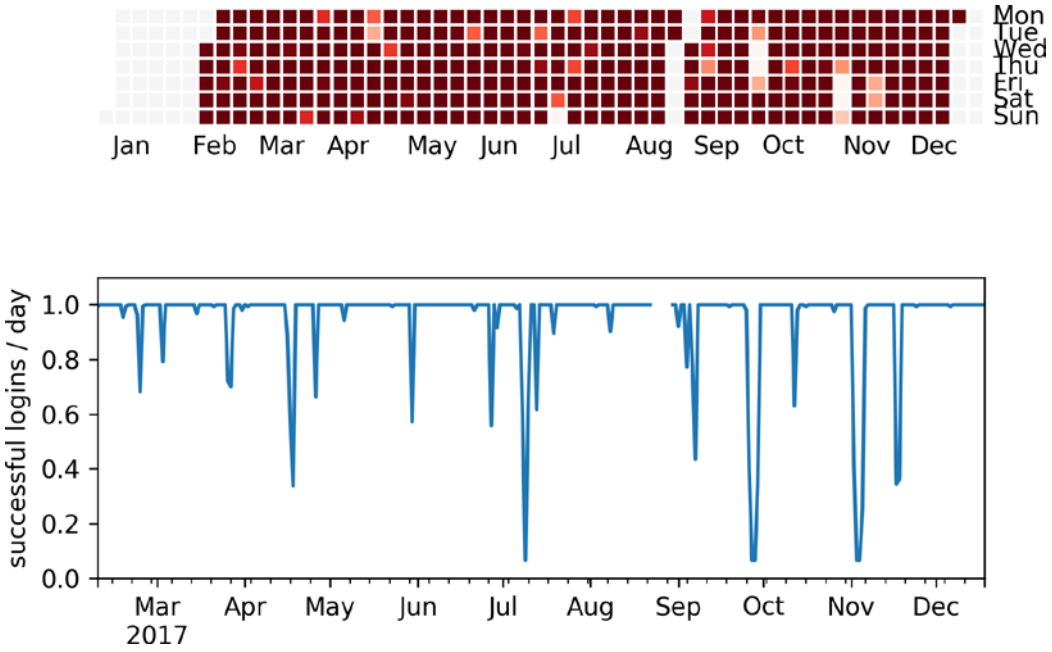


Fig. 96 Login node uptime of Marconi.

We also monitor the uptime of the Marconi file system. The resulting data can be found in Fig. 97. It uses a modified script, based on the script used for the login node uptime monitoring. File system monitoring was started much later than login node availability monitoring.

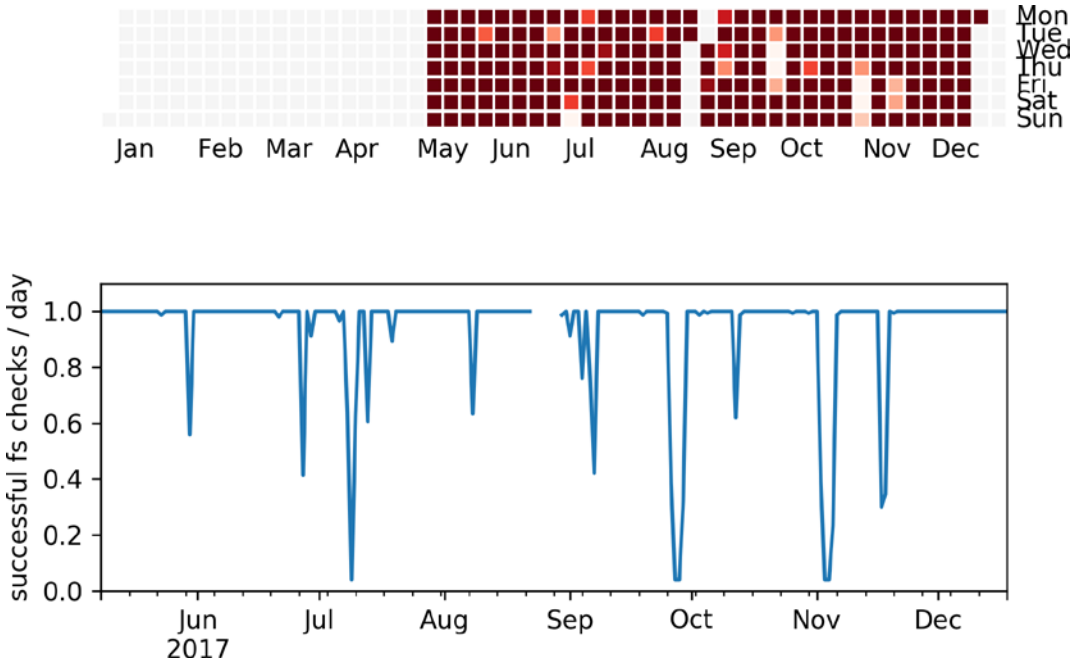


Fig. 97 File system uptime of Marconi.

7.8. ***References***

Dongarra, J., Bunch, J., Moler, C., & Stewart, G. (n.d.). Retrieved from www.netlib.org/benchmark/hpl

Hoefler, T., & Lumsdaine, A. (2008). Accurately Measuring Collective Operations at Massive Scale. IEEE, 978-1-4244-1694-3.

McCalpin, J. D. (n.d.). Retrieved from https://www.cs.virginia.edu/stream

8. **Final report on HLST project JORSTAR2**

The JORSTAR2 project is a continuation of the JORSTAR project [1] and dedicated to the implementation of parallel I/O in the STARWALL output and JOREK input subroutines. The large STARWALL matrices are distributed over MPI tasks to reduce memory consumption and to allow for running larger simulations in terms of the JOREK computational grid and the number of triangles used in STARWALL to discretize wall structures. A sequential part of JOREK in which the input matrices from the STARWALL code are used has to be parallelized as well.

8.1. *Goal of the project*

Large scale plasma instabilities inside a tokamak can be influenced by the currents flowing in the conducting vessel wall. In order to study this problem the code that solves the magneto-hydrodynamic (MHD) equations, called JOREK [2,3], was coupled [4] with the model for the vacuum region and the resistive conducting structure named STARWALL [5,6]. The JOREK-STARWALL model has already been used to perform simulations of the Vertical Displacement Events (VDEs), the Resistive Wall Modes (RWMs), Quiescent H-Mode [7], and vertical kick ELM triggering [8].

Thanks to the JORSTAR project [1] it is now possible to resolve the realistic wall structure with a large number of finite element triangles in the STARWALL code. However, the output subroutine is still sequential. This project concentrates on the MPI parallelization of the sequential I/O part in both the JOREK and STARWALL codes and adapting the JOREK code for using the STARWALL response matrices now distributed over MPI tasks.

8.2. *Parallelization of the STARWALL output subroutine*

Before starting the implementation of the parallel I/O modules in both STARWALL and JOREK a variety of libraries and subroutines were analyzed in order to find the best candidate.

Most linear algebra subroutines were parallelized in the previous JORSTAR project by means of the parallel ScaLAPACK library [9]. This library requires the so-called block-cycling matrix distribution format described in detail in [1]. Before they can be used in an output procedure, all these matrices have to be converted to a standard contiguous format. We first investigated if the ScaLAPACK library has a suitable subroutine for parallel I/O with a direct conversion to this standard format. Only one subroutine named *PDLAPRNT* was found. This subroutine collects all distributed local matrices, converts them from the block-cycling distribution format to the contiguous one and writes a global matrix. It could have been a straightforward solution for our problem as everything in the STARWALL code is already prepared for the ScaLAPACK library. However, after applying and testing this subroutine we realized that it only works for small problem sizes. The subroutine is written in such a way that only one MPI task locally collects all distributed matrices and afterwards writes them to a file. This makes the output very slow and restricts the maximum global matrix size to the memory capacity of one computing node or less (<180 GB on the Skylake partition of the Marconi supercomputer). Therefore, this subroutine does not meet our requirements as some global matrices of a STARWALL production run can have sizes of about 500 GB.

The next step was to test the ROMIO library which is an implementation of the MPI 3.0 standard. This library includes many different MPI I/O subroutines which were tested for our project. We started with *MPI_File_seek, MPI_File_write* and *MPI_File_write_at*. The first subroutine seeks to the writing position, while the second subroutine performs the writing itself. The last subroutine is a combination of the first two. These subroutines were working fine and provided the correct output. However,

they are not collective routines which makes the output very slow for large problem sizes (a couple of days for a matrix larger than 500 GB). Moreover, they require an additional calculation for transforming the block-cycling distribution to the contiguous format. Therefore, these subroutines are not suitable for our project.

Next, we tested the collective subroutine *MPI_File_write_at_all,* which has the same functionality as the previously described subroutines with the difference that all MPI tasks write simultaneously to a file. Again this subroutine works fine and much faster than the previous ones but it also has one restriction. Each MPI task has to call it the same number of times as it is a collective subroutine but sometimes the sizes of the local distributed matrices are not identical for each MPI task. Therefore, in such cases the subroutine gets stuck and the program deadlocks.

Finally, a solution was found by using the subroutines *MPI_Type_create_darray* and *MPI_File_set_view*. The former creates a description of any complex data structure, for example block-cycling distributed submatrices, while the latter defines an independent file view for each MPI task. Therefore, each MPI task can write its own specific data structure concurrently to the same file by means of a single call to the *MPI_File_write_at_all* subroutine described before. This method works very fast for any problem size and delivers correct results. Similar routines can be used in JOREK to provide a different distribution of matrices over MPI tasks already when reading them. This is described in the following section.

However, we decided to continue to investigate further possible candidates for our problem and tested the parallel HDF5 library as it was already successfully used in some parts of the JOREK code. We were able to achieve correct and fast performance for equally distributed matrices (each submatrix has the same size). However, we did not find a possibility by means of the HDF5 library to in parallel write not equally block-cycling distributed matrices. Therefore, we kept the solution described in the previous paragraph.

8.2.1. **STARWALL parallel I/O performance test**

After implementing the solution for each output matrix in STARWALL, performance measurements were conducted. Different problem sizes as well as different numbers of computing nodes involved in the writing process were tested. All tests were done on the Broadwell partition of the Marconi supercomputer, which offers 36 cores per node. The wall clock time for the complete output was about 83 seconds for a moderate problem size with *ntri_w*=120,000 finite element triangles in the wall using two computing nodes. The output file size for this case was about 110 GB. Only ~100 seconds were needed for a production run output with *ntri_w*=500,000 and *ntri_p*=202,000 using 64 computing nodes creating a file with a size of about 1 TB. A complete STARWALL production run, including all computations and output procedures, with *ntri_w*=500,000 and *ntri_p*=200,000 using 64 computing nodes takes about nine hours. This is much less time than is required by the project coordinator (<24 hours). At this step all work concerning the STARWALL code is finished.

8.3. ***Parallelization of the JOREK input subroutine***

The same solution that was used for the STARWALL output was applied to the JOREK input subroutine including only small modifications which are described here. The global matrices should not be read from the input file (i.e. the STARWALL output file) in the block-cycling distribution format but in an ordinary way using a row– or a column–wise distribution. Therefore, instead of the *MPI_Type_create_darray* subroutine, the *MPI_Type_create_subarray* subroutine was used. The data structure of the matrices was also modified. Instead of using a simple allocatable array, we introduced a data type that includes local allocatable submatrices, the starting and ending indices of the global matrix and the type of distribution (column–or row–wise).

Results obtained from the new parallel reading subroutine were compared with the old sequential version. They were identical up to a relative error of ~10^{-13}. Afterwards the execution time of the whole reading procedure for the production run (*ntri_w*=500,000, *ntri_p*=200,000) was measured. Using 16 computing nodes and 36 MPI tasks per node the wall clock time was less than one minute.

8.4. *Restrictions of the Intel MPI 3.0 library*

One important limitation in the subroutines *MPI_File_write_at_all* and *MPI_File_read_at_all* was found during the development of the parallel MPI I/O. The amount of elements to be read/written from/to a file by each MPI task is an input parameter for both subroutines. According to the Intel MPI documentation [10] this variable is a four byte integer that can have a maximum value of 2147483647. This corresponds to approximately two GB of data. For double precision (eight bytes per value) arrays, which are used in our codes, the maximum size of data that can be read/written by each MPI task is limited to 2147483647*8 bytes ≈ 16 GB. This means that if an array size is larger than *number_of_MPI_tasks**16 GB the code will fail. For a production run, the largest output matrix uses about 500 GB. Therefore, we need a minimum of 32 MPI tasks in order to overcome this limitation and to perform the correct reading/writing procedure. The next MPI standard, 4.0, should correct this limitation by changing the data type of the count variable. However, it was decided to introduce modifications in the reading subroutine in order to avoid this limitation. If the local matrix size is larger than 16 GB, the reading procedure will be performed in several steps, reading a data chunk that is less than 16 GB on each step.

As described above, and according to the MPI 3.0 standard, it should be possible to read 16 GB of data per MPI task in one operation for a double precision array. However, in the Intel MPI implementation of these subroutines, the variable *count* is multiplied by the type of the read array (eight for double precision) and the result of this operation is stored in a four byte integer variable. Therefore, if we try to read the maximum possible amount of elements (2147483647) for a double precision data type the code crashes with a segmentation fault. As long as this bug is present in the Intel MPI library we need to restrict our reading chunk size to less than two GB. This bug was reported to the Intel support team [11] and should be fixed in the next version of the library.

Additional modifications were made which ensure that each MPI task can read a large submatrix (> 2 GB) without any errors, circumventing the bug in the Intel MPI library.

8.5. *Parallelization of the JOREK subroutines*

After the JOREK parallel input was successfully developed and tested the rest of the code that uses the distributed matrices from STARWALL had to be parallelized as well. This mainly concerns the parallelization of the linear algebra operations.

8.5.1. **Data structure of distributed matrices**

A new data structure (Fig. 98) was introduced in JOREK in order to encapsulate properties of a distributed matrix. *loc_mat* represents the local chunk (two dimensional array) of a distributed matrix. *distrib* tells us if a matrix is distributed or not. *row_wise* shows the type of the distribution: if *row_wise=.true.* the matrix is distributed row–wise; if *row_wise=.false. the matrix is distributed* column–wise. *ind_start* and *ind_end* denote the starting and ending indices of the current chunk in the global matrix. *step* is the chunk size of the local matrix. *dim* defines the global matrix dimensions.

```
type :: t_distrib_mat
   real*8, allocatable :: loc_mat(:,:)
   logical             :: distrib
   logical             :: row_wise
   integer             :: ind_start
   integer             :: ind_end
   integer             :: step
   integer             :: dim(2)
end type t_distrib_mat
```

Fig. 98 Data structure for a distributed matrix.

8.5.2. **Parallelization of the *update_response* subroutine**

This subroutine constitutes most of the linear algebra calculations in the code that had to be parallelized. The most time consuming operation, which appears in many places in the code, is a generalized matrix-matrix multiplication. This operation needs to be done for different combinations of distributed matrices. For clarification we take the following example from the code:

response_m_e(:,:) = sr%a_ee(:,:) + matmul(sr%a_ey(:,:), response_m_a(:,:)).

Four matrices are used in this example: *response_m_e, a_ee, a_ey* and *response_m_a.* A matrix-matrix multiplication is performed between the *a_ey* and *response_m_a* matrices by using the standard sequential *matmul* subroutine. The resulting matrix is added to the matrix *a_ee* and finally saved as the *response_m_e* matrix. The difficulty is that the matrices can be distributed using different patterns (row–wise or column–wise). In addition, some small matrices in the code stay unmodified (they are not distributed). In our example, the matrices *a_ee* and *a_ey* are distributed via a row–wise pattern, the matrix *response_m_a* is distributed via a column–wise pattern and the matrix *response_m_e* is not distributed at all. There are many matrix-matrix multiplications in the JOREK code and their participating matrices have different distributions and different orders. Therefore, a suitable parallel matrix-matrix multiplication subroutine is required covering the whole spectrum of distributed (column– or row–wise) or non distributed matrices. Such a subroutine named *matrix_multiplication* was successfully developed. The subroutine works for all types of distributed matrices and generates identical results in comparison with the original code version.

Finally, the complete *update_response* subroutine was parallelized including the matrix-matrix operations as well as the matrix-vector calculations and matrix reassignments.

8.5.3. **Parallelization of the remaining part of the JOREK code**

The JOREK subroutines that include distributed matrices were parallelized next. We will not report in detail about each subroutine, because the parallelization procedure and the type of parallelization were very similar. The main difficulty was with sequential subroutines that were only called by the master MPI task. In the parallel version all tasks must call and enter these subroutines and corresponding changes for a correct execution were performed.

Here is a list of all subroutines which were parallelized: *get_vacuum_response, read_starwall_response, broadcast_starwall_response, log_starwall_response, update_response, coil_current_source, evolve_wall_currents, reconstruct_triangle_potentials, equilibrium, poisson, boundary_check, vacuum_equil, vacuum_boundary_integral.*

The accuracy of the modified subroutines was compared with the accuracy of the subroutines in the original code version. Both code versions (original and parallel) provide the same results with a relative error of 10^{-11}.

The parallel code version was also successfully tested for a production run using input matrices from STARWALL with a size of around 500 GB.

8.5.4. **OpenMP parallelization of the matrix multiplication subroutine**

During the performance tests described below it became clear that the matrix multiplication subroutine takes most of the computational time inside the *read_starwall_response* and *update_response* subroutines. Therefore, it was decided to implement an OpenMP parallelization on top of the MPI parallelization.

Results obtained from the new MPI+OpenMP subroutine were compared with the old MPI version. They were identical up to a relative error of ~10^{-12}. Afterwards the execution time of the multiplication of the two largest matrices in the code was measured. This test had the following parameters: *n_tor*=11, *n_period*=1, *n_plane*=32, *n_har*=6, *n_pol*=160, *nwu*=*nwv*=300 and *ntri_w*=180000. Using 18 computing nodes and 2 MPI tasks per node with 24 OpenMP threads the wall clock time was 6.6 s. This is 16.7 times faster in comparison to the old MPI version (110.7 s).

8.6. *Bugs in the original code version*

During the parallelization of the JOREK code a few bugs were found in the original code version and reported to the project coordinator. Among them use of uninitialized variables and wrong parameters in subroutines:

1) The variables *heat_src* and *part_src* in the file *diagnostic/integrals.f90* were without initialization for certain conditions.
2) In the file *vacuum/vacuum_response.f90* the variables: *old_thick*, *old_res*, *old_tstep*, *old_theta*, *old_zeta*, *old_reswall* were used without initialization. The main difficulty was that all these variables have the "*save*" attribute and the standard Intel Fortran debugging flag (-check uninit) can't detect them.
3) The variable *vertical_FB* in the file models/equilibrium.f90 was also used without initialization for certain conditions.
4) A wrong parameter was used in the subroutine *integrals*. Instead of using the variable *psi_bnd* the variable *psi_lim* was used. This caused wrong output results for one particular diagnostics of the JOREK code.

A few bugs were also found in the JOREK regression tests:

1) In the file *diagnostics/rst_hdf52bin.f90* and *diagnostics/rst_bin2hdf5.f90* a call to the initialization subroutine *update_time_evol_params* was missing.
2) One minor bug is still not resolved in the regression test named *freebound_equil_aug: "forrtl: error (65): floating invalid"* appears for the following line: write(11,'(8e16.8)') surface_list%psi_values(i), dp_int/sum_dl, zjz_int/sum_dl, F0 * q / (2.d0 * PI). The project coordinator will resolve this issue after the current project.

8.7. *Merging different JOREK development branches*

At the end of the project three branches of the JOREK code: (i) *develop* – the main branch, (ii) *feature/IMAS-668* – the branch for the current project and (iii) *feature/IMAS-961-speed-up-boundary-int* – the branch for speeding up the two most time consuming subroutines (implemented by the project coordinator) were merged. The resulting code version was tested for accuracy and performance (results are presented in the next section). Finally, a pull request on the *git* system was initiated in order to assign this code version as the main develop version.

8.8. ***Performance tests***

In this section we compare the performance of the most important JOREK subroutines for three code versions: (i) *develop* – the main version; (ii) the *speed up* version described in Sec. 8.7 and (iii) the *merge* of *develop*, *speed up* and the version for the current project (JORSTAR2). Table 18 shows the execution time of three different test cases for these three code versions.

One can see that the total wall clock time (last column) from the original *develop* version is much higher (depending on the test case) in comparison to the *speed up* and *merge* versions. For test case number three the *develop* version did not even finish within 24 hours of computation. On the other hand the *speed up* and *merge* versions provide very similar results for the test cases one and two. For test case number one the *speed up* version is a little bit faster in comparison to the *merge* version, while for test case number two the *merge* version takes the lead. Only the *merge* version is able to run very large problem sizes due to the STARWALL response matrices being distributed over all MPI tasks.

The performance improvement was mainly achieved in two subroutines named *vacuum_boundary_integral* and *global_matrix_structure_vacuum*. The computational algorithm was changed for the latter subroutine in a way which prevents an execution of the subroutine inside a loop. Therefore, it is called only once in the *merge* and the *speed up* version and 23 times in the *develop* version. One can also see the important improvement of the *vacuum_boundary_integral* subroutine, where the wall clock time of the *develop* version is 200–770 times higher in comparison to the wall clock time of the *merge* and the *speed up* version. This improvement was mainly achieved by reordering some nested loops (there are a total of 12 nested loops) inside an OpenMP region.

The *boundary_check* subroutine was in some cases slower in the *merge* version in comparison to the *develop* and/or *speed up* version. Therefore, it was decided to implement an OpenMP parallelization for it as well. The project coordinator was responsible for this part. The subroutine after the improvement is about one order of magnitude faster using 48 OpenMP tasks in comparison to the original version.

The *merge* code version works as fast as the *speed up* version and much faster than the original *develop* version. Moreover, the *merge* version can perform calculations with larger matrices due to the MPI parallelization and a global matrix distribution of the *vacuum_response* part of the JOREK code. This is the reason why test case number three fails in the *develop* and *speed up* version due to memory limitations. On the other hand, the *merge* version executes this test case without any problems.

1) Test case: n_tor=21, n_period=1, n_plane=64, n_har=11, n_pol=120, nwu=nwv=64, MPI=11, compute_nodes=11, OMP=48					
Code version	vacuum boundary integral	boundary check	update response	global_matrix structure vacuum	complete code
develop	2329,27	3,44	21,55	8,74*23	**54903**
speed up	5,89	1,04	21,01	0,62	**1177**
merge	4,57	4,63	12,49	0,60	**1308**

2) Test case: n_tor=21, n_period=1, n_plane=64, n_har=11, n_pol=160, nwu=nwv=64, MPI=11, compute_nodes=11, OMP=48					
Code version	vacuum boundary integral	boundary check	update response	global_matrix structure vacuum	complete code
develop	11147,05	8,74	92,05	19,46*10	**>24 hours**
speed up	14,45	2,53	37,81	1,19	**1883**
merge	9,91	6,20	22,54	1,03	**1649**

3) Test case: n_tor=11, n_period=1, n_plane=32, n_har=6, n_pol=160, nwu=nwv=330, MPI=48, compute_nodes=24, OMP=24					
Code version	vacuum boundary integral	boundary check	update response	global_matrix structure vacuum	complete code
develop	Not enough memory				
speed up	Not enough memory				
merge	1,35	9,52	213,80	0,29	**2385,00**
All values are given in seconds					

Table 18. The wall clock time in seconds of the most important JOREK subroutines for three different test cases for three code versions.

8.9. *Scalability tests*

We also tested how the computational time scales for the final *merge* version for different parameters of the code. Fig. 99 shows the wall clock time versus (a) the number of finite element triangles representing the wall, (b) versus the number of MPI tasks and (c) versus the number of the nodes in the poloidal direction in the JOREK grid. In test (a) we kept the number of MPI tasks=12 and *n_pol*=200 constant; in test (b) we set *ntri_w*=100,000 and *n_pol*=200 and in test (c) we used MPI tasks=12 and *n_pol* =200.

The total computational time for all test cases (even for production runs with *ntri_w*=300,000 or *n_pol*=300) and for any presented subroutine is not longer than 70 seconds. These results are satisfactory as all these subroutines, except for the *boundary_check,* are called only once. The *boundary_check* subroutine is executed inside a small loop, however the calculation time of this subroutine is less than two seconds for a large production run which makes its influence on the total computational time relatively small. Besides, the parallelized *read_starwall_response*

and *matrix-matrix* multiplication subroutine scale quite well. All obtained data, which is used for Fig. 99, is summarized in Table 19.

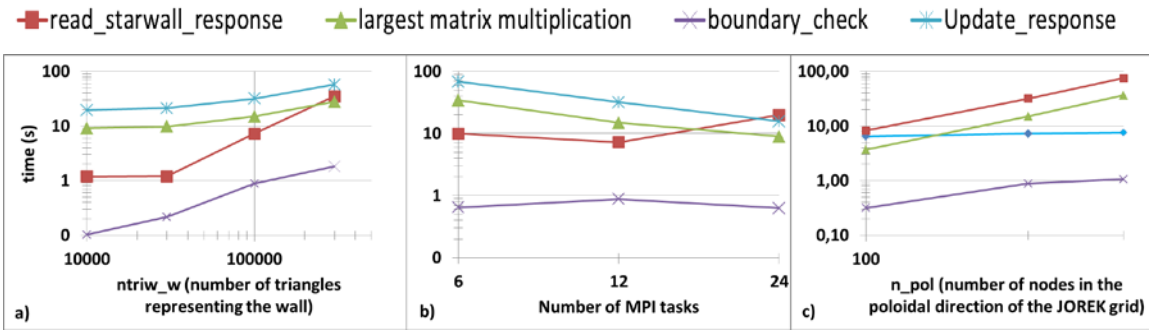


Fig. 99 Computational time in seconds of some JOREK subroutines versus different parameters of the code. The following constant parameters were used in all calculations: *n_tor*=11, *n_period*=1, *n_plane*=32, *n_har*=6.

MPI	ntri_w	n_pol	read starwall response	update response	Matrix–matrix multiplication	boundary check
12	10,000	200	1,19	19,65	9,06	0,10
12	30,000	200	1,20	21,19	9,74	0,22
12	100,000	200	7,24	31,77	14,88	0,88
12	300,000	200	35,03	57,33	27,72	1,81
6	100,000	200	9,96	68,47	34,32	0,64
24	100,000	200	19,93	15,68	8,91	0,62
12	100,000	100	6,47	8,18	3,68	0,32
12	100,000	300	7,64	75,20	36,32	1,06

Table 19 Computational time in seconds of the four parallelized subroutines in the JOREK-STARWALL part.

The execution time of one time step in the global loop was measured in the JOREK code with and without the STARWALL part in order to estimate the overhead of this part of the code (Fig. 100). The time of the JOREK-STARWALL run is about a factor of two higher in comparison to the pure JOREK run. It grows with increasing problem size, while the JOREK part stays constant (a). This is because the tested parameter (*ntri_w*) has no influence on the JOREK part.

There are five nested subroutines in the JOREK-STARWALL part: *construct_matrix* → *vacuum_baundary_integral* → *evolve_wall_currents* → *write_wall_vtk* → *reconstruct_triangle_potentials*. It was measured that the time difference between the pure JOREK and the JOREK-STARWALL code version came mainly from the last two subroutines *reconstruct_triangle_potentials* and *write_wall_vtk.* For example, the execution time of the *write_wall_vtk* subroutine for the test case Fig. 100 (a) with *ntri_w*=300,000 is 21 seconds. In comparison the kernel loop of the *vacuum_baundary_integral* subroutine for this test case takes about two seconds. Thus, *write_wall_vtk* can be the first candidate for a future code optimization. All obtained data, which is used for Fig. 100, is summarized also in Table 20.

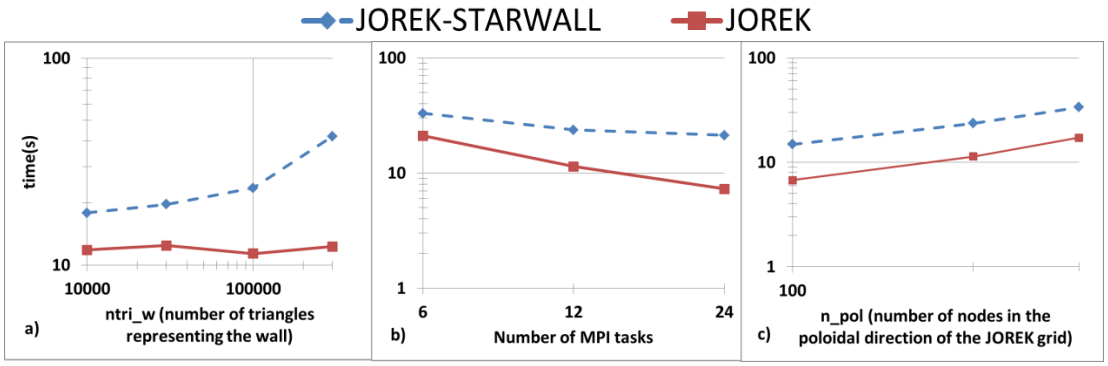


Fig. 100 Computational time in seconds of one global loop step in the JOREK code with the STARWALL part (dashed blue line) and without (solid red line) versus different parameters of the code. The following constant parameters were used in all calculations: *n_tor*=11, *n_period*=1, *n_plane*=32, *n_har*=6.

MPI	ntri_w	n_pol	JOREK	JOREK–STARWALL
12	10,000	200	11,85	17,88
12	30,000	200	12,42	19,69
12	100,000	200	11,37	23,60
12	300,000	200	12,29	42,00
6	100,000	200	21,05	32,92
24	100,000	200	7,27	21,27
12	100,000	100	6,75	14,85
12	100,000	300	17,17	33,60

Table 20 Computational time in seconds of one global loop step in the pure JOREK and JOREK–STARWALL version.

Additionally, a prediction for the memory consumption (total and per MPI task) was implemented in JOREK and STARWALL to help the user to choose the appropriate number of MPI tasks.

8.10. *Conclusions*

Different libraries (e.g. ScaLAPACK, HDF5 and MPI) were analyzed in order to find the best possible solution for parallel I/O. The MPI library was chosen as it can directly translate the format of the output submatrices from the block-cycling distribution to an ordinary format during the writing procedure.

MPI parallel I/O was implemented in both the STARWALL output and the JOREK input subroutines. The execution time of the complete reading and writing procedure for a production run is only up to a few minutes when using 16 and 64 computing nodes.

During the development, a bug was found in the Intel MPI library that significantly limits the size of the read/written data per operation. It was reported to the Intel support team and should be corrected within the next version of the library. In order to be able to work with large matrices and to overcome this bug, several subroutines were modified. Several small bugs were also found and corrected in the code.

All sequential subroutines that use distributed matrices from the STARWALL input file were parallelized and tested. All of them provide identical results in comparison to the original code version.

The final version of the code, achieved during this project, was merged with the *develop* branch and with the *speed up* version of the code. The final code version provides results much faster than the *develop* version and can work with very large matrices from STARWALL output.

8.11. ***References***

[1] Mochalskyy S. et al, “Report on the MPI parallelization of the resistive wall code STARWALL (Project of the EUROfusion High Level Support Team)”, (2016).
[2] Huysmans G.T.A. and Czarny O. MHD stability in X-point geometry: simulation of ELMs NF 47, 659 (2007)
[3] Czarny O. and Huysmans G. Bézier surfaces and finite elements for MHD simulations JCP 227, 7423 (2008)
[4] Hoelzl M., Merkel. P., Huysmans G.T.A., Nardon E., McAdams R., Chapman I. Coupling the JOREK and STARWALL Codes for Non-linear Resistive-wall Simulations. Journal of Physics: Conference Series, 401, 012010 (2012)
[5] Merkel P. and Sempf M. 2006 Proc. 21st IAEA Fusion Energy Conf. (Chengdu, China) TH/P3-8; URL
http://www-naweb.iaea.org/napc/physics/FEC/FEC2006/papers/th_p3-8.pdf
[6] Merkel P., Strumberger E., Linear MHD stability studies with the STARWALL code arXiv:150804911 (2015)
[7] Hoelzl M., Huijsmans G.T.A., Merkel P., Atanasiu C., Lackner K., Nardon E., Aleynikova K., Liu F., Strumberger E., McAdams R., Chapman I., Fil A. Non-Linear Simulations of MHD Instabilities in Tokamaks Including Eddy Current Effects and Perspectives for the Extension to Halo Currents. Journal of Physics: Conference Series 561, 012011 (2014).
[8] Artola F.J., Huijsmans G.T.A., Hoelzl M., Beyer P., Loarte A., Gribov Y. Non-linear magnetohydrodynamic simulations of Edge Localised Modes triggering via vertical oscillations. Nuclear Fusion (in preparation).
[9] http://www.netlib.org/scalapack/
[10] http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf
[11]https://software.intel.com/en-us/forums/intel-clusters-and-hpc-technology/topic/737449

9. **Final report on HLST project REFMUL3**

9.1. ***Introduction***

Simulation of reflectometry using a Finite-Difference Time-Domain (FDTD) code is one the most popular numerical techniques used, as it offers a comprehensive description of the plasma phenomena. However, to keep the error to a minimum, this method requires a fine spatial grid discretization, which also implies a high-resolution time discretization to comply with CFL stability condition. As a consequence, simulations in three-dimensional become prohibitive in terms of both memory resources and computational time. This is particularly true if one considers full-sized simulations of large devices, like JET or ASDEX Upgrade or even next generation machines like ITER. The only way to circumvent these issues is to develop a memory distributed parallel three-dimensional code. The REFMUL3 project aims precisely at this goal, namely, obtaining a scalable parallel three-dimensional FDTD simulation code with the same name. The REFMUL3 project aims precisely at this goal, namely, obtaining a scalable parallel three-dimensional FDTD simulation code with the same name. REFMUL3 is in an early stage of development. Since such a code has forcibly to be parallel, it makes sense to act at an early stage to enforce a structure compatible to the parallelization needs. Moreover, the parallelization solutions developed during the HLST-REFMULXP (2013) and HLST-REFMUL2P (2014) projects for the bi-dimensional version of the REFMUL<*> code family serve as a sound starting point for the work proposed here.

9.2. ***Single-core optimisation***

9.2.1. **Code checking and profiling**

REFMUL3 is a full-wave code using a FDTD Yee scheme [1] with full polarization, able to cope simultaneous with o- and x-mode, which supports a general external magnetic field and a dynamic plasma. Its serial version constitutes the starting point for this project, and as such, the first step, after some basic checking and porting to the machines at hand (TOK cluster available at IPP, HELIOS available at CSC Japan at the time and its replacement Marconi at Cineca Italy), is to profile the code in terms of computational cost. To that end, measurements with GPROF were made on the IPP's TOK-I Linux cluster. The results are listed in Table 21. The rows depicted in magenta correspond to the three most expensive subroutines, which together are responsible for about 68% of the total cost. If we add up the remaining kernel routines of the code, depicted in green, plus the addFLDCUDE subroutine shown in black, then we reach 98% of the total cost. This is the expected behaviour, since it is in these subroutines that most of the floating-point operations (FLOPs) are computed. This also indicates clearly where to start to increase REFMUL3's performance, namely, optimise the single-core calculation of FLOPs.

9.2.2. **Optimisation steps**

Since REFMUL3 is written in C, the first thing to do is to check that the pointers used in these subroutines have their scope properly restricted. If data modifications through any given pointer can not affect the values read through any other pointer within the same scope, which in this case means within the data-loops inside the kernel subroutines, then pointer aliasing is not required and this information should be explicitly available at compile time (for instance by declaring the corresponding pointers with the __restrict keyword). This can in principle aid the compiler in its optimisation tasks by relaxing the need for data reloading to the hardware registers at every loop iteration. It was verified that the pointer non-aliasing properties were properly declared in REFMUL3, which came as no surprise since this knowledge had been already exploited successfully within projects HLST-REFMULXP (2013) and HLST-REFMUL2P (2014) on the predecessor bi-dimensional codes of the REFMUL<*> family.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
22.92	639.69	639.69	200	3.20	3.20	calcJxField
22.87	1278.14	638.45	200	3.19	3.19	calcJzField
22.86	1916.28	638.14	200	3.19	3.19	calcJyField
8.06	2141.14	224.86	600	0.37	0.37	addFLDCUBE
2.40	2208.11	66.97	100	0.67	0.67	calcEyxField
2.39	2274.89	66.78	100	0.67	0.67	calcEzxField
2.09	2333.32	58.43	100	0.58	0.58	calcEyzField
2.07	2391.18	57.86	100	0.58	0.58	calcExzField
1.77	2440.49	49.31	100	0.49	0.49	calcHzxField
1.70	2488.01	47.52	100	0.48	0.48	calcHyxField
1.63	2533.50	45.49	100	0.45	0.45	calcExyField
1.61	2578.39	44.89	100	0.45	0.45	calcHxzField
1.60	2623.03	44.64	100	0.45	0.45	calcEzyField
1.59	2667.53	44.50	100	0.45	0.45	calcHyzField
1.21	2701.40	33.87	100	0.34	0.34	calcHzyField
1.20	2734.94	33.54	100	0.34	0.34	calcHxyField
1.02	2763.27	28.33	61	0.46	0.46	initFLDCUBE
0.71	2783.00	19.73	100	0.20	0.20	setPEC

Table 21 GPROF profiling measurement of the original REFMUL3 serial code on a representative spatial grid-count of N_x=800, N_y=450 and N_z=450, for a fraction of the times steps of a production run.

The next step is to verify that the expressions used in the numerical kernel subroutines are implemented in a way to minimize the number of FLOPs inside the loops. In practice this means that all FLOPs involving constants should be moved outside the loops, as these potentially can stay in the hardware registers. Also, expressions should be changed to put in evidence common terms, as well as to minimize the number of divisions required. The following very basic example illustrates the idea, whereby one should replace the following expression

$$f(x) = \frac{g(x)}{1-a(x)} + \frac{a(x)\cdot g(x)}{1-a(x)} + \frac{h(x)}{1-a(x)}$$

which involves six FLOPs and three divisions, with the equivalent expressions

$$b(x) = \frac{1}{1-a(x)}$$
$$f(x) = \{[1+a(x)]\cdot g(x) + h(x)\}\cdot b(x)$$

obtained by introducing an extra quantity *b* whose expression allows to obtain the same result with only five FLOPs and one division. The reason for reducing the number of divisions and replacing them with multiplications whenever possible, lies in that a division needs significantly more clock cycles than a pure FLOP (a multiplication or an addition).

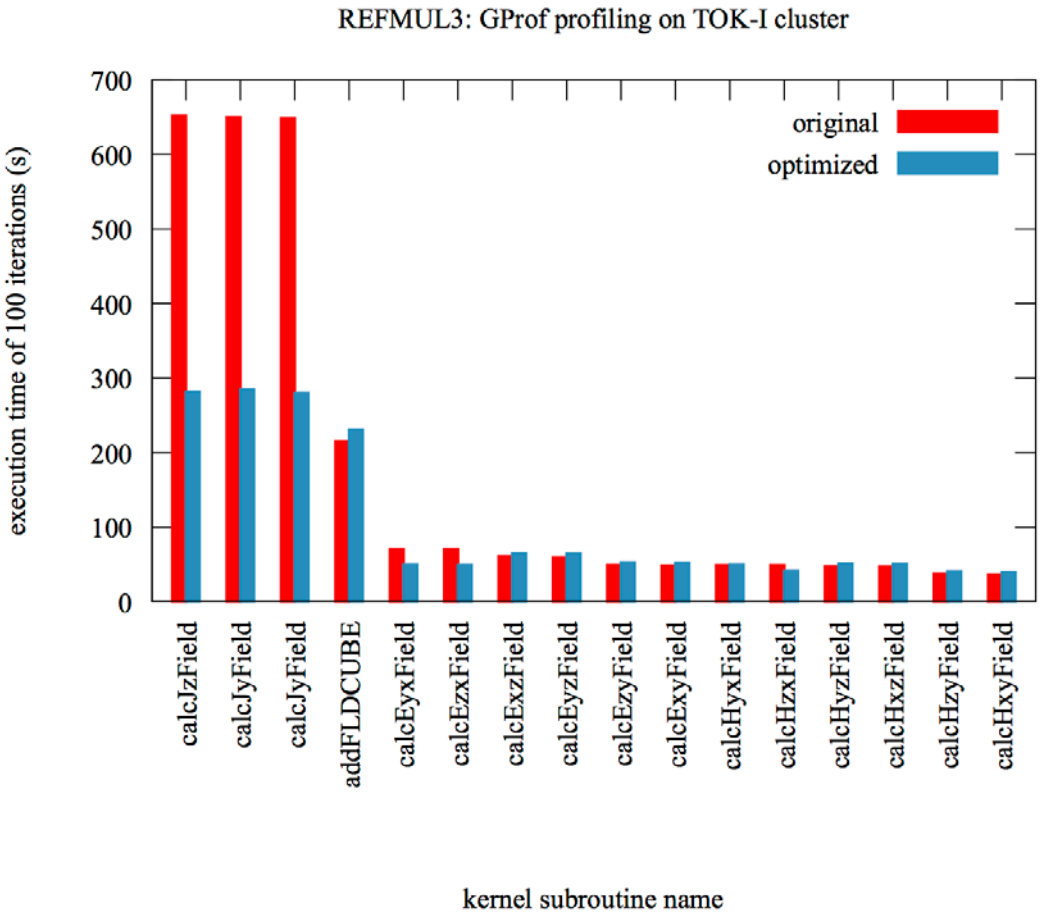


Fig. 101 Optimisation of the hotspots (costly subroutines) in the serial code. The red bars represent the original cost in seconds and the blue bars correspond to the cost of the optimised versions. Same problem size used for Table 21 was used here.

Minor modifications were made to comply with these rules in most of the kernel subroutines, except for the three represented in magenta in table Table 21, which according to the project coordinator (PC), had not yet been optimised in this sense. These required therefore a more extensive set of modifications. Their impact is shown in Fig. 101. As expected only these subroutines showed a significant improvement, of the order of 2.3x. For the others, the differences are not really statistically relevant. The measurements presented here correspond to a single run for each of the two versions of the code (original and optimised) compiled with the Intel C compiler. As a rule of thumb applicable for any scientific computation intensive code, it is always a good idea to apply the simple FLOP minimization rules outlined above, since they cost very little effort and can potentially yield quite significant single core performance gains.

9.3. *Parallelization strategy*

After the initial basic serial optimisation of REFMUL3, the work proceeds towards the main project goal, namely, the implementation of a parallel version, followed by the assessment of the resulting performance gains. Since the idea is to scale the code beyond a single compute-node, the use of a domain decomposition invoking the MPI standard for explicit communication is planned. However, in order to gain experience with the code and, at the same time, improve the code's performance within a smaller time investment, we start by parallelising the code's hotspot subroutines using OpenMP threads. The implementation of the MPI parallelisation will then follow, in order to have both parallel paradigms available depending on the user's choice, including a simultaneous combination of both, to provide a so-called hybrid OpenMP/MPI version. This has the additional benefit of being in line with the present

tendency of increasing the number of cores-per-processor/socket that share common memory regions.

9.3.1. **Thread parallelism**

Since the numerical kernel is largely symmetric in its three spatial directions, both in terms of the numerical stencil, as well as of the domain's grid-count, the most efficient way to proceed is to apply the thread-based parallelisation #pragma omp parallel for to the slowest varying index (outer loop), which in REFMUL3 corresponds to the *z*-direction. Doing so on all the subroutines listed on Table 21 (except for setPEC) and taking care to properly set the data scoping (shared/private variables) inside the parallel regions, as well as to do multi-thread initialization of the corresponding memory (initFLDCUBE), led to a speedup factor of 18.6x, when using 36 threads on a Marconi Broadwell node, compared to the serial version of REFMUL3 we started the project with. The measurements were made on the largest case provided by the project coordinator at the time, a 800x450x450 domain in the *x*, *y* and *z* directions, respectively, corresponding to a representative set of code input parameters, evolved for a reduced, but nevertheless sufficient to obtain meaningful measurements (6000 iterations). Disabling I/O operations, the original serial code took 58 hours to finish, whether the threaded (optimised) version took three hours (36 threads) to do the same task.

When switched on, the cost of the I/O operations specified in this test-case, implemented sequentially in the code, adds already over one hour to the simulation time, which for the threaded code corresponds to more 30% of the total runtime cost. Of course this number varies significantly depending on how much data one decides to write to disk during the simulation. Nevertheless, this figure provides a clear message. Namely, that the I/O operations become a strong bottleneck for the parallel code, already at the node-level, even before moving to the MPI domain decomposition that distributes the problem over several nodes. This subject is going to be revisited later in section 9.6, but for the time being it is removed from the analysis reported in the next few sections, which are concerned only with the scaling of the numerical algorithm.

The thread parallelism together with the single-core optimisation allow to run REFMUL3 one order of magnitude faster that originally. In practice, this enables already unprecedented results, like the ones illustrated in Fig. 102, from a simulation on a 720x700x700 grid in (*x,y,z*), which took about 7 hours to complete 6000 iterations using 36 threads on a Marconi Broadwell node (discarding I/O operations). Although positive, this milestone can only be considered an intermediate one for two main reasons. The first being that the problem size just discussed is the biggest that fits in a Marconi node, which means that bigger problem sizes are disallowed by the shared-memory (OpenMP) version of REFMUL3. The second reason is simply that the speedup obtained is still not enough. The test-case used was evolved for a reduced number of iterations, whereas a proper production simulation on the same grid-count would demand a one or two orders of magnitude higher number of iterations to converge. Further, bigger physics-relevant grid-counts, by one to three orders of magnitude are desired. In practice, this means that at least an additional order of magnitude in speedup must be achieved, although two would be a more desirable target. For that to happen, one has to be able to distribute the problem over several computational nodes, or in other words, a domain decomposition with explicit inter-core data communication is implied.

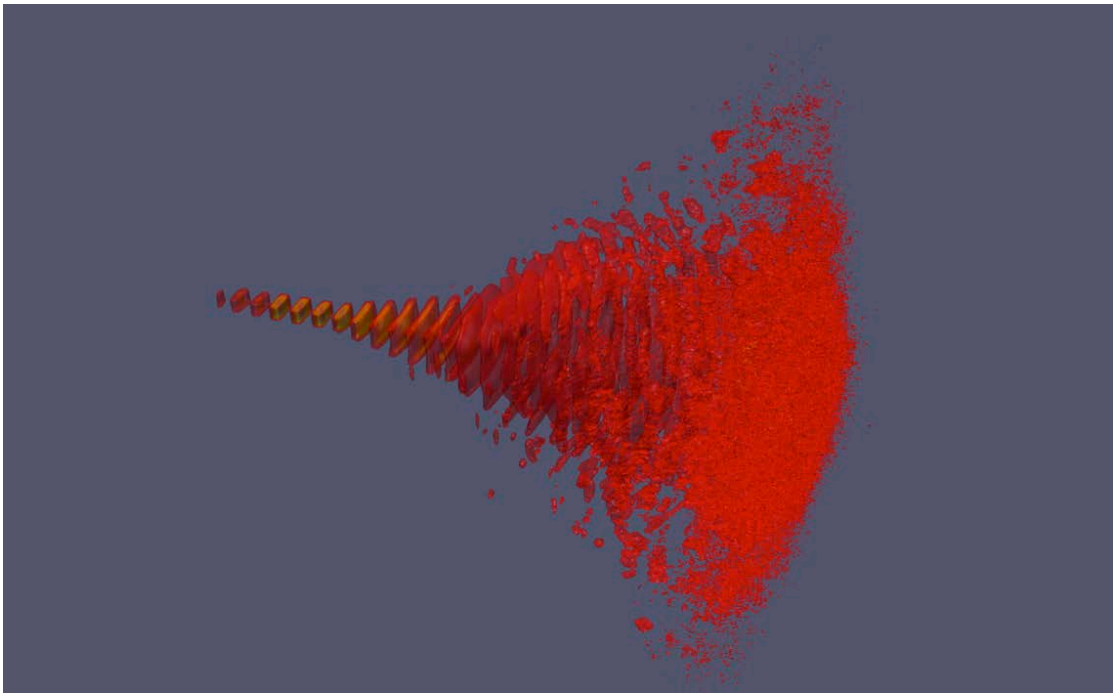


Fig. 102 Snapshot of the *z*-component of the electric field of a microwave wave propagating in the presence of a turbulent plasma. The domain was discretized on a 720x700x700 grid in the *x*, *y* and *z* directions, respectively. The simulation used 36 OpenMP threads and took about 7 hours to finish 6000 iterations (discarding the I/O cost).

9.3.2. **Task parallelism: cost/benefit**

In the course of projects HLST-REFMULXP (2013) and HLST-REFMUL2P (2014), the bi-dimensional domain of the code REFMULX was successfully decomposed over one of the dimensions. Comparatively, REFMUL3's three-dimensional numerical kernel has a similar structure, simply involving more equations to be solved. So, deploying the same parallelisation concept could be a valid option, mainly because it would be essentially risk-free from the technical point of view. However, being less ambitious comes at a price. In terms of problem sizes, the higher dimensionality of REFMUL3's domain further translates into increasing the grid-count by two or three orders of magnitude, depending on the size used for the additional third dimension. Given that the maximum number of resources that can be used in parallel is limited by the number of grid-nodes in the direction over which the domain is distributed, this yields a too strict upper bound on the maximum theoretical speedup that can be achieved. Therefore, extending the domain decomposition to at least another dimension is mandatory.

Another argument in favour of increasing the dimensionality of the domain decomposition is related to the ratio between data communication and the computation effort. It is easy to deduce that, minimizing such ratio, whose numerator is proportional to the surface of the boundaries between sub-domains, and denominator to the volume of the sub-domains, requires having sub-domains with similar grid-counts in all directions. To demonstrate this, consider the following simple example.

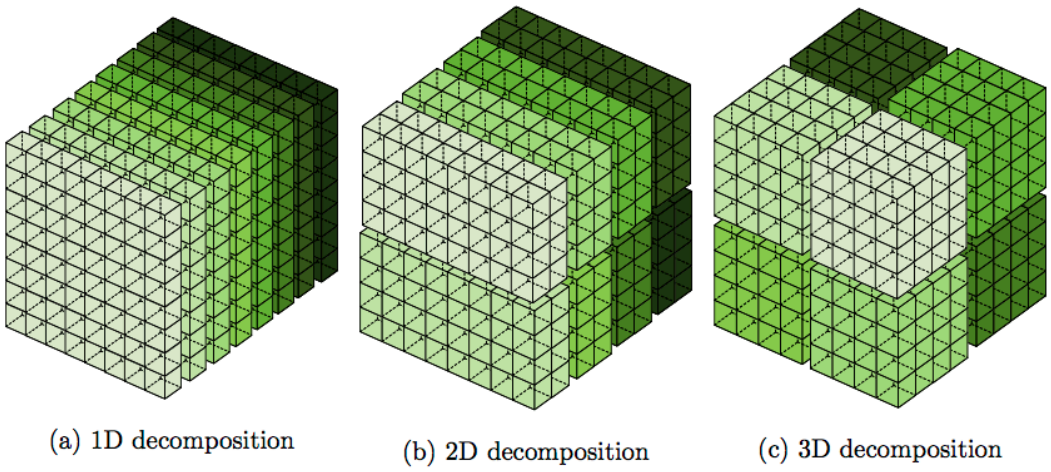


Fig. 103 Illustration of a N^3-sized three dimensional (x,y,z) data-set decomposed in the z-dimension (1D), in both (y,z)-dimensions (2D) and in all three (x,y,z)-dimensions (3D).

Assume you have a N^3 structured Cartesian grid in (x,y,z), a symmetrical numerical stencil in all three dimensions and a halo size w, which is the number of ghost-cells to store neighbouring grid-node values on each sub-domain boundary. Then, a one-dimensional domain decomposition over p processes in the z-direction, Fig. 103a, leads to an amount of communication per sub-domain proportional to

$$N^2 \cdot 2 \cdot w \cdot 1, \tag{1}$$

namely, one xy-plane of depth w per boundary in the z-dimension. On the other hand, if the decomposition is made instead over two dimensions, namely over y and z, as illustrated in Fig. 103b, then the amount of communication required per sub-domain becomes $\left(N^2/p_z + N^2/p_y\right) \cdot 2 \cdot w$, where p_z and p_y are the amount of processes in the y and z dimensions, respectively, such that $p_z \cdot p_y = p$. Further assuming for simplicity that $p_z = p_y = \sqrt{p}$ leads to

$$N^2 \cdot 2 \cdot w \cdot \frac{2}{\sqrt{p}} \tag{2}$$

Finally, applying the same reasoning to a decomposition over all three dimensions as schematised in Fig. 103c yields a value for the sub-domain communication proportional to

$$\left[N^2/(p_z \cdot p_y) + N^2/(p_z \cdot p_x) + N^2/(p_y \cdot p_x)\right] \cdot 2 \cdot w$$

that with the assumption $p_z = p_y = p_x = \sqrt[3]{p}$ further simplifies to

$$N^2 \cdot 2 \cdot w \cdot \frac{3}{(\sqrt[3]{p})^2}. \tag{3}$$

Comparing the factors in red in the previous expressions leads to the following conclusions: (i) for an isotropic domain, a bi-dimensional domain decomposition is more communication-effective than a one-dimensional counterpart whenever the number of processes used is $p>4$; (ii) a three-dimensional domain decomposition becomes the most efficient of all when the number of processes used is $p>11$. In sum, we have

$$\frac{3}{(\sqrt[3]{p})^2} < \frac{2}{\sqrt{p}} < 1 \qquad \text{if} \qquad p > 11.$$

Such rules hold for REFMUL3 as well, whose typical grid-count in all three spatial directions is comparable, and the numerical stencil depends isotropically on nearest-neighbours only (w=1). Additionally, it is such domain decomposition that obviously maximizes the number of cores over which a given domain size can be distributed,

with the upper limit being to use as many cores as grid-nodes in the domain. In other words, we can push the strong scaling limits to the maximum in this case.

Therefore, it makes sense to proceed with its implementation, which moreover has the additional advantage that it can easily be reduced to the bi- or one-dimensional domain decomposition cases. This can be done with simple changes to the source code, simply setting explicitly to unity the number of resources allocated to one or two of the three domain directions, respectively, and adapting accordingly the distribution of the requested MPI tasks over the remaining direction(s). However, there is an obvious disadvantage of a three-dimensional domain decomposition. Namely, from the technical point of view, it is much more complicated than the lower dimensionality decomposition counterparts. To minimise the time investment risk we start by checking the expectation that the potential performance benefits compensate the higher implementation cost implied, by implementing and assessing the concept outside REFMUL3, in a simpler standalone test-code.

9.4. *Standalone test-code: Jacobi iteration solver*

To develop and test the three-dimensional domain decomposition concept before it is attempted inside REFMUL3, a standalone test-code is introduced. A three-dimensional Poisson problem solver using the Jacobi iteration method was chosen, mainly for its simplicity, while still retaining the main property of the FDTD algorithm at hand from the MPI communication point-of-view. Namely, it is a three-dimensional explicit method with a stencil that depends on nearest neighbours only.

The method seeks for solutions *u* of the Poisson equation with the source term *f*

$$\nabla^2 u(x,y,z) = f(x,y,z)$$

by solving a discrete version obtained by replacing the continuous derivatives with their central finite difference discrete counterparts

$$\frac{u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k}}{h_x^2} + \frac{u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k}}{h_y^2} + \frac{u_{i,j,k+1} - 2u_{i,j,k} + u_{i,j,k-1}}{h_z^2} = f_{ijk}$$

where the labels (*i*,*j*,*k*) and the quantities (h_x, h_y, h_z) correspond to the grid-node indexes and their (equidistant) inter-node distance, in the (*x*,*y*,*z*) directions, respectively. The discrete approximate solution is obtained by iteratively computing

$$u_{ijk}^{m+1} = \frac{h_y^2 h_z^2 (u_{i+1,j,k}^m + u_{i-1,j,k}^m) + h_x^2 h_y^2 (u_{i,j+1,k}^m + u_{i,j-1,k}^m) + h_x^2 h_y^2 (u_{i,j,k+1}^m + u_{i,j,k-1}^m) - h_z^2 h_y^2 h_z^2 f}{2(h_x^2 h_y^2 + h_y^2 h_z^2 + h_x^2 h_z^2)}$$

while specifying the solution at the domain boundaries, where the labels *m*+1 and *m* represent the current and the previous iteration, respectively. This algorithm is implemented in a C code with the domain decomposed in all three dimensions. The implementation description is postponed until the next section, where the parallelisation concept, also applied to REFMUL3, is explained in detail. For the moment we focus solely on the scaling results obtained.

Fig. 104 shows the strong scaling obtained when running our Poisson Jacobi solver on the TOK-P cluster, at IPP. A similar domain size to the test-case given by the project coordinator for REFMUL3 was used, namely, a grid-count of 800x450x450 in (*x*,*y*,*z*). The code was executed for 6000 iterations and the elapsed time per iteration is plotted. Firstly, this result by itself suggests the technical feasibility of the concept within the available project time budget. Secondly, it is interesting to note that a similar intra-node strong scaling behaviour was obtained with the bi-dimensional REFMULX code (dashed yellow curve), here re-scaled to be superimposed with the red curve. This points in the direction that the Jacobi iteration algorithm is a good choice to mimic the scaling behaviour of REFMUL3's FDTD algorithm, as we suggested before. However, its inter-node scaling, being closer to the ideal than that of REFMULX, supports the argument that a three-dimensional domain decomposition

should be advantageous in REFMULX, and hence motivates the decision to implement it therein. In any case, the actual behaviour of this code's parallel scalability is hard to predict since it depends on the details of its algorithm, as well as the problem sizes under consideration. Therefore, this question can only be settled unequivocally by means of scaling measurements made using REFMUL3 after it has been parallelised in this fashion. This is the subject of the next section.

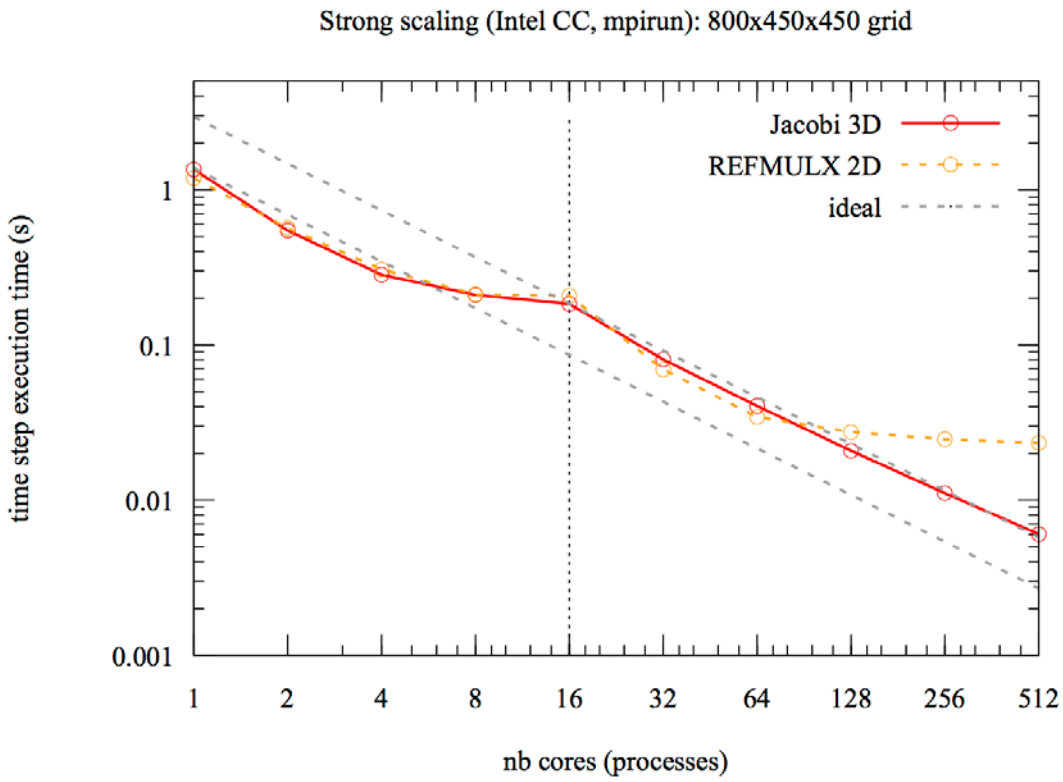


Fig. 104 Strong scaling of the three-dimensional Poisson solver using the Jacobi iteration method, employing a three-dimensional domain decomposition, measured on the TOK-P cluster (solid red). Re-scaled strong scaling of the parallel version of the code REFMULX, developed during the project HLST-REFMUL2P, measured on the HELIOS supercomputer (dashed yellow). Note that the underlying architecture of both machines is similar, namely, compute nodes with two Intel Xeon SandyBridge CPUs, which makes the comparison meaningful.

9.5. *Domain decomposition of REFMUL3*

Here we implement the domain decomposition parallelisation used in the Jacobi iteration test-code of the previous section in the node-level optimised threaded version of REFMUL3 (sub-section 9.3.1). As expected, the task revealed itself to be considerably more involved, not only because REFMUL3's source code has many more lines of code than the simpler test-code, but also because it involves staggered grids and several functions that are localised in parts of the domain, like antennas and wave guides. In the next paragraphs we describe the major points of the employed parallelisation concept, namely, the memory distribution and the inter-task communication.

9.5.1. **Sub-domain bounds and sizes**

Instead of allocating the full computational domain, the task-parallel version of REFMUL3 divides it into sub-domains, each belonging to a different MPI task that locally allocates only the corresponding memory. An exception to this rule is made for quantities with lower dimensionality than three, as well as for three-dimensional quantities with very different sizes, typically smaller, compared to the main variables of the code, which are the fields and currents. These quantities are not decomposed,

but simply replicated for each task, for two main reasons. First, the extra memory cost of having this global data replicated is hardly relevant, especially in comparison with the original serial code, where all the data was by definition global. Second, and more importantly, doing otherwise would impair the flexibility in the number of sub-domains that could be used for any given problem size.

For all quantities whose domain is decomposed, the concept of first and last grid-node global index is introduced. As the name suggests, these specify the global index values of the first and last grid-nodes of each sub-domain in each dimension (*x,y,z*). They are calculated using the global domain size and the number of MPI ranks used, both input parameters, together with the MPI rank of the task to which the sub-domain belongs. Each sub-domain contains one extra ghost-cell per face in all three dimensions, where the values of the corresponding faces of the neighbouring sub-domains are stored. This is a necessary and sufficient condition to enforce continuity of the distributed solution across the sub-domains for the algorithm under consideration. This rule is relaxed for the sub-domains which include faces of the global domain boundaries, where the neighbours are replaced by boundary conditions, and therefore no ghost-cells are needed. This implies that no communication is required at those locations. Note that making this explicit distinction between internal and external sub-domain faces has the advantage that index-values of the sub-domains remain exactly the same as the corresponding ones in the original global domain. In other words, the grid-node with coordinates or index-values (x_i,y_j,z_k) in the original global domain, has the exact same ones in the sub-domain which contains that grid-node.

In practice, for each domain dimension, the formula for the first grid-node index of the sub-domain on MPI task MyRank belonging to the global grid of size NSize decomposed over NRanks is

$$\text{First} = (\text{NSize}*\text{MyRank})/\text{NRanks} + (\text{BndyF}–w) \qquad (4)$$

where '/' is an integer division operator and w=1 is the number of ghost-cells. BndyF vanishes for all sub-domains, except those containing the global domain origin in the dimension under consideration, in which case it is set to unity. This condition is verified on sub-domains for which MyRank=0 (in that dimension), and its purpose is to locally cancel w, since no ghost-cells are required there. In practice, this just yields First=0, which is what ensures that the index-values within a given sub-domain remain exactly the same as the corresponding ones in the original global domain, as explained before.

The last global grid-node index is given by using MyRank+1 in the same expression, but without the last term

$$\text{Last} = (\text{NSize}*(\text{MyRank}+1))/\text{NRanks} - \text{BndyL}. \qquad (5)$$

This ensures the required overlap of one real grid-node between neighbouring sub-domains. To see that, simply compare the expressions yield for First using MyRank+1 with Last using MyRank, or check the numerical examples given in Table 22. The exception is made at the end of the global domain, where there is no neighbour. For the corresponding sub-domains, which by the way have MyRank=NRanks-1 (in that dimension), the previous expression is simply replaced by

$$\text{Last} = \text{gridSize}-1 \quad \text{if} \quad \text{MyRank} = \text{NRanks}-1. \qquad (6)$$

Note that we used gridSize here instead NSize because they can differ. If a grid is staggered in the dimension under consideration, then gridSize=NSize-1, otherwise they are the same. There is one important consequence of using NSize in Eqs. (4)–(5) but gridSize in Eq. (6). Namely, the sub-domain bounds yield for each MPI rank are the same for all grids, irregardless of their relative staggered-ness, except at the end of the global domain, where MyRank=NRanks-1. There the last grid-node is yielded by Eq. (6), corresponding to the actual last grid-node of the global domain,

which depends on whether the grid is staggered or not. The best way to understand this subtle difference is to take a look at the following example which illustrates the method.

Table 22 shows a simple one dimensional case to illustrate the method with NSize=10 decomposed over three MPI tasks. Two grids are considered. One with gridSize=NSize grid-nodes and another one staggered with gridSize=NSize-1 grid-nodes. We can see that the sizes of the sub-domains vary depending on the task rank, simply because the total grid-count is not a multiple of the number of tasks chosen. However, for each MPI rank, except the highest one, they remain the same for both grids, because so do the sub-domain bounds. In any case, it should also be noted that the figures yielded for the sizes of all sub-domains are similar, which ensures a very good load balance between MPI tasks.

		global domain	**sub-domain**		
		Serial	rank 0	rank 1	rank 2
grid 1	gridSize	10	4	5	5
	First:Last	0:9	0:3	2:6	5:9
grid 2	gridSize	9	4	5	4
	First:Last	0:8	0:3	2:6	5:8

Table 22 Illustration of one-dimensional domain indices using a simple example with NSize=10 and two staggered grids with sizes NSize and NSize-1, respectively, distributed over three MPI tasks, including ghost-cells.

With the information about the sub-domain bounds at hand, the size of each sub-domain in each dimension, including the extra ghost-cells, can be directly computed by each MPI task via the expression

$$\text{NSizePar} = \text{Last}–\text{First}+1. \tag{7}$$

Applying this procedure to each of the domain's three dimensions and for all of REFMUL3's domain decomposed variables yields the local memory size to be allocated independently by each MPI task, namely,

$$\text{NSizePar3D} = \text{NSizePar\_i}*\text{NSizePar\_j}*\text{NSizePar\_k} \tag{8}$$

where the i, j and k subscripts refer to the coordinates in the *x*, *y* and *z* dimensions, respectively. As we have exemplified before, this technique is quite flexible as it allows the sub-domain sizes to differ between MPI tasks, lifting the common constraint that the grid-count must be a multiple of the number of tasks used.

9.5.2. **Local memory allocation and access: global vs. local array indices**

The concept introduced with Eqs. (4)–(5) describes the construction of the sub-domains based on specifying their first and last grid-node indices in each direction in terms of the original global indices. Now it is time to realise the corresponding implementation in terms of memory allocation and access. The missing ingredient is how to establish the map between local array index values for each sub-domain and the corresponding global array index values on the global domain, which shall be used to access the data stored in memory. Let’s start with the local sub-domain memory allocation, which is done straightforwardly. We simply allocate the multi-dimensional arrays in a linear fashion using a C-pointer, with the length yielded by multiplying the grid-counts in each dimension (including the ghost-cells). Invoking a bi-dimensional example for simplicity, we have

```
double *f = malloc( NSizePar2D*sizeof(*f) );
```

where the sub-domain size was calculated in the same fashion as shown in Eq. (8), namely,

$$NSizePar2D = NSizePar\_i*NSizePar\_j.$$

The second step is to provide the mask to access this locally allocated memory using the global index values in each dimension. This is done with a C macro that makes this translation mask on-the-fly. For the same simplified two-dimensional domain decomposed over both dimensions, the following snippet illustrates how such macro looks like, using a similar notation to subsec. 9.5.1:

```
#define f(F,j,i)
 ( F.f[((j)-(F.First_j))*(F.NSizePar_i)+((i)-(F.First_i))] )
```

where where F mimics the structure type used in REFMUL3 to store the electromagnetic fields, here simplified to a bi-dimensional case. It contains the field values on the sub-domain (F.f), the corresponding indices for the first (F.First_*) and last (F.Last_*) grid nodes, as well as the sub-domain size (F.NSizePar_*), in all spatial dimensions. These are calculated according to Eqs. (4), (5) and (7). Lastly, the j and i represent the grid-node indices in the *x* and *y* dimensions, respectively. The last one (i) corresponds to the fast-index, since the code is written in C, which uses the row-major memory ordering. Hence, to access for instance the grid-node value of the E_{xy} component of the electric field at the position specified by the global index values (j,i), one simply uses f(Exy,j,i), provided that the appropriate conditions on the local memory bounds are fulfilled, namely

$$j \in [Exy.First\_j,\ Exy.Last\_j]\ \&\&\ i \in [\ Exy.First\_i,\ Exy.Last\_i].$$

These essentially yield a method to find which sub-domains contain the desired grid-nodes, since only those ones will fulfil that condition. For the remaining sub-domains, the access is simple suppressed, to prevent access to spurious memory data, or even segmentation faults. The generalisation to three dimensions, used in REFMUL3 is straightforward.

A consequence of the memory distribution concept described here, is that the local sub-domain memory access is greatly simplified, since the same global indices of the global domain in the serial code are used in the domain decomposed version. On the one hand, the bounds for the for-loops, which originally spanned over the whole global domain, now simply need to be replaced to span instead from the first to the last sub-domain grid-nodes (4) and (5), which by definition automatically take the correct global index values on each MPI task. On the other hand, dealing with spatially localised quantities, like antennas and wave guides, which therefore belong to a sub-set of the sub-domains only, also becomes much simpler. Indeed, a bug which was introduced while parallelising one of the antenna functions, and that proved to be particularly time consuming to fix, would have certainly been even harder to find without the global-local index map explained above.

9.5.3. **MPI topology and communication directives**

From the previous sub-section we know how to decompose each direction of the global domain over a given number of MPI tasks into sub-domains and handle the corresponding local memory. What is still missing is how to, in practice, effectively choose how many MPI tasks are attributed to each of the three spatial dimensions, as well as, perform the ghost-cell exchanges at every iteration to ensure spatial continuity across the sub-domains of the quantities calculated numerically. The answers are covered in the next lines.

The answer to the first question is MPI_Dims_create, which is used to automatically find a good distribution of the resources over each of the three dimensions of the domain. A Cartesian virtual topology is then created by invoking MPI_Cart_create and the process ranks (MPI tasks) are mapped to a three-dimensional coordinate

system using MPI_Cart_coords. This considerably facilitates finding out the MPI ranks of the neighbouring sub-domains in all three directions since we can now use MPI_Cart_shift to address this problem. These ranks are needed for the ghost-cell exchanges at every iteration, and this partially answers the second question concerning the communication details. The rest of the answer is to use calls to the asynchronous point-to-point communication directives MPI_Isend and MPI_Irecv, with MPI_Waitall invoked afterwards to ensure that the message exchange is completed before the ghost-cells' data is needed in a subsequent point in the code. In practice, these directives (send/recv/wait) are currently called together inside a single function, meaning that the advantages of an overlap between communication and computation are not yet exploited in REFMUL3. Although we expect a relatively easy implementation of it within the existing code infrastructure, this optimisation route is left for the future due to project time constraints. Besides, as shall be seen in the next sub-section, the scaling results are quite good already, comfortably fulfilling the main project goal. The MPI communication directives are however implemented already in a way to be executed on demand and separately for each dimension because each kernel function (calc<*>Field) requires populating the ghost-cells for one or two dimensions, but never for all three. Moreover, since these synchronisations are never symmetric but rather uni-directional, an explicit split is made between communication in the positive and negative direction of each dimension. For the kernels which require two-dimensional communication (calcJ<*>Field), the calls are made sequentially for each dimension, such that only when the communication over the first dimension is complete will the one over the second dimension start, the order not being important. This ensures that the edges and vertices of the boundary faces are correctly included in the exchanged data as well.

9.5.4. **Tests and scaling results**

The correctness of new MPI parallelised version of REFMUL3 (including the hybrid MPI/OpenMP version) is verified by checking against the original code's results. This is illustrated in Fig. 105. It shows a time trace of a physically relevant quantity, the E_z electric field component, for different domain decomposition scenarios. As expected, the results do not change with the number of MPI tasks used.

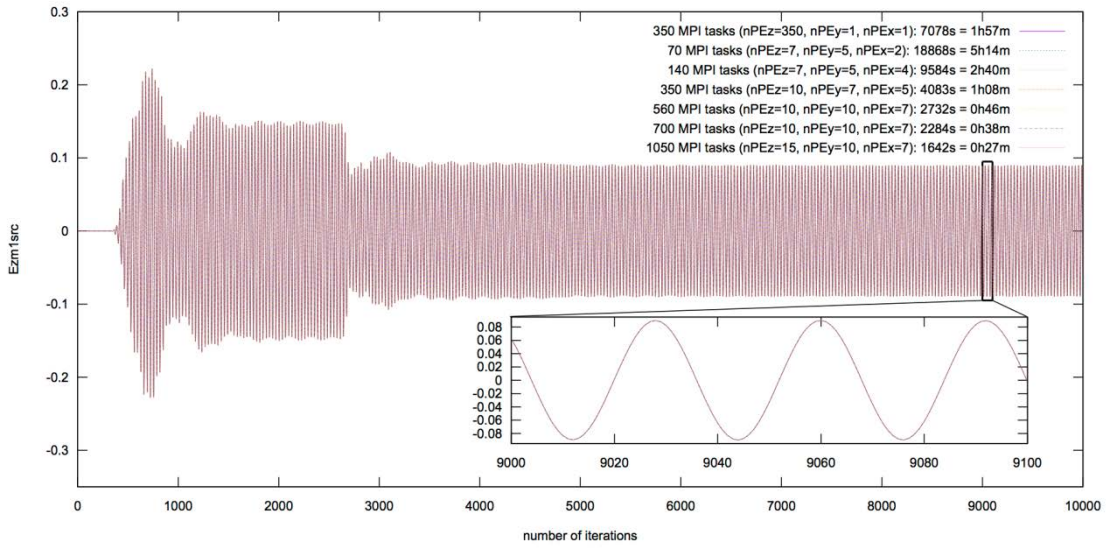


Fig. 105 Time traces for the E_z electric field component measured at a single position of the global domain for different numbers of MPI tasks used. All the curves sit on top of each other, demonstrating the invariance of the results with respect to the number MPI tasks used.

The next step is the assessment of the parallel scaling performance of the final algorithm in its different flavours (OpenMP, MPI, hybrid). Fig. 106 shows the strong scaling behaviour of the new parallelised REFMUL3 code. It summarises the main results obtained during the project by plotting the total simulation time divided by the

number of iterations for the different versions of the code, as a function of the amount of resources (cores) used. The horizontal dashed and dot-dashed dark-blue lines on the top correspond to the original and optimised serial versions of the code (Sec.9.2.2). These are included for reference. Obviously they were computed on a single core, but those data points are extended horizontally for easier visualisation.

The light violet curve corresponds to the pure OpenMP version of the code, which can only be used inside a single compute node, indicated by the vertical dotted line passing through 36 cores. The case shown distributes the threads with compact affinity to the cores, and shows the typical behaviour for a memory bound algorithm on a two CPU ccNUMA architecture. Namely, a saturation of the scaling as the cores of the first CPU are being filled, which is then lifted once the cores on the second CPU start to be used. If the curve corresponding to a scatter affinity of the OpenMP threads were to be shown, not done here simply to avoid cluttering too much the graph, a scaling behaviour closer to the ones of the pure MPI curves (green and blue) within one node would be shown. The reason for this is that, for all pure MPI cases, a bunch affinity [5]–[6] has been invoked. This ensures that both CPUs are used for all numbers (>1) of cores used.

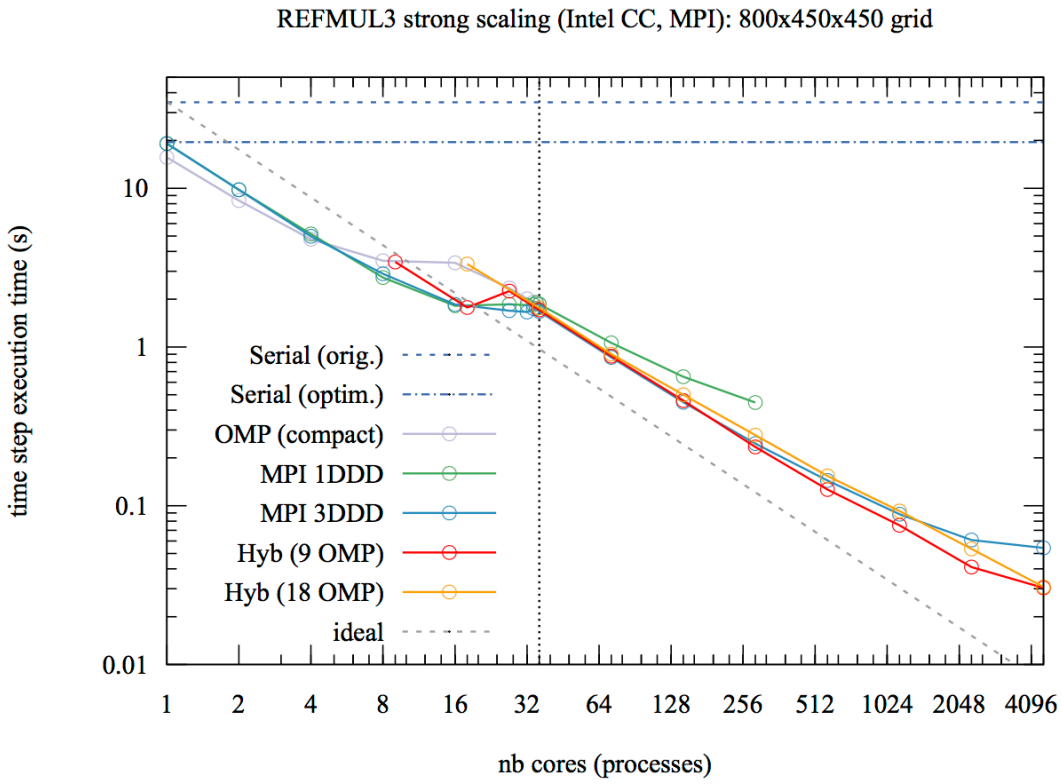


Fig. 106 Strong scaling of the three-dimensional FDTD REFMUL3 code, in its different flavours on a 800x450x450 (*x*,*y*,*z*)-grid: original (dashed dark-blue), serial single-core optimised (dot-dashed dark-blue), pure OpenMP with compact thread affinity (light violet), pure MPI with one-dimensional domain decomposition (green) and three-dimensional domain decomposition (blue) and hybrid OpenMP/MPI (with three-dimensional domain decomposition) with 9 (red) and 18 (yellow) OpenMP threads per MPI domain. The measurements were made on the Broadwell nodes of Marconi, and the total execution time divided by the number of iterations is plotted.

Still within a single compute node, there are also a few data points corresponding to the hybrid version of REFMUL3. This version uses MPI tasks that form domains of OpenMP threads. Two configurations using 9 and 18 OpenMP threads per MPI domain, respectively, are shown in red and yellow. Both cases use a bunch affinity for the MPI domains (tasks), so as to map them consecutively as closely as possible, while still ensuring a balanced distribution over both CPUs [6]. The threads within each MPI domain are mapped compactly. This explains why the first data point in

both curves lies on top of the pure OpenMP compact affinity curve, since when only one MPI domain is used, this configuration yields the exact same affinity. As soon as more than one MPI domain is used, these domains will be distributed evenly across both CPUs. Furthermore, the apparent loss in performance in the third data point for the yellow curve, can also be explained by this, since it corresponds to an odd number of MPI domains, namely three. This obviously creates an imbalance when distributed over two CPUs.

Going beyond one compute node, only the MPI and hybrid curves occur, since only the corresponding code versions include a distributed memory infrastructure. The green curve refers to the pure MPI code, distributing the domain over a single dimension, namely the slowest varying index in memory (*z*-dimension). Within one compute node there is little difference between this case and the one with a full three-dimensional domain decomposition (blue curve). As the total number of cores increases, the difference becomes more pronounced, since the former starts to degrade. This is the expected behaviour, if we recall the argumentation presented in Subsec.9.3.2 about the ratio between data communication and the computation cost that led to Eqs.(1)–(3). Moreover, the other limitation of a one-dimensional decomposition is also apparent in the plot. Even discarding the signs of an approaching saturation, the size of 450 grid-nodes in the *z*-dimension simply dictates the maximum number of MPI tasks that can be used. On the contrary, the full three-dimensional domain decomposed code, either pure MPI, or hybrid MPI/OpenMP, is able to use much more resources, since the theoretical maximum number of tasks allowed is given by the total three-dimensional grid-count. The scaling is also much better due to the lower amount of communication per sub-domain in relative terms, as explained before. For the problem size under consideration, it behaves quite well up to 2304 cores. Looking at it in detail, it seems that the hybrid version using MPI domains with 9 threads offers the best performance. A more detailed investigation on the reasons as to why this is the case, as well as to why the pure OpenMP version of the code seems to be slightly more efficient than the pure MPI counterpart within one node, needs to be postponed to the future due to the limited time budget of the current project. In any case, the main goal of the project was met, namely, to achieve a parallel three-dimensional FDTD code with very good scaling properties, as can be seen by comparing the blue, red and yellow curves with the grey dashed line in the plot. The latter indicates the ideal scaling starting from the cost of the original serial code, providing a visual indication of how the actual scaling of REFMUL3 deviates from the theoretical optimum. It further shows that the three of orders magnitude speedup aimed for at the beginning of the project was met.

For completeness, since the case with a 720x700x700 grid was mentioned in Fig. 102 of Subsec.9.3.1, we also run it using the latest hybrid version of REFMUL3. On 128 Broadwell Marconi nodes, each using four MPI tasks with 9 OpenMP threads, for a total of 4608 cores, the case took 5 minutes to complete (without I/O operations). Before (pure OpenMP version of the code), 7 hours were needed to perform the same task on a single node (36 cores). If we further compare the hybrid compute time with the one for the original serial code, namely 132 hours, we see once more that the goal of a three orders of magnitude speedup (1584x) is comfortably met for the slightly bigger case. Moreover, for even bigger grids, which could be distributed over larger numbers of resources, higher speedup factors are expected. However, one has to bear in mind that those would no longer fit in the memory of a single typical HPC compute node, like the ones on Marconi, making a direct speedup calculation (to the serial code) impossible.

9.6. ***Parallel I/O***

As foreseen in the project proposal, the results obtained so far confirm that the task of writing the output results of a production run to disk becomes a bottleneck. This implies that parallel I/O techniques must be invoked. One simple solution is devised here and due to its simplicity, we were even able to implement it neraring the end of

the project. It builds on top of the preexisting serial I/O infrastructure of REFMUL3, which uses HDF5 [2] to store the simulation output data and a descriptor file in XDMF format [3] to allow the former to be opened directly using ParaView [4] for data visualisation. The new parallel I/O solution devised has each MPI task writing an independent HDF5 file to disk with the requested data from its spatial sub-domain. Only a single XDMF descriptor file is saved by the master MPI task. It contains a collection of all descriptors, one for each sub-domain. This is possible since the only information contained in each descriptor are the coordinates of the first grid-node and the size of their data-set. This can be easily calculated by the master rank for all the participating MPI ranks using Eqs. (4)–(5). A new function was written to do so for any given MPI rank. This is the exact same operation performed when the memory for each sub-domain is allocated. As for the function writing/reading HDF5 files, the only changes made are the size of the data-set, which now corresponds to the sub-domain, not the original global domain, and the filename, which now includes the information about the task rank in each direction (recall the usage of the MPI_Dims_create directive, as explained in Subsec. 9.5.3). Fig. 107 illustrates the decomposition of the domain over eight MPI tasks in terms of the HDF5 files outputted, where to each sub-domain corresponds a different colour.

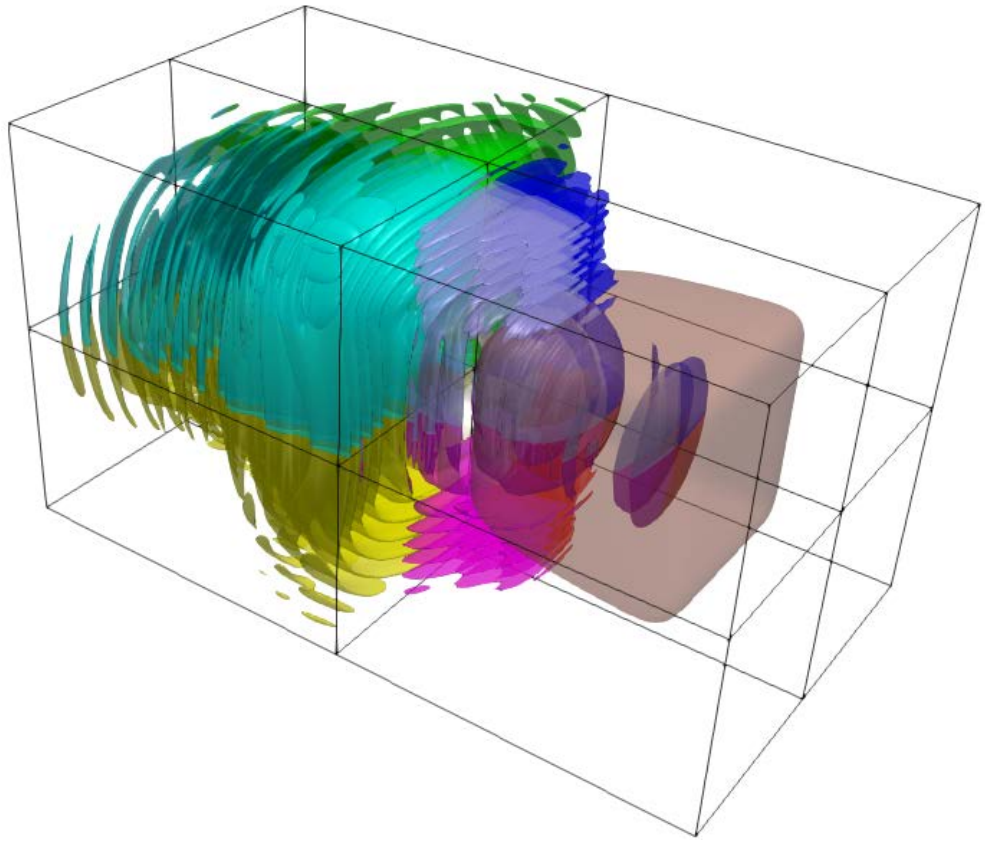


Fig. 107 Illustration of the three-dimensional REFMUL3 800x450x450 domain decomposition over eight MPI tasks. Each sub-domain is represented with a different colour.

The described solution fulfils its purpose, in the sense that it allows performing I/O operations on domains which do not fit memory-wise in a single compute node. This was not possible in the original serial version. Even though the current I/O implementation yields, in theory, the best possible I/O performance, provided that the number of MPI tasks is not too large (below a couple of thousands), it should not be regarded as the final I/O concept. Firstly, as already mentioned, this solution breaks down for sufficiently large numbers of MPI tasks, as the file system can potentially start to face difficulties due to the large numbers of HDF5 files (one per MPI rank) being accessed simultaneously. Secondly, a proper performance assessment is still lacking, ideally comparing it with alternative solutions which might prove more flexible from the point of view of the project coordinator, e.g., using the native parallel HDF5

library capabilities or simply MPI I/O. Finally, there is currently no restart infrastructure in the code, and this should be eventually added. Due to lack of time in the current project, and also because they fall beyond its scope, these tasks shall be postponed for now. A strong possibility to consider is to address this point through a new HLST project, to be submitted in the future. Meanwhile, the recommended best practice in terms of I/O operations for the current implementation is to run the hybrid version of the code. If large numbers of resources are needed, one is able to decrease the total number of MPI ranks (and therefore HDF5 files) and increase the number of OpenMP threads accordingly. The extreme case of which would be to use one MPI task per compute node with one OpenMP thread per core.

9.7. *Visit from the Project Coordinator to Garching*

Filipe Silva visited the IPP for a week and a half (21–30 November 2016) in the framework of the HLST-REFMUL3 project to get acquainted with its current status. Not only could he get familiarized with the strategy chosen for the domain decomposition of his code, but he could also be directly involved in the process of implementing the necessary changes in the source code of several functions, as they were being adapted at the time. In that process, a couple of bugs were found and corrected in the original code, and a new smaller but still relevant reference test-case was devised as well. This was requested from the HLST side, and greatly facilitated all subsequent code changes, since the new test-case was small enough to be easily run interactively. Finally, the visit was also very useful in that it allowed both parties to discuss and decide on important practical implementation details that were still open at the time, as well as to merge the HLST and the project coordinator's (PC) branches of the code, that had meanwhile diverged.

9.7.1. **Visit to the Project Coordinator in Lisbon**

Tiago Ribeiro visited IPFN-IST in Lisbon for a week and a half (30 January – 8 February) at the end of the HLST-REFMUL3 project. The main goals were to finish the implementation of the last steps of the parallel communication in the parallelised REFMUL3 code and to give its source code to the project coordinator. Several hands-on sessions were conducted together with Filipe da Silva, where we inspected the source code of every function, highlighting in detail the differences to the original serial code and why they had to be made. This was fundamental to help the project coordinator getting acquainted with the parallel version of his code, a necessary condition for future developments in REFMUL3.

9.8. *Summary*

The project finished successfully fulfilling the major goals of the proposal. The single-core code profiling and optimisation were the first steps taken, from which an overall 1.8x speedup factor for the reference test-case, used throughout the project, resulted. OpenMP threads were then implemented to exploit the parallelism at the node-level, with a 18.6x speedup obtained with 36 threads on a Marconi Broadwell compute node. The bulk of the work was the implementation of a three-dimensional MPI domain decomposition and corresponding communication. A domain decomposition concept was defined, and initially implemented in a simpler standalone test case outside REFMUL3, namely, a Jacobi iteration Poisson solver, which has similar numerical properties to REFMUL3. The purpose was to check the concept's feasibility and adequacy before attempting its deployment in REFMUL3, which implies substantial changes to the whole code, and therefore a major coding effort. Indeed, the very good scaling properties that the parallel Jacobi iteration solver yielded, gave confidence on the parallelisation strategy chosen for REFMUL3.

The deployment of the parallelisation concept in REFMUL3 proceeded as the major task of the project, including the infrastructure for the domain decomposition, the MPI initialisation and the corresponding communication steps. After a time consuming code development phase, followed by a debugging phase, the parallel REFMUL3

code was finally finished. It has good scaling properties and, compared with the original serial code, runs the reference test-case given by the project coordinator three orders of magnitude faster, if enough resources are used. Additionally, within the available project time budget, an effort was made to implement a working parallel I/O technique, with minimal changes to the original serial counterpart. For the bigger domains that are now computationally available, especially if they don't fit within the memory of a single compute node, no straightforward serial I/O technique based on data gathering can simply work. The devised parallel solution makes this possible by having one HDF5 file written per MPI task.

In sum, the main goals of the project could be successfully achieved. They allow novel three-dimensional FDTD simulations with realistic problem sizes, which was not possible before.

9.9. *Outlook*

There are two topics that stand out, namely, overlapping communication with computation and putting more effort in parallel I/O techniques. The former should, at least in theory, further improve the parallel scalability of the code, especially when higher number of resources are used. It should be relatively straight forward to implement in REFMUL3, as the existing code already includes all the necessary ingredients. The latter topic is the natural next step, since I/O is currently the most pressing issue, especially for bigger production runs. The reasons aren't just about the ability to read/write enough data fast enough from/to disk, but also to facilitate the input of localised functions in the domain, like antennas and wave guides. These could be simply read from pre-produced files outside REFMUL3, rather than being hard-coded inside internal parallelised functions that run during the initialization phase of a simulation, like it is currently done. This would make changing these types of input, which is a common practice for physics studies, much easier. The parallel I/O technique implemented so far is working and seems to have good potential performance-wise. However, no effort was put into assessing the current I/O performance, nor in investigating other techniques (MPI I/O or parallel HDF5). These are desirable milestones for the future of REFMUL3, including a restart infrastructure in the code, which is so far absent. However, since the project time budget was exhausted, the above proposed tasks must remain as recommendations for future work, potentially as part of a future HLST project on REFMUL3.

9.10. *References*

[1] K.S. Yee, *IEEE Trans. on Antennas and Propagation* **14** (1966) 302.

[2] The HDF Group, *Hierarchical data format version 5* (2000-2017) http://www.hdfgroup.org/HDF5

[3] *eXtensible Data Model and Format* (2000-2017) http://www.xdmf.org/

[4] Sandia National Labs, Kitware Inc and Los Alamos National Labs, *Paraview: Parallel visualization application* (2000-2017) http://paraview.org

[5] Intel MPI Library Developer Reference, *Environment Variables for Process Pinning* https://software.intel.com/en-us/node/528818

[6] Intel MPI Library Developer Reference, *Interoperability with OpenMP API*

https://software.intel.com/en-us/node/528819

10. Final report on HLST project VIRIATO2

10.1. *Introduction*

This project aims at improving the scalability of the *VIRIATO* code, which uses a unique framework to solve a reduced (4D, instead of the usual 5D) version of gyrokinetics applicable to strongly magnetised plasmas. VIRIATO is parallelized with MPI using domain decomposition over two directions in the configuration space domain. It is pseudo-spectral perpendicular to the magnetic field, and employs a high order upwind scheme for discretizing the equations along the magnetic field. The time integration is done via an iterative predictor-corrector scheme. A distinguishing feature of the code is its use of a spectral representation (Hermite polynomials) to handle the velocity space dependence. This converts a drift-kinetic equation for the electrons into a coupled set of fluid-like equations for each of the coefficients of the Hermite polynomials. In terms of numerical accuracy, this is rather advantageous: spectral representations are more powerful than grid-based ones, and this framework enables the deployment of the standard tools of CFD to deal with what would otherwise be a difficult kinetic equation. On the other hand, the scalability of the resulting algorithm, which extensively uses bi-dimensional Fourier transforms, becomes a non-trivial problem, whose solution is the main goal of this project.

10.2. *Bi-dimensional Fourier transforms and data transposition*

VIRIATO is pseudo-spectral in the *xy*-plane, which is perpendicular to the magnetic field, and uses a spectral Hermite polynomial representation for the velocity space (*ng*-coordinate). It employs the package FFTW v2.1.5 to perform the corresponding fast Fourier transforms (FFTs). The domain is parallelized in two directions of the configuration space, namely the *y*- and *z*-directions. Therefore, the bi-dimensional FFTs must involve the transposition of data across a sub-set of the participating MPI tasks (cores) in an all-to-all fashion. This operation poses a challenge when it comes to scalability on a large number of cores in the perpendicular plane (npe), the reason being the complexity of the underlying communication. The number of MPI messages required grows with $O(\mathrm{npe})^2$, as clearly shown in Fig. 104. There, the all-to-all communication patterns involved in transposing the same bi-dimensional dataset distributed over four or eight cores are illustrated. Note that besides the global all-to-all data exchange illustrated in the figure, each message, once received, further needs to be locally transposed in order to obtain the final transposed data-set.

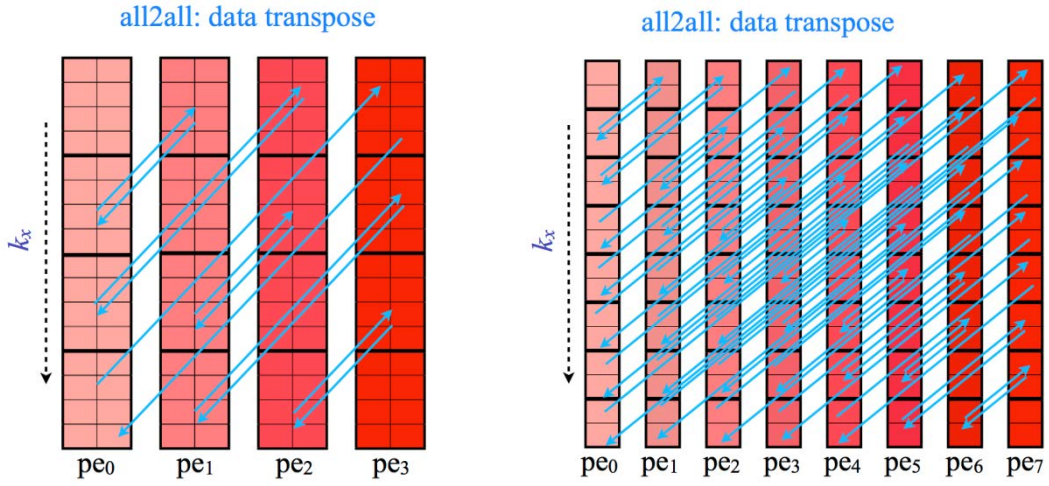


Fig. 108 Illustration of the all-to-all communication pattern for the parallel transposition of a bi-dimensional dataset parallelized over four (left) and eight (right) cores.

10.3. *VIRIAT0 transpose optimisation: previous results*

The algorithm which was originally used for the bi-dimensional Fourier transforms in VIRIAT0 has been assessed in detail during the previous HLST-VIRIATO (2015) project. This revealed that the algorithm was prone to suffer from network communication latency accumulation. The culprit was the calculation of the parallel transposes that was made sequentially on each *xy*-plane for every grid-node in the remaining *z* and *ng* dimensions of the sub-domains. While this maximised flexibility in the sense that the same single subroutine could be called for all Fourier transforms, regardless of the dimensionality of the quantity it acts upon, it also significantly impaired the scalability of the algorithm for larger number of resources. Ultimately, this created an important scaling bottleneck in VIRIAT0 and to understand why, we need to recall Fig. 104 and the corresponding discussion.

We have seen that, as the number of cores used in the perpendicular plane (npe) is doubled, the size of each MPI message to be exchanged between them is reduced by a factor of four, but at the same time, their total number increases by the same factor of four. This means, as expected, that the total amount of data to be exchanged between the resources remains constant, only their granularity changes. In an ideal network system, this would imply that the cost of the transpose would be independent of the number of cores involved. However, in the real world, every message exchange via the network incurs a fixed cost independent of the message size, which can conceptually be defined as the (finite) time of exchanging a zero-sized message. This is called latency. As the message size decreases, it starts to dominate the communication process.

In our case, the impact of latency, for large enough npe, is that the transpose algorithm will be negatively affected, independent of its quality. Therefore, sequentially repeating the perpendicular *xy*-plane transpose for each of the grid-nodes in the remaining dimensions just makes the problem worse by increasing this cost linearly. To avoid this accumulation, the bi-dimensional parallel transpose algorithm was modified to communicate all the data in one (*z*) or both (*z* and *ng*) of the remaining dimensions in a single operation. The corresponding results are shown in Fig. 109. They refer to the calculation of a sequence of bi-dimensional FFTs both in forward and backward directions on a 4D dataset. This allows comparing the final doubly transformed data-set with the original one, and therefore directly checks the correctness of the implementation. VIRIAT0's original algorithm is represented in red and the extended algorithm with the aggregated transpose including data (i) over the *z*-dimension and (ii) over both the *z*- and *ng*-dimensions are represented in light-blue and yellow, respectively.

A detailed discussion of the scaling results can be found in the final report of the HLST-VIRIATO (2015) project. Here it shall suffice to highlight some features that motivate the subject of the remaining sub-sections. First, we note that the extended transpose algorithms are always better than the original, even though they all start to degrade for high enough numbers of resources employed. This is the expected behaviour based on the increase in complexity of the parallel transpose algorithm with the number of cores (npe) in the *y*-direction (recall Fig. 104). That the degradation is much more pronounced for the red curve is also clear, due to the latency accumulation problem discussed in the previous paragraphs. But why does the light-blue curve follow the red curve so closely, when we specifically addressed this issue by aggregating data in the *z*-direction? Because beyond 1024 cores, indicated by the dashed vertical line, the amount of resources over which the *z*-dimension is distributed is fixed to npez=128, with the remaining available resources corresponding to npe. For the domain size under consideration (256^3 configuration space with 16 velocity space moments), this yields two *z* grid-nodes per sub-domain, so the aggregation done in this dimension during the transpose amounts to just two *xy*-planes. In other words, we only improve the latency penalty by a factor of two compared to the original transpose algorithm. Conversely, the transpose used for the

yellow curve additionally uses data aggregation in the moment direction (*ng*), which is not distributed. Therefore, the reduction in the latency accumulation becomes much more pronounced in this case (in theory by a factor of 32 compared to the original method).

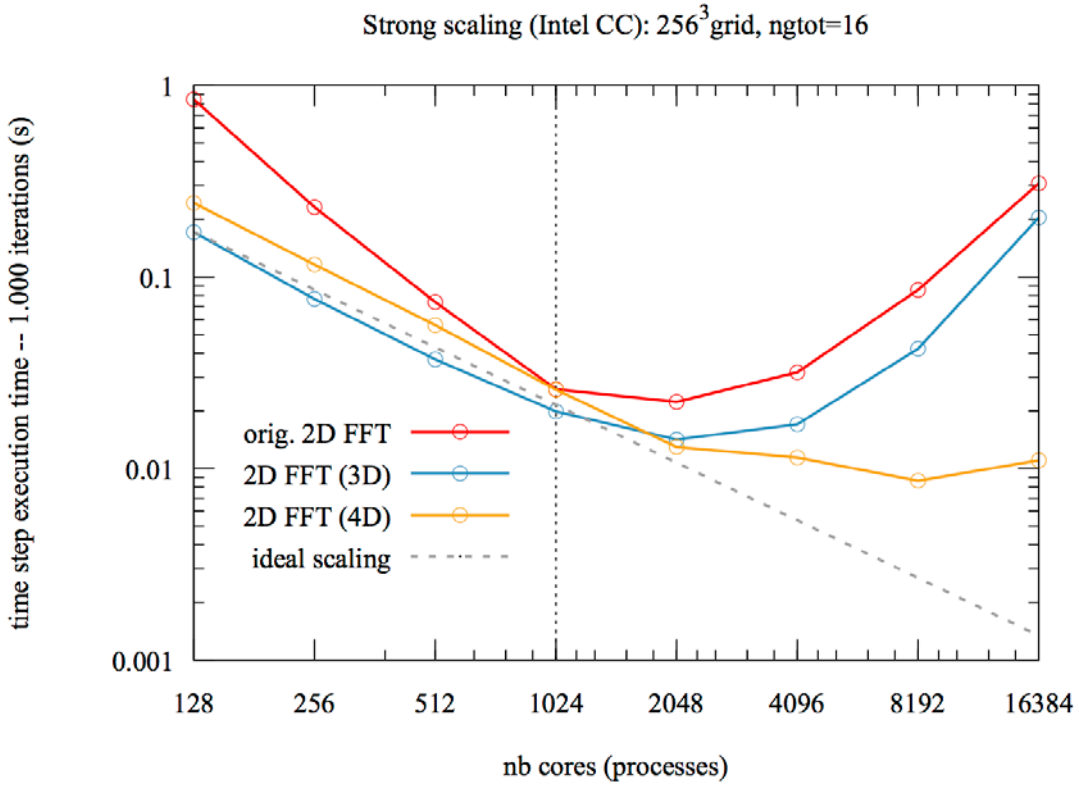


Fig. 109 Strong scaling of the parallel bi-dimensional FFT algorithm on a typical medium-sized domain of `VIRIATO`: configuration space grid-count of 256^3 with 16 velocity moments. A sequence of bi-dimensional FFTs both in forward and backward directions is executed 1000 times and the average elapsed time is considered. The `VIRIATO`'s original algorithm is shown in red. The extended algorithm with the data aggregated over the *z*-dimension and over the *z*- and *ng*-dimensions are shown in light-blue and yellow, respectively. Results obtained on the former HELIOS-CSC computer in Japan using the BullX MPI library.

Last, and most important for what's coming, the results presented here refer to the bi-dimensional algorithm only acting on a four-dimensional data-set. This is an idealised situation. Unsurprisingly, when the best performing transpose algorithm, which uses data aggregation in the *z*- and *ng*-directions, was integrated in the main `VIRIATO` code, the whole code yielded a scaling behaviour that was much more modest than the corresponding yellow curve in Fig. 109. There are two main reasons for this. First, several quantities in the code are only three-dimensional, for which only the data aggregation in the *z*-direction is possible. Second, even for the four-dimensional quantities in the code, there are computationally cost-intensive regions where using additional data aggregation in the *ng*-direction during the transpose step is disallowed by data dependencies imposed by the numerical scheme. Therefore, in all these cases, the strong scaling of the new bi-dimensional FFTs necessarily follows the light-blue curve in Fig. 109 and not the yellow one. Their weight is large enough in `VIRIATO` to make the code's overall strong scaling be closer to the light-blue curve than to the yellow one. This raises the question: Can we do better than this?

10.4. ***New optimisation strategy***

So far we have improved the parallel scalability of the parallel FFT algorithm by reducing its latency accumulation. However, the result did not carry over to `VIRIATO` with the degree we had hoped for. We now attempt something else, namely, the

reduction of the overall complexity of the parallel transpose by reducing the amount of resources involved in the inherent all-to-all communication step. This can be done by adapting the bi-dimensional FFT algorithm at hand to exploit the NUMA hardware topology of modern HPC machines.

The main idea behind the concept of a NUMA-aware parallelization scheme stems from the fact that the topology of modern HPC facilities is not flat. The physical cores are grouped into processors/sockets, which in turn are grouped into compute-nodes that share memory. A typical HPC machine is a collection of several of these compute-nodes and, as a consequence, its memory bandwidth is non-uniform. As such, treating all the resources (cores) on equal footing, like it is the case for a flat-MPI parallelization, is not necessarily the most efficient solution. This is where a NUMA-aware parallelization concept comes in, as an attempt to better exploit the hardware resources. The same argument is even more relevant in accelerator architectures, e.g. the Intel many-integrated-core (MIC) processors, where an increasing number of cores share memory resources. There are two main possibilities to achieve a NUMA-aware parallelization and these are covered in the following.

10.4.1. MPI/OpenMP hybridisation

The first possible solution uses parallel OpenMP regions in the *y*-direction of the domain, which is already parallelized with MPI. Each MPI task in this direction then uses threads (up to `nthreads`=48 on Marconi's Skylake partition) to share the work within its local sub-domain. Since only the MPI tasks, and not the OpenMP threads, need to be involved in the MPI transpose communication, the number of messages can be significantly reduced, from $O(\texttt{npe})^2$ to $O(\texttt{npe/nthreads})^2$ (Ribeiro & Haefele, 2014). Naturally, the size of each message increases by the same factor of $O(\texttt{nthreads})^2$, in accordance to what was shown in Fig. 104. This is exactly the reduction in the complexity of the all-to-all communication involved in the bi-dimensional FFTs that we are aiming for.

However, there is a noteworthy drawback to this configuration. Once implemented, the MPI/OpenMP hybridisation concept has to be extended to the remaining computationally intensive parts of `VIRIATO` (besides the FFTs). Otherwise, at runtime, the extra resources allocated for the threads would be idling during those other calculations. The problem would become even more pressing as the time spent in the FFT computation is expected to decrease as a result of the optimisations proposed, yielding the remaining parts of the `VIRIATO` code necessarily more costly in relative terms. Further, using OpenMP in `VIRIATO` would also mean that thread-safety would have to be guaranteed by any invoked external library. Noteworthy is that there is no technical impediment in doing this other than the man-power needed to implement such changes in the available time frame.

10.4.2. Two-level MPI communication

In order to avoid the MPI/OpenMP drawbacks, but still take advantage of the node-level NUMA topology to reduce the transpose communication complexity, other schemes using pure MPI implementations can be devised. The general idea is to split the existing communicator(s) in the pure-MPI `VIRIATO` code into sub-communicators to yield different levels of communication that would better map to the hardware topology of a modern HPC machine. For the FFT algorithm in particular, this would mean splitting the `npe` communicator responsible for the decomposition over the *y*-direction into two kinds of sub-communicators. One kind to group tasks below the compute-node level into a set of disjoint sub-communicators (one sub-communicator per group of tasks) according to a given criterion (e.g. cores-per-node, cores-per-socket, etc.). The other kind to group the 0^{th}-rank task from each of the previously established sub-communicators into a single set (or sub-communicator).

In the remaining discussion, we denote the former kind as *intra-group communicators* and the latter kind as the *inter-group communicator* (as noted already, there will only be one of the latter). In this scenario, only a subset of the `npe` tasks would be involved in the "all-to-all" communication pattern, namely the tasks belonging to the inter-group communicator. The remaining resources would be responsible for a local transposition of the data, using resources contained inside a compute-node. The complexity of the all-to-all communication is reduced in the same fashion as before (refer to subsec. 10.4.1), namely, the amount of MPI messages involved is this time reduced from $O(\texttt{npe})^2$ to $O(\texttt{npe/npe\_intra-group})^2$, while their size increases by the same factor of $(\texttt{npintra-group})^2$. However, as mentioned before, a clear benefit of this model with respect to the MPI/OpenMP paradigm is that the total number of MPI tasks remains unchanged compared to the original flat-MPI `VIRIATO` code. Therefore, the same amount of resources (cores) would still be available for the floating point operations (FLOPs) in the rest of the code, through the original flat `MPI_COMM_WORLD` communicator, without any modification to its source-code.

There are two main possibilities for implementing the pure-MPI NUMA aware concept into the bi-dimensional FFT algorithm of `VIRIATO`. The differences between them lie in the choices made inside each of the intra-group communicators. Either (i) one invokes explicit intra-group data communication via calls to `MPI_Gather/Scatter`, or (ii) one allocates so-called shared memory windows. The former uses a master MPI task within each intra-group communicator with a local memory buffer large enough to store the exchanged messages and perform the local transpose operations after the inter-group communication takes place, as was already studied elsewhere (Ribeiro & Haefele, 2014). The latter uses instead shared memory windows as inter-group communication buffers within each intra-group communicator to make the data directly accessible to all the MPI tasks therein. As explained before, no changes to the rest of the code are required in either case, since `npe` is not modified (compared to the original flat-MPI domain decomposition). However, the latter method has one further potential advantage. Namely, the local transpose operations can be performed in parallel by all MPI tasks within each intra-group communicator, whereas in the former only one MPI task per intra-group communicator is involved in those. We decided to attempt the latter within this project, mostly to gain knowledge on an interesting new concept, which moreover has very good portability characteristics, since it is part of the newest MPI standard.

10.5. *Implementation of the shared memory windows*

The concept using shared memory windows is called pure-MPI hybridisation and is available since the introduction of the MPI-3 standard. It completely suppresses the need for explicit communication inside each group of resources sharing a memory window, which is by construction directly accessible by all of them in a concurrent manner. This comes at a price, however, which should be kept in mind. Namely, data consistency must be ensured across the shared memory window, which might introduce overhead compared to the purely local memory access of a flat-MPI implementation.

10.5.1. XOR transpose

The work on the subject of shared memory windows began already in the course of the original HLST-VIRIATO (2015) project. At the time, to prove the principle while minimising the time investment risk, the pure-MPI hybridisation has been attempted, not on `VIRIATO`'s transpose method, but rather on an alternative one, with which we were more familiar from previous projects – for more information refer to the HLST-NEMOFFT (2012) project. Such method, which we shall refer to as XOR transpose, uses a sequence of `MPI_Sendrecv` point-to-point directives dictated by a bit-wise exchange pattern computed using an exclusive-OR (XOR) logical operation between

the MPI ranks and an appropriate data-indexing within each sub-domain (Ribeiro & Haefele, 2014).

The preliminary performance results obtained then were promising, as can be seen in Fig. 110. The scaling curve yielded by the pure-MPI hybridised XOR transpose with data aggregation over the *z*-dimension are compared to the ones obtained previously for VIRIATO's transpose methods (Fig. 109). The case shown shares memory between all the cores inside each socket (solid green curve). A better scaling performance can be seen when compared to the blue curve, which corresponds to the same level of data aggregation. This in principle proves the devised hybridisation concept correct and motivates the work being presented in this report. A word of caution should be given here, however. Namely, the scaling results using shared memory windows seem to be highly dependent on both the architecture and the MPI library used, as well as the problem size considered. This shall become evident later on, when we discuss the newest results obtained on MARCONI. For the time being it is important to highlight that the results of Fig. 109 were obtained using the BullX MPI library that was available at the time on HELIOS. Finally, it should be reiterated that the reason to use data aggregation only over the parallel direction (*z*) for the new hybridised transpose methods is that this can be applied everywhere inside VIRIATO, unlike the case using additional moment *ng*-data aggregation, as we have discussed in Sec. 10.3.

Although the XOR transpose method is not easily deployable to VIRIATO for technical reasons related to the layout of the subdomain data distribution (refer to project HLST-VIRIATO (2015)), the scaling results herein are expected to generalise roughly to the hybridised VIRIATO transpose method with the same degree of data aggregation. Devising, implementing and testing such a hybridised version of VIRIATO's transpose are the main tasks of the current project. The encouraging results obtained with the pure-MPI hybridised XOR transpose are its chief motivation.

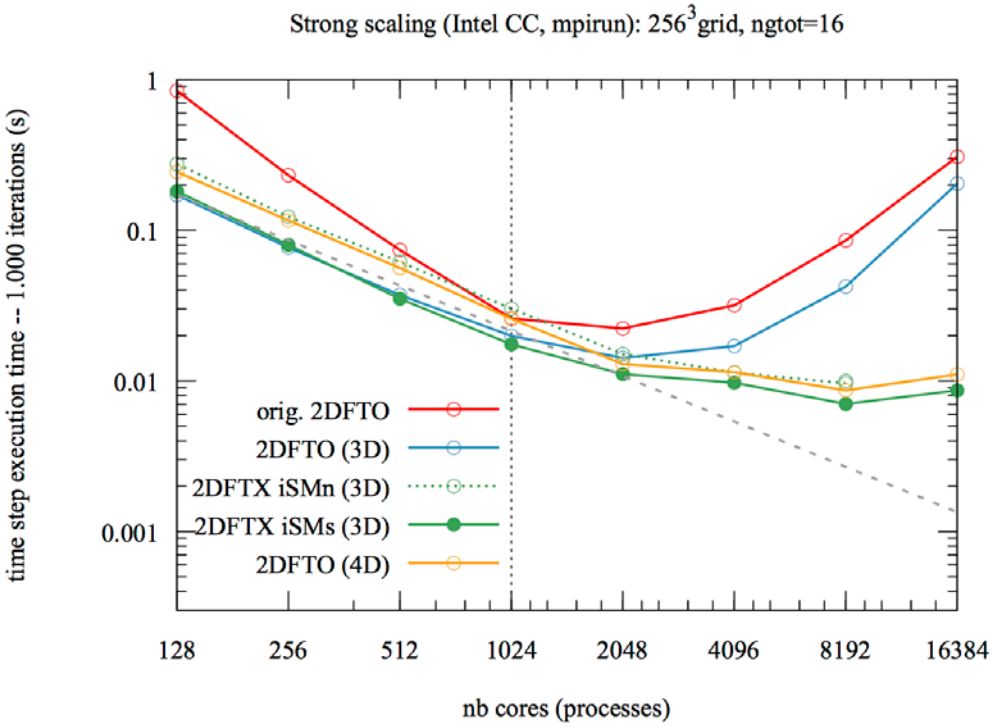


Fig. 110 Same as in Fig. 109 with the addition of curves for the XOR-transpose with data aggregation over the *z*-dimension using shared memory windows over the cores belonging to the same compute-node (dashed green) and compute-socket (solid green).

10.5.2. **VIRIATO's transpose**

Hybridising VIRIATO's transpose requires extensive redesigning of the algorithm. To begin with, it implies splitting the global communicator (MPI_COMM_WORLD) into one inter-group and several intra-group sub-communicators, as well as the allocation of two shared memory windows within each of the latter, for sending and receiving data, respectively. These memory windows serve as large communication buffers for the new inter-group all-to-all MPI communication. This replaces the original global (MPI_COMM_WORLD) all-to-all MPI communication and their (smaller) communication buffers that used to be local to each single MPI task. Finally, the local transpose operations required when storing the data from the shared memory windows into the original sub-domains after each MPI communication step become more involved, since they must be performed concurrently by all intra-group MPI tasks. These topics are covered in more detail in the remainder of this section.

The part of the original VIRIATO's transpose algorithm which was responsible for figuring out which blocks of sub-domain data are copied to the (former) local communication buffers prior to each communication step of the global all-to-all communication is kept. However, it is generalised such that, rather than being used to write the blocks of data to the local send buffers, which are now suppressed, it is used instead to store them directly in the sending shared memory window of the corresponding intra-group communicator. Each intra-group task sequentially stores as many (original) buffer data blocks as there are member ranks in its intra-group communicator (npintra-group). This is done concurrently by all ranks, each writing to its designated region in their shared (sending) window, following the specifications in the MPI-3 standard. The process fills up the sending shared memory windows, whose size is given by the original send buffer size times a factor of $\texttt{npintra-group}^2$ (recall subsec. 10.4.2). Once the previous stage is completed, which includes ensuring shared memory data consistency within each intra-group communicator (MPI_Win_fence), the first step of inter-group communication is performed using MPI_Sendrecv point-to-point directives. The appropriate receiving shared memory windows receive the data according to an all-to-all pattern within the inter-group communicator (more about this later). Similarly to before, the part of the original algorithm that maps the original local receive buffer data to the sub-domains is employed and generalised, this time to fetch the data from the receiving shared memory window and to store it in the different sub-domains with the correct layout. This amounts to the local data transposition operations that are required to finalise the complete transposition of the whole data-set, just as it is the case for the flat-MPI algorithm. The difference is that now this operation, rather than acting upon the local buffers independently on each MPI task, as was the case before, is performed concurrently by all intra-group communicator ranks upon the whole shared window data, which is accessible without explicit communication. Shared memory data consistency needs to be ensured once more (MPI_Win_fence).

The process described in the previous paragraph is repeated until all the required inter-group communication steps are fulfilled and therefore all the sub-domain data has been exchanged, i.e., transposed. The all-to-all exchange pattern between sending and receiving shared memory windows involved in each of the inter-group communication steps was devised based on the XOR-transpose method discussed in subsec. 10.5.1. It is computed using an exclusive-OR (XOR) logical operation between the MPI inter-group ranks and inter-group communication step index.

One final technical remark on how the shared memory windows are implemented in the code is in order, before moving to analyse the performance results of the new algorithm. In particular, it was decided to employ a one-dimensional mapping for the pointers used to access the shared memory windows. This means that each shared memory window is treated as a contiguous one-dimensional array. So, moving data between the (three-dimensional) sub-domains and the (one-dimensional) shared memory windows implies explicitly establishing the correspondence between the

three indexes used to access the former and the single index used for the latter. The motivation for this choice stems from the original algorithm employed in VIRIATO, which does the same for the original communication buffers. Therefore, the new algorithm can be seen as the direct generalisation of the original one when replacing the local communication buffers with shared memory windows, which are nothing but a collection of an established number of the former. This also means that their source code is relatively similar, which is an advantage for the main developers of VIRIATO. However, in its present version, this method has a noteworthy shortcoming. Namely, it limits the number of intra-group members to be smaller or equal to the number of tasks involved in the parallel transpose (npe). This limitation can in principle be lifted, by further generalising the memory mapping expressions used to access the shared memory windows. Due to the project's finite time budget, it was not possible to do so before the end of the project was reached, and it is left here as a suggestion for future improvement. An alternative would be to employ a three-dimensional mapping to the pointer used to access the shared memory window. This would make the task of lifting the above mentioned limitation much easier to achieve, at the cost of weakening the similarities between the original and the new generalised transpose algorithms. In practice, this has already been successfully done in the XOR transpose algorithm (Ribeiro & Haefele, 2014) addressed in subsec. 10.5.1.

10.6. *Results and discussion*

In order to use the time budget available within the project efficiently, the development of the new parallel bi-dimensional FFT method was made outside the main VIRIATO s in a stand-alone test-code, already devised in the preceding HLST-VIRIATO (2015) project to serve as a test-bed. It includes only the parts of VIRIATO which are responsible for the FFT algorithm, therefore allowing to study their behaviour without the overhead of executing the whole VIRIATO code. During the current project the test-code was extended to accommodate the new VIRIATO FFT method with shared memory windows described in the previous section. As before, the test-code is written in a way which makes it fully compatible with the main VIRIATO code, in order to yield the subsequent integration of the additional source-code lines in VIRIATO a straight-forward procedure. The results shown in the remainder of this section were obtained using the stand-alone test-code.

10.6.1. Suitable three-dimensional domain

It is important to recall that the motivation behind the MPI-3 shared memory hybridisation of VIRIATO's FFT algorithm is the reduction of the underlying MPI communication complexity involved in the parallel transpose operation, which grows with the square of the number of participating resources ($O(\mathrm{npe})^2$). The idea is to collect a given number of the original local communication buffers into larger regions in memory, called shared memory windows. While the former were used to exchange data between all resources (cores) in an all-to-all communication pattern, the latter serve the same purpose but restricted to a subset of the whole participating resources. This therefore decreases the number of messages to be exchanged at the cost of increasing their size by the same factor, which is given by the square of the number of resources sharing each memory window $(\mathrm{npintra\text{-}group})^2$.

To be able to see the negative impact of the transpose communication complexity on the strong scaling behaviour of the original algorithm, there must be enough grid-nodes in the directions over which the Fourier transform is applied (*xy*-plane). Only then it is possible to distribute them over a large enough number of resources (npe). Furthermore, the problem size must also be big enough to ensure that the cost of the explicit MPI communication stays the dominant factor in the scaling behaviour of the algorithm. Otherwise, it could get overshadowed by memory access costs, leading to different results, as we shall see later on. To fulfil these requirements we use a three-dimensional domain (no *ng*-velocity moments) with the a grid-count of

2048x1024x512 in the (*x*,*y*,*z*) directions, respectively. All the resources are used in the distribution of the domain over the *y*-direction (npe) only, meaning that `npez`=1. This setup, unlike the case of Fig. 109 and Fig. 110 which represents a typical `VIRIATO` production domain, facilitates the analysis of the Fourier transform algorithm scaling properties, which are independent of the distribution over the *z*-direction.

The corresponding strong scaling results are shown in Fig. 111. Each point in the plot corresponds to a single simulation. However, the yielded scaling behaviour is statistically meaningful. It has been consistently reproduced whenever the simulations were repeated, which was done several times. The dashed blue curve corresponds to the original algorithm. It shows the breakup of the scaling behaviour, similar to what was observed before (Fig. 109 and Fig. 110), which for the current domain size occurs beyond `npe`=256 cores. The solid curves correspond to the new MPI hybridised scheme. Each colour corresponds to a different size of the shared memory windows employed, which is established by the number of resources chosen to belong to each of the intra-group communicators. The cases tested correspond to `npintra-group` $\in$ {1,2,4,8,16,32}.

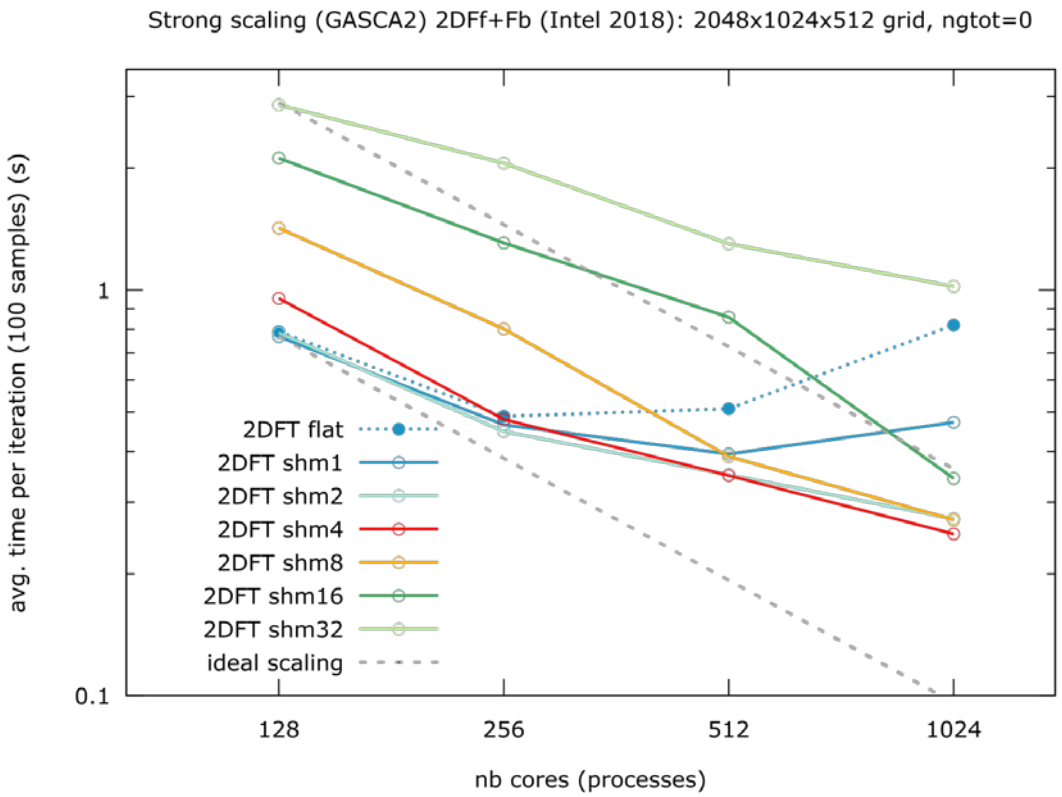


Fig. 111 Strong scaling of the parallel bi-dimensional FFT algorithm on a three-dimensional test domain, corresponding to, in `VIRIATO` terms, a configuration space grid-count of 1024x2048x512 in (*x*,*y*,*z*) with no velocity moments. A sequence of bi-dimensional FFTs both in forward and backward directions is executed 100 times and the average elapsed time is considered. `VIRIATO`'s original algorithm is shown in dashed blue. The extended algorithm using shared memory windows is shown in solid colours, each corresponding to a different number of cores per intra-group communicator, namely, `npintra-group` $\in$ {1,2,4,8,16,32}. Results have been obtained on the A3 (Skylake) partition of the MARCONI supercomputer at CINECA, using the Intel Parallel Studio XE 2018 suite (compiler and MPI library).

The solid blue curve is the limit-case that uses a single core per shared memory window. Hence, the original local communication buffers and the new shared memory windows are equal in size and function. This makes it the ideal test to prove that the new method, under the same circumstances, does not introduce important

overhead costs that could degrade the overall performance of the algorithm. This is not the case, and in fact, it yields even a slight edge in terms of scalability. The reason for this advantage lies in the differences in the implementation of the point-to-point communication directives. The new algorithm uses the XOR algorithm to find the ideal sequence of message exchange pairs, whereas the former one uses a simpler method based on the parity of the MPI ranks to avoid deadlocks.

The remaining solid curves all monotonically decrease, which, even though deviating somewhat from the ideal theoretical linear scaling, is still considerably better than the previous cases. The differences between all the shown hybridised cases stem from the scalability properties of the explicit MPI communication step. The overhead cost related to the access to the shared memory windows has been measured and displays a quantitatively similar scaling for all `npintra-group` values. This shows that, at least for the problem sizes at hand, these operations, which are executed concurrently by all resources inside each intra-group communicator, are efficiently parallelized therein. Of the cases that actually share memory between different resources (`npintra-group`>1), the best results in absolute terms are yielded by the ones with the smallest shared memory windows, namely `npintra-group` $\in$ {2,4} in light-blue and red, respectively. They yield practically indistinguishable results.

As the number of cores per shared memory window increases however, an increasing offset on the curves arises. Looking in more detail at the performance counters in the source code (refer to subsec. 10.6.3 for more details) allows to conclude that the increase in cost is solely due to the explicit MPI communication part of the algorithm. This can be clearly seen in Table 23, where the time spent on MPI communication during the execution of a single pair of forward and backward Fourier transforms distributed over `npe`=256 cores is shown. Up to four cores per window, the time remains roughly constant, but beyond that value, it increases proportionally to the change in `npintra-group`. Further noting that the number of shared memory windows per compute-node dictates, by algorithmic design, how many cores are globally involved in the MPI communication, raises questions about the maximum bandwidth available per core. In other words, how many communicating cores per compute-node are necessary to reach the maximum communication bandwidth? According to the figures in Table 23, the answer seems to be at least 8 cores (`npintra-group`$\leq$4). If this hypothesis turns out to be correct, it would be unfortunate, as it clearly impairs the performance of the hybridised FFT algorithm developed for `VIRIATO`. In relative terms, the case whose scaling deviates the least from the ideal linear scaling is the one with 16 cores per shared memory window (darker green curve in Fig. 111). This yields two shared memory windows per compute-node (one per socket). However, according to the previous discussion, only a fraction of the communication bandwidth is attainable in this configuration. This inflates the total cost of the algorithm by roughly a factor of four when compared to the blue and red curves, which renders its better scaling properties useless.

`npintra-group`	1 core	2 cores	4 cores	8 cores	16 cores	32 cores
nr. SHM windows/node	32	16	8	4	2	1
elapsed time (on 256 cores)	20.1 s	17.8 s	20.4 s	52.4 s	101.4 s	174.7 s

Table 23 Average time spent on explicit MPI communication for different sizes of `npintra-group` (number of cores per shared memory window) on 100 pairs of forward and backward bi-dimensional Fourier transforms performed on the same domain used in Fig. 112 distributed over 256 cores.

Naturally, the proposed hypothesis regarding the communication bandwidth needs to be checked by means of a dedicated test. Namely, an empirical direct measurement of the network bandwidth using point-to-point MPI directives to exchange a fixed amount of data between two compute-nodes as a function of the participating MPI tasks. This test falls outside the scope of the current project, but its relevance means

that it should be made in the near future, probably within the framework of the HLST-CINCOMP3 (2018) project.

To conclude, the scaling results in this section show that, even in the presence of the issues discussed, the new hybridised algorithm can perform better than the original one (flat-MPI) in absolute terms. This shows that MPI-3 shared memory windows can bring advantages, at least for big enough problem sizes, and that the motivation for the project was not ill-posed. On the other hand, while the original transpose was plagued by network latency issues, the pure-MPI hybrid version seems to be limited by communication bandwidth issues, at least on MARCONI with its current software stack. However, it is noteworthy that it is possible to devise a relatively simple change in the current algorithm to address the bandwidth issue. Namely, by allowing to choose how many cores per intra-communicator participate in the communication, one would be able to tune this figure to gain access to more network channels and therefore a higher bandwidth values. Currently that number is fixed to a single core, but since they all access the whole shared memory window, there is no technical limitation which prevents them from sharing the communication step as well, whereby each participating MPI task exchanges a different part of the shared memory window. The communication complexity would still be kept relatively low, as the number of messages would increase linearly with the number of cores involved in the inter-node communication, and not quadratically, as is the case for an all-to-all exchange pattern. Moreover, as this change affects only the communication step of the algorithm, the rest of the algorithm stays untouched, which makes it relatively straightforward to implement. However, as the time budget of the project has been exhausted, this shall be left a recommended improvement for the future.

10.6.2. Typical `VIRIAT0` production domain

In this section we report on results obtained using similar scaling studies as described before, but this time applied to a typical domain size of `VIRIAT0`. Namely, a grid-count of 256^3 in (x,y,z), respectively, with 16 velocity *ng*-moments, the same number as in sec. 10.3. Unfortunately, the gains observed in the previous subsection for the hybridised algorithm do not carry over to this case, as can be seen in Fig. 112. It shows the measurements made on the HELIOS supercomputer (dashed curves), which were already displayed in Fig. 110, and adds their counterpart measurements made on the MARCONI supercomputer (solid curves) for comparison. Barring the case using shared memory windows (green curves), which we shall discuss in more detail shortly, one observes that MARCONI yields better results than HELIOS, not only in absolute terms, but also in terms of the scaling behaviour. Recall that for this setup, it is only beyond 1024 cores (highlighted by the dashed vertical line) that the number of MPI tasks involved in the *xy*-plane transpose starts to increase, while the number of MPI tasks in the *z*-dimension is kept fixed at `npez`=128. For the curves shown, it spans from `npe`=8 to `npe`=32 on MARCONI and to `npe`=128 on HELIOS. The reason why we stopped earlier on MARCONI is that, at the time of the writing of this report, the biggest queue available on the A3 (Skylake) partition allowed the allocation of only 229 nodes. Since we restrict the number of used cores per node to 32 and use only powers of two for the total amount of resources requested, which is an appropriate choice for FFT algorithms, the maximum number of resources achievable is 4096. However, this is enough for our purposes, as the degradation of the scaling already begins at 2048 cores for the problem size under consideration. The yellow curves are an exception to this, but since they correspond to a variant of the algorithm that can not be used effectively in `VIRIAT0` (recall sec. 10.3), we don't need to find were its scaling breaks. In this range of resources, it is clear that, comparing the dashed (HELIOS) and solid (MARCONI) curves for each colour, all flat-MPI algorithm variants scale better on the newer computer. Further, looking more closely at the differences in the measured values for 4096 cores, MARCONI yields speedup factors between two and three relative to HELIOS. This should not come as

a surprise, simply because MARCONI's Skylake partition represents a more modern generation of the same type of NUMA architecture formerly employed on HELIOS.

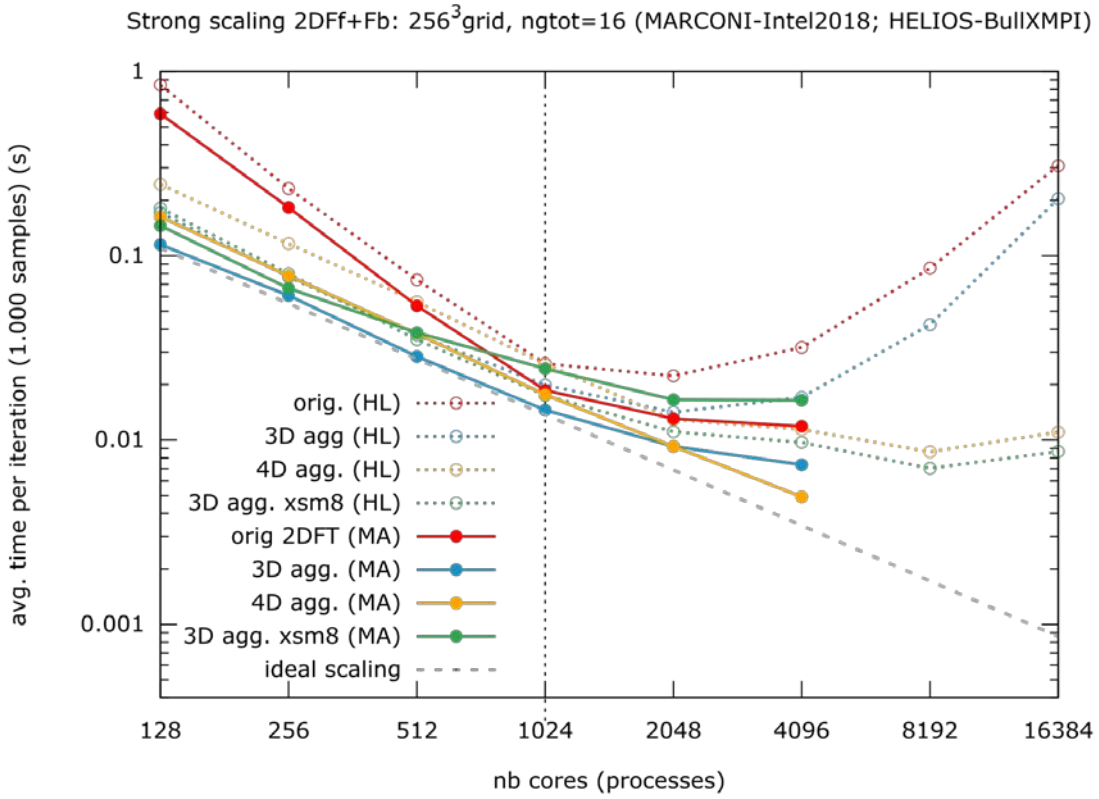


Fig. 112 Same as in Fig. 110 with the addition of curves measured on the MARCONI supercomputer at CINECA, using the Intel Parallel Studio XE 2018 suite (compiler and MPI library). The label 'xms8' stands for XOR transpose method using 8 cores per shared memory window, which is the same hybrid setup used in Fig. 110.

While on one hand, the obtained flat-MPI results are the expected ones, the results yielded for the hybridised case came as a surprise. Not only is it not the best performing algorithm among all the variants, as it was the case on HELIOS, but it is also slower, in absolute terms, than the same simulation performed on HELIOS. The first part can perhaps be justified by the different MPI libraries used on both machines. It is not unrealistic to assume that, had one used the Intel MPI library on HELIOS for the same measurements instead of the BullX MPI library, this could have led to different scaling behaviours for the several variants of the algorithm, including rendering other variants faster than the hybridised one. However, the fact that the hybridised algorithm is slower on MARCONI than on HELIOS is slightly more puzzling. Note that, to enable a fair comparison with the previous HELIOS results, the hybridised XOR transpose method is used in Fig. 112. Nevertheless, the same simulations using the new hybridised version of the FFT algorithm of `VIRIATO` (refer to subsec. 10.5.2) were also performed, yielding very similar results (not shown).

In order to understand the reasons why the improvements yielded in the previous subsection did not carry over to the current case, several additional simulations were performed. For those, we measured in detail the different components of the `VIRIATO`'s transpose algorithm, and how their weight changes with the domain size. These comprise essentially the explicit MPI communication and memory access overhead due to reading/writing data from/to the communication buffers/shared memory windows and, additionally in the hybrid case, the shared memory window data synchronisation cost (`MPI_Win_fence`). From this analysis (not shown here) we could conclude that the typical domain sizes used for physics studies with

VIRIATO are too small to feel the benefits of the hybridisation, especially considering that the domain is distributed over two dimensions.

For a typical VIRIATO simulation, one starts by maxing out the number of resources over which the *z*-direction can be distributed (npez), because the decomposition in this direction has the best scaling properties (it does not involve all-to-all communication steps). This yields sub-domains that are two orders of magnitude smaller than the global domain, and that need to be further distributed in the *y*-direction (npe), which involves all-to-all data exchange patterns. Unlike the case studied in subsec. 10.6.1, for which we kept npez=1, the explicit communication in the algorithm is no longer its dominant part. Instead, the shared memory access overhead, which is found not to scale for these problem sizes, starts to play a more dominant role. Consequently, all the expected scaling benefits stemming from the hybridisation, which were measured before, are simply not reflected in the overall scaling of the algorithm in this case.

It seems that, if the sub-domains are small enough, the flat-MPI approach that treats the memory as purely local to each task has advantages compared to the approach invoking a shared memory windows, whose data consistency needs to be explicitly enforced. The hardware's cache hierarchy is probably more efficiently exploited in the former. In light of this, cases like the current one, for which the problem size means that the cost of the shared memory access becomes comparable to the explicit MPI communication cost, are expected to be less performing than their flat-MPI counterpart. One could still argue that there is still room for optimisation of the shared memory access in the algorithm, which is true. However, the fact that similar results were obtained with the hybridised XOR transposition method, whose implementation of the shared memory data access is quite different, seems to support the idea, for these problem sizes, the shared memory windows fail to bring performance advantages.

Obviously, thoroughly checking the hypothesis proposed in this section to explain the underwhelming results obtained on MARCONI with the implemented pure MPI hybridization techniques is something highly desirable. As mentioned before, there are already plans to perform, at least, some of these checks in the framework of the HLST-CINCOMP 3 (2018) project. The conclusions that will be drawn should be of general interest to the MARCONI users community. For the time being, the absence of a clear benefit in terms of scalability for typical problem sizes of VIRIATO, together with the limited amount of time available to conclude the project, led to the decision to not integrate the changes back into VIRIATO. Despite that, the source code of the new algorithm has been made available to the code developers, who are encouraged to experiment with the new algorithm, and if they so decide, later on integrate it themselves into the main source code. As highlighted before, this task is a straightforward one, as the new algorithm has been constructed with this specific requirement in mind.

10.6.3. Issues encountered with time measurement facilities

After the first working version of the new pure MPI hybridized FFT algorithm for VIRIATO was available, the performance assessment phase began. This coincided with the transition from the A1 Broadwell partition to the newest A3 Skylake partition and it was soon realised that there were issues with the obtained results. Initially, it was thought that the problem was in the new algorithm, which had just been developed and probably needed more time invested in its optimisation. However, when we tried to reproduce the scaling results obtained on HELIOS using the algorithms already developed in the previous HLST-VIRIATO (2015) project, we failed. It became clear that the issue was not in our code, but rather in the MARCONI hardware configuration. It took quite some time (roughly one month), in close

collaboration with other members of the HLST Core Team, to find out that the problem was being caused by the choice of the system clock, namely the HPET. Apparently this timer is not only slow, but also does not scale when resources are used in parallel (refer to the HLST-CINCOMP2 2017 for a detailed analysis). Time measurements (made via `MPI_Wtime()` calls) were completely shadowing the actual cost of the parallel FFT algorithms we were trying to instrument. As a work around, the profiling module that had been previously developed to instrument the standalone test-code was extended with a C function, called using ISO C bindings in FORTRAN, which reads the TSC invariant counter. This returns the number of CPU cycles, which can be normalised using the processor clock frequency to yield time measurements. Moreover, varying CPU clock frequencies, which are allowed in the Intel Skylake architecture, are automatically taken into account by the TSC invariant counter. Contrary to the HPET, the TSC counter measurements behave in an efficient manner, and allow the proper profiling of the algorithms at hand. Indeed, the MARCONI scaling curves shown in Fig. 112, as well as all remaining scaling measurements made in this project, were only possible after the described workaround was implemented. Meanwhile, the issue has been acknowledged by the MARCONI Support Team and the replacement of the HPET timer with the TSC counterpart on the A3 partition of MARCONI is planned.

10.7. ***Summary and outlook***

The main goal of this project was to improve the parallel scalability of `VIRIATO` by hybridising the current implementation of the parallel Fast Fourier Transforms in the framework of the new MPI-3 standard. This is a very interesting new concept with very good portability characteristics, as it is part of the newest MPI standard. In the context of `VIRIATO`, it allows changing the FFT algorithm separately, leaving the remaining source code's flat-MPI structure untouched. In other words, it allows a partial hybridisation of an existing flat-MPI implementation, while still ensuring an efficient usage of resources at runtime by the non-hybridised parts of the code, something that would be impossible with an OpenMP hybridisation.

The MPI-3 hybridisation concept with shared memory windows had already been successfully demonstrated using the XOR-transpose method towards the end of the previous HLST-VIRIATO (2015) project. The development, implementation and verification of the new parallel FFT algorithm for `VIRIATO` using shared memory windows took most of the current's project time budget. The complexity of this task has been underestimated, as it was assumed that the previously acquired experience in doing a similar generalisation on the XOR transpose algorithm would be directly applicable to `VIRIATO`'s algorithm. Undoubtedly, it provided a lot of useful knowledge on how to handle MPI-3 shared memory windows. On the other hand, however, it turned out that the algorithm differences between both methods meant that very little algorithmic knowledge was applicable, and a completely new transpose method, compatible with `VIRIATO`, had to be devised from scratch.

The task was successfully finished, and a new hybridised parallel FFT algorithm for `VIRIATO` is now available. The complexity (number of message exchange pairs) of the required all-to-all communication could be greatly reduced (Ribeiro & Haefele, 2014) by making use of the shared memory resources available inside a compute-node, as was planned. Even though the pure MPI hybridization is a very promising concept that seemed well fitted to `VIRIATO`'s parallel FFT algorithm on paper, in reality obtaining good scaling performance turned out to be something rather dependent on both the hardware and software stack (MPI library), as well as the problem size under consideration. Previous results on HELIOS for typical `VIRIATO` problem sizes revealed clear advantages compared to the original flat-MPI version of the algorithm, but these could not be reproduced on MARCONI. On the other hand, for better suited (larger) problem sizes, a sound benefit was also achieved on the Skylake partition of MARCONI, despite a potential limitation regarding

communication bandwidth which was found when larger shared memory windows were used. At this stage it is not yet clear whether this is related to inter-node network properties, or rather to intra-node NUMA communication characteristics. Future dedicated communication bandwidth measurements (ping-pong tests) are needed to clarify this point unambiguously. As a intermediate step, a simple improvement to the existing algorithm to address this issue has even been proposed, although not implemented due to lack of time. It exploits the idea of involving more than one core per shared memory window on the explicit inter-node MPI communication, and is left as a suggestion for future work.

A few simulations were performed on the Intel Xeon Phi (Knights Landing) partition of MARCONI. None of the obtained results, however, showed advantages of the new hybridised algorithm over the flat-MPI counterpart. Not even for the bigger domain sizes, for which advantages could be measured on the Skylake partition. Additionally, compared to the Skylake results, the execution times on KNL hardware were always slower in absolute terms. However, taking into consideration that not much time was invested in this subject, these findings should be considered still preliminary.

In practice, since no clear-cut advantage of the new hybridised FFT algorithm (on MARCONI) could be found for the typical problems sizes of `VIRIATO`, it was decided to not integrate the changes back into `VIRIATO`'s source code. Nevertheless, the source code of the new algorithm has been made available to the code developers, who are encouraged to experiment with the new algorithm, and if they so decide, later on integrate it themselves in `VIRIATO`. As highlighted before, this task is a straightforward one, as the new algorithm has been constructed with this specific requirement in mind.

As a final word, it should be highlighted that the experience gained with the advanced topic of pure MPI hybridisation in the framework of this project is very valuable. This experience is expected to be beneficial for future HLST projects, not necessarily restricted to `VIRIATO`. The positive results that could also be obtained with the newly hybridised FFT algorithm stand to prove exactly that.

References

Fehér, T. (2015). *Final report on HLST project PARSOLPS.*

Intel. (2013). *MPI Benchmark.* Retrieved 2015, from https://software.intel.com/en-us/articles/intel-mpi-benchmarks/

LRZ. (2016, January 14). Introduction to hybrid programming in HPC. Garching, Germany.

Reid, J. L. (2010). *Porting and parallelising SOLPS on HECTOR.* EUFORIA Report.

Reiter, D. (2009). *The EIRENE Code User Manual.*

Reiter, D. (2015). Private cammunication.

Ribeiro, T. (2012). *Final report on HLST project NEMOFFT.*

Ribeiro, T. (2015). *Final report on HLST project VIRIATO.*

Ribeiro, T. T., & Haefele, M. (2014). *Parallel Computing: Accelerating Computational Science and Engineering*, *25*, p. 415.

RWTH Aachen University. (2015). *MUST MPI Runtime Error Detection Tool*. Retrieved from https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST