EUROfusion

EUROFUSION WPISA-REP(18) 23083

R Hatzky et al.

HLST Core Team Report 2018

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 and 2019-2020 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

HLST Core Team Report 2018

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 and 2019–2020 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
AMGBOUT	Kab Seok Kang	finished
BIT2-3	Kab Seok Kang	running
CINCOMP3	Serhiy Mochalskyy	finished
JOKLA	Tamás Fehér	finished
MPI3-DG	Tiago Ribeiro	running
PICOPT	Nils Moschüring	running
REFMULIO	Tiago Ribeiro	finished
SPICE	Serhiy Mochalskyy	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 JOKLA project.

The JOREK code is a nonlinear extended MHD code that can resolve toroidal X-point geometries. Its key application areas are the simulation of Edge Localized Modes (ELMs) and disruptions. The JOKLA project improved the performance of the JOREK code on the KNL and Skylake architectures.
The OpenMP scaling of the matrix construction part of the code was investigated. Synchronization using atomic directives scales better for a large number of threads, but its overhead is comparable to the actual computation that needs to be synchronized. A modified version of the critical section synchronization together with a properly chosen hybrid MPI/OpenMP configuration gives the best performance.
It was found that the linear equation solver is also sensitive to the MPI/OpenMP execution configuration, and the best performance can be achieved if we limit the number of threads per node that is used by the solver.
The work continued by improving the vectorization of the matrix construction. In a separate test bed, the matrix construction subroutine was vectorized, and the data locality was improved, which lead to a 4x speedup for this subroutine on Intel Skylake and a 3x speedup on Intel KNL. After vectorization the code performance is limited by memory and cache bandwidth. The same changes, when integrated into the main code led to a factor of two speedup of the matrix construction part. The execution time of the JOREK code is generally 2.2–2.9 times longer on KNL than on Skylake in a node to node comparison.

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

For the AMBGOUT project an algebraic multigrid solver branch (petscamg) concerning the BOUT++ code was set up in `github` and installed on the Marconi machine. As an initial step K.S. Kang installed the PETSc library on the Marconi machine to use two third party solvers, the ML solver from the Trilinos package and the BoomerAMG solver from the Hypre package.

J. Omotani from CCFE prepared a realistic test case so we could verify the data conversion from BOUT++ to the PETSc library (and vice versa) with several numerical tests. We achieved numerical performance results for different solvers: a direct solver on a single core and the PGMRES solver with different preconditioners: the block Jacobi, the GAMG for various aggregation smoothing steps, the ML from Trilinos, and the BoomerAMG from Hypre. The PGMRES solver was executed on a single core and on multiple cores. We concluded from the results that the BoomerAMG solver from Hypre is the fastest solver. It was accepted by the BOUT++ developers.
In addition, we prepared a 3D solver with the algebraic multigrid algorithm in the framework of the PETSc library. It is planned that the project coordinator will debug and finalize it.

For the BIT2-3 project support was given to test and implement the new 3D multigrid solver version. Currently, we are waiting for feedback from the PI which might trigger some adjustments.

Serhiy Mochalskyy worked on the CINCOMP3 (Part 1) and SPICE projects.

For the CINCOMP3 project we analyzed the performance of the Marconi supercomputer which went into operation in July 2016. Different issues were found that significantly limit its use. Some of them were resolved by the Marconi support team, others however are still under investigation.
Three different benchmarks were used to test the Intel Omni-Path interconnect. It was found that the bandwidth can be saturated by using two MPI tasks (reaching ~92 % of the theoretical peak value). The bi-directional bandwidth rate was checked as well. It works properly providing ~92–95 % of the theoretical value.
The inter-node benchmarks on the A2 partition revealed a drastic bandwidth drop for message sizes larger than 64 MB.
The STREAM benchmark shows an intra-node memory bandwidth drop for the *copy* test using the Intel 2018 compiler and the *-xCORE-AVX512* and *-mtune=skylake* compilation flags.
The so-called “three code benchmark” was executed regularly in order to check the stability of Marconi in terms of the execution time for real production codes. It was found that the wall clock time of identical codes can fluctuate significantly (~20 %) from one run to the next. It was also detected that these fluctuations appear only in the MPI communication, the computation time always stays constant.

In the SPICE project a 3D PETSc parallel Poisson solver was developed. The solver was validated using a variety of tests using synthetic and real data. The code scales linearly up to 128 MPI tasks and provides good performance. The solver works correctly using both the typical SPICE3 grid (128×128×129) and a larger grid (550×550×550). The solver was also tested on real data from the SPICE3 code providing identical results to the serial multigrid solver for test cases that include internal objects inside the simulation domain and different potential at each object. The solver has been integrated into the SPICE code and production runs were performed.

Nils Moschüring worked on the CINCOMP3 (Part 2) and PICOPT projects.

For CINCOMP3 we investigated the performance fluctuations of the MPI_Allreduce function using a custom code. This code measures the run time of single function calls in order to provide data for histograms detailing the mean run time and the standard deviation of the run time (or run time fluctuation).
Analyzing the Marconi A3 partition with this tool while varying several parameters, it was found that it exhibits run time fluctuations of several orders of magnitude in all cases. A very strong reaction was found when varying the amount of cores per node.

Using fewer cores per node decreases the fluctuations decisively. These findings are a hint that hardware interrupts may be at fault.
In order to identify the specific element responsible for the fluctuations, we performed the benchmark on six other supercomputers. We could therefore test the architecture and the interconnect independently. The obtained results are very dissimilar. The different machines have mean run times which differ in the range of half an order of magnitude. The fluctuations are even more diverse and differ on the order of multiple orders of magnitude. Unfortunately the obtained data does not single out specific architectures or interconnects. In any case, the Marconi machine exhibits the largest fluctuations and its mean run time is not the fastest (nor the slowest). The investigations were expanded to include several other parameters, like the software version and other configuration parameters. No conclusive result has been obtained so far.

We started the PICOPT project with an in-depth familiarization with the underlying theory. Previous experience with the standard particle-in-cell algorithm was leveraged and extended with the necessary details and differences to gyrokinetic particle-in-cell. After getting acquainted with the mathematical background, we continued with researching the current state of auto vectorization. This field is rapidly developing and staying on top of its details will prove invaluable for the envisioned optimizations of the GK PIC algorithm.
Subsequentially, multiple codes were investigated. From the widely used GK PIC code ORB5 we continued to the very modern PIC code PIConGPU. ORB5 has a very novel and interesting way of structuring a GK PIC code. Implementing the gyro-points as first order objects enables some powerful optimization schemes, mostly related to the spatially localized nature of the gyro-points. Leveraging the full potential of these changes in terms of the exploitation of vectorization is still not finished and may require substantial additional investments. We made suggestions for additional improvements to the structure of the algorithm. PIConGPU has a completely different approach to scientific computing than ORB5. This code was written using elaborate software engineering and therefore exhibits a lot of advantages which cannot be introduced into other codes without a complete rewrite of the code base. Unfortunately, it is also impossible to utilize it partially in order to optimize other codes, including ORB5. In summary, using the PIConGPU code base directly for GK PIC involves lots of obstacles and problems and will therefore not be pursued directly.

Tiago Ribeiro worked on the MPI3-DG and REFMULIO projects.

The high order 3D Discontinuous Galerkin code `Fluxo` has been developed to solve the 3D full MHD equations, including nonlinear and resistive terms. `Fluxo` is fully MPI parallelised and exhibits very good weak and strong scaling properties on current NUMA architectures up to $\mathcal{O}(10{,}000)$ MPI ranks. The main goal of the HLST-MPI3-DG project was to devise a strategy to further extend its scalability. To that end, a detailed assessment of the communication behaviour within `Fluxo` was performed. It showed that `Fluxo`'s implementation of communication and computation overlapping works on Marconi. At the same time, it provided hints on how to improve its ratio by changing the order of some computational intensive instructions. Additionally, an optimised communication strategy has been devised and implemented. It better reflects the NUMA architecture by introducing a two-level communicator hierarchy that distinguishes between communication inside and across compute-nodes. This change alone resulted in better scaling performance of the whole code. The granularity refinement made to the initiation and completion steps of the communication processes affects the non-trivial global data dependency equilibrium between all sub-domains in the simulation. Whether or not it makes sense to further replace the intra-node communication with MPI-3 shared memory (SHM) windows, which requires extensive code changes, is still an open question. To try to answer it, several detailed intra-node and inter-node communication bandwidth

studies were made, not only on Marconi, but also on other machines, like the SuperMUC (LRZ) or Cobra (MPCDF). In that process, a bug was found in the Intel MPI library running on Intel Omni-path interconnected networks. It caused an extensive bandwidth drop when large MPI-3 SHM windows were used. The issue was first acknowledged and later fixed by Intel in the official release *Intel(R) MPI Library for Linux* OS, Version 2019 Update 1* (*Build 20181016*).

The REFMUL3 code, developed at the IPFN-IST, is a 3D full-wave code using the Yee scheme with full polarisation, that simultaneously copes with o- and x-modes and supports a general external magnetic field and a dynamic plasma. Its parallel implementation was done within a HLST project (2016) yielding very good scalability over a few thousands of cores. However, I/O became a major bottleneck, as the parallelisation allows for much larger grids. The HLST-REFMULIO project, requested support for the development of parallel input/output (I/O) capabilities in REFMUL3. Two parallel I/O solutions are now available, which further allow the output data to be directly analysed in ParaView. The first is simpler in its implementation. It writes one (serial) HDF5 file per subdomain (MPI task), which is enough to address the basic I/O bottleneck issue to some extent. It has, however, an important drawback in that it is domain decomposition dependent. The second solution lifts this limitation by having all subdomain data stored within a single file, which is accessed collectively by all MPI ranks via the parallel HDF5 (PHDF5) library. Because these files are compatible with serial access, it is further possible to read antennas and wave-guides from external files. Additionally, the PHDF5 I/O functions available provide the ingredients needed for a basic check-pointing and restart file infrastructure. In terms of performance, the PHDF5 solution is superior to the simpler multi-file solution up to roughly 64 MPI sub-domains. Since the code is hybrid, in the limit, this should allow for efficient I/O in simulations up to 64 compute-nodes. Beyond this point, however, the situation inverts, probably because no time was left to invest in performance tuning of REFMUL3's PHDF5 functions. This is left as a recommendation for the future. Finally, the possibility to perform the impulse response calibration for domain decomposed simulations was also implemented, as specifically requested by the project coordinator.

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

1.2.1. Dissemination

Ribeiro, T. and Hatzky, R.: MPI communication bandwidth, *HPC workshop*, 11th October 2018, Rokkasho, Japan.

1.2.2. Training

Kab Seok Kang has visited:
The project coordinator J. Parker to work for the AMGBOUT project, 12th–16th March 2018, CCFE, UK.

Serhiy Mochalskyy has visited:
The project coordinator M. Komm to work for the SPICE project, 24th–28th September 2018, Institute of Plasma Physics of CAS, Czech Republic.

Tiago Ribeiro has visited:
The project coordinator Filipe da Silva to work for the REFMULIO project, 9th–27th July 2018, IPFN-IST, Lisbon, Portugal.

1.2.3. Internal training

Roman Hatzky has attended:
Numerical Methods for the Kinetic Equations of Plasma Physics (NumKin2018), 22nd–26th October 2018, IPP, Garching, Germany.

Kab Seok Kang has attended:
PRACE PATC Course: *Uncertainty quantification*, 16th–18th May 2018, Maison de la Simulation, Saclay, France.
European Workshops on Automatic Differentiation, 19th–20th November 2018, Friedrich-Schiller- Universität, Jena, Germany.

Serhiy Mochalskyy and Tiago Ribeiro have attended:
Course: *Python for HPC*, 20th-21st November 2018, MPCDF, Garching, Germany.

Nils Moschüring has attended:
PRACE PATC Course: *Introduction to hybrid programming in HPC*, 18th January 2018, LRZ, Garching, Germany.

Tiago Ribeiro has attended:

- GoLP Mini Course: *An introduction to turbulence in magnetized plasmas* by Nuno Loureiro (MIT), IST Lisbon, 18th–20th July 2018.
- NAG webinar*: How to identify and quantify causes of MPI underperformance using the Intel® Trace Analyzer and Collector (ITAC)*, 6th December 2018.

1.2.4. Workshops & conferences

Roman Hatzky has attended:
IPP Theory Meeting, 19th–23th November 2018, Schloss Ringberg, Tegernsee, Germany.

da Silva, F., Heuraux, S., Ricardo, E., Ribeiro, T., Despres, B. and Campos Pinto, M.: Modelling Reflectometry Diagnostics: Finite-Difference Time-Domain Simulation of Reflectometry in Fusion Plasmas, *5th International Conference on Frontier in Diagnostic Technologies (ICFDT5)*, 3rd–5th October 2018, Frascati, Italy.

Latu, G., Asahi, Y., Bigot, J., Fehér, T.B. and Grandgirard, V.: Scaling and optimizing the GYSELA code on a cluster of many-core processors, *9th International Workshop*

on Applications for Multi-Core Architectures, 24th–27th September 2018, ENS Lyon, France.

Kang, K.S.: Multigrid solver using PETSc, *BOUT++ Workshop*, 5th–7th December 2018, York Plasma Institute, University of York, UK.

Vicente, J., da Silva, F., Heuraux, S., Conway, G.D., Silva, C., and Ribeiro, T.: Turbulence level effects on conventional reflectometry measurements observed with 2D full-wave simulations, *18th Topical Conference on High Temperature Plasma Diagnostics (HTPD)*, 15th–19th April 2018, San Diego, USA.

1.2.5. Publications

Manz, P., Stegmeir, A., Schmid, B., Ribeiro, T.T, Birkenmeier, G., Fedorczak, N., Garland, S., Hallatschek, K., Ramisch, M. and Scott, B.D.: Magnetic configuration effects on the Reynolds stress in the plasma edge, *Phys. Plasmas* **25** (2018) 072508.

Vicente, J., da Silva, F., Heuraux, S., Conway, G.D, Silva, C., and Ribeiro, T.: Turbulence level effects on conventional reflectometry measurements observed with 2D full-wave simulations, *Rev. of Sci. Instrum.* **89** (2018) 10H110.

1.2.6. 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 JOKLA

2.1. *Introduction*

The JOREK code (Huysmans & Czarny, 2007) is a nonlinear extended MHD code that can resolve toroidal X-point geometries. Its key application areas are the simulation of Edge Localized Modes (ELMs) and disruptions. It is the most important numerical tool in Europe to study MHD instabilities in realistic tokamak geometry. JOREK has been granted a large amount of CPU hours on the Knights Landing (KNL) partition of the Marconi-Fusion supercomputer. The aim of the JOKLA project is to improve the performance of the JOREK code on the KNL architecture and in general for many-core CPUs.

The two main challenges are the efficient usage of the wide vector registers and the scaling over a large number of threads. We should note that these are not unique requirements for the KNL architecture. In fact, the Skylake nodes on Marconi have an identical vector register size, and the number of cores per node is also comparable. Because of these similarities, we will test and optimize the code for both the KNL and the Skylake architectures.

2.2. *Profiling*

The first phase of the project started with a detailed profiling on both the KNL and the Skylake partitions. We used an ASDEX Upgrade ELM simulation setup (Hölzl, 2018). The main parameters for the test case are listed in Table 2. We have two variants: a small (*ntor*=3) and a medium (*ntor*=17) size test case. The benchmark was prepared by running the test for 2000 time steps. Using the restart files of this simulation we then performed three time steps in the nonlinear phase of the ELM crash, and used measurements from these three steps to analyze the code performance.

Resolution		JOREK parameters		
			small	Medium
n_tht	173	model	303	303
n_radial	120	N_tor	3	17
n_open	10	N_period	8	1
n_pol	160	N_plane	4	32
n_leg	18	MPI tasks	2	18
n_private	8	Compute nodes	1	18

Table 2 Main parameters of the AUG ELM test case

We compared the execution time of different code regions in the nonlinear phase of the simulation (after time step 2000). Fig. 1 shows the execution time of different parts of the computation on Skylake (red) and KNL (green) for the small test case. At the first step after the restart, we have an LU decomposition which dominates the execution time. In subsequent time steps, the LU decomposition is not repeated (unless the convergence becomes bad); instead the factorized matrix is used as a preconditioner. The right hand side of Fig. 1 shows the execution time of the third time step after the restart. At this time step the matrix construction and the iterative solver take the largest share of the execution time. During the first time step the SKL node is 2.9 times faster than the KNL node, later the ratio becomes 2.2.

Similar measurements were performed in the linear phase of the simulation (time steps 500–503). The results are very similar to the nonlinear case, only the execution time of the GMRES solver differs. In the following we will focus on the nonlinear phase of the simulation.

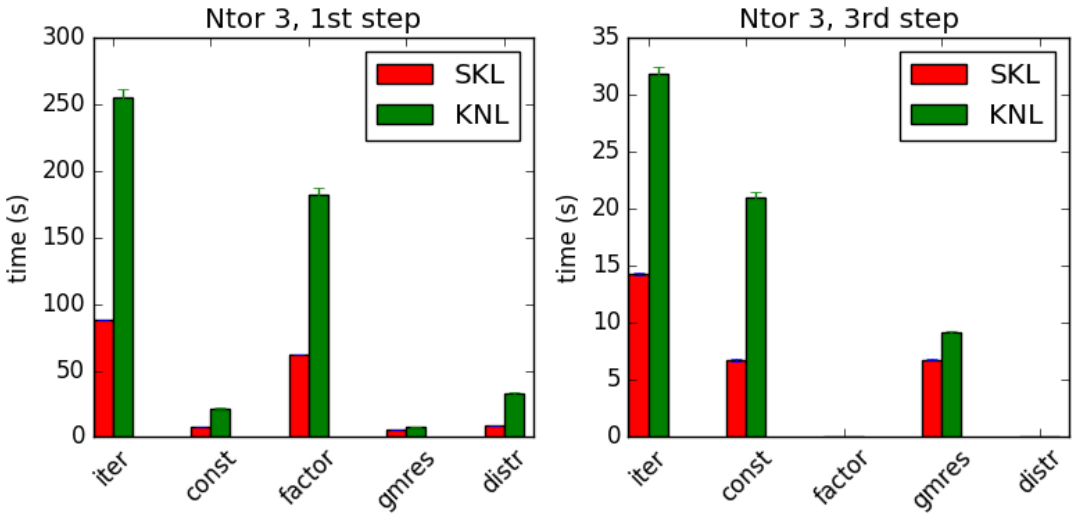

Fig. 1 JOREK small test case execution time: first time step (left) and third time step (right) after restart. Red bars show the execution time on Skylake, while green bars show the execution time on Knights Landing. The first group of bars denote the total time of an iteration, which can be decomposed into matrix construction (const), LU factorization (factor), iterative solver (GMRES) and matrix distribution (distr).

2.3. *Compiler options*

The JOREK code uses the PaStiX (Parallel Sparse Matrix) library as a linear solver. The PaStiX library depends on the Scotch package to calculate the matrix ordering (permutation of the matrix to reduce fill-ins). We used the following optimization options during compilation of all three packages: *-O3 -no-prec-div -xCORE-AVX512 -mtune=Skylake.* For the compilation of the JOREK code, we additionally used the *-align array64byte* option.

We tested whether changing the optimization level or changing the precision of divisions would improve the execution time. Fig. 2 shows that the default settings (*-O3*) were already optimal.

Due to a compiler bug, initially it was not possible to test the effects of interprocedural optimizations among separate files (*-ipo* flag). This problem was reported to Intel, and it was later fixed in update 3 of the Intel 2018 compiler suite. With the latest compiler version, it was found that overall the *–ipo* flag does not improve the performance.

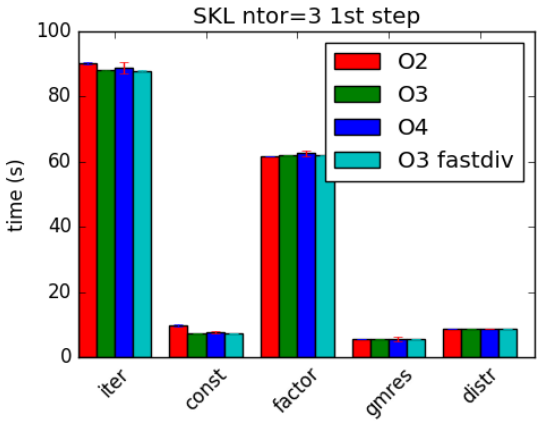

Fig. 2 Execution times using different compiler flags

2.4. *OpenMP scaling*

2.4.1. Skylake

The OpenMP scaling of different parts of the code was tested. In this section we present strong scaling tests, where the amount of work is kept fixed as we increase the number of threads. First, we focus on the matrix construction. While the small test case scaled almost ideally (see Fig. 3), the large test case did not scale above four threads. Using the VTune Amplifier tool, it was identified that a critical section in the *construct_matrix* subroutine is responsible for the scaling problem. This subroutine has a large critical section around the loops which update the following global variables: `irn_glob(ilarge2)`, `jcn_glob(ilarge2)`, `A_glob(ilarge2)`.

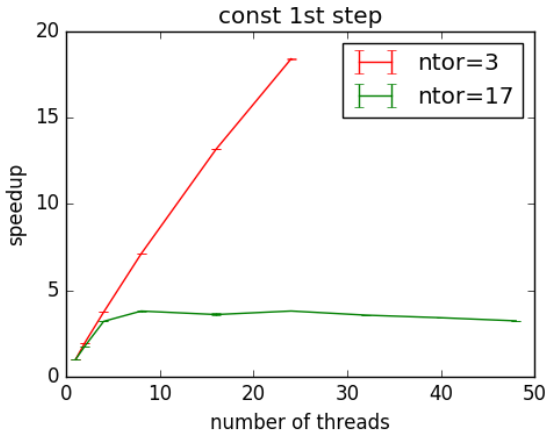

Fig. 3 Speedup of the matrix construction for the small test case (red) and the medium size test case (green) on a Skylake node. The problem size is kept constant (strong scaling).

Two different methods were implemented in order to improve the scaling:

- Atomic: using a single atomic directive to update `A_glob`.
- Critical buffer: each thread stores values in a local buffer with 1M elements, when buffer full: small critical section to update `A_glob`.

We should note that synchronization is not absolutely necessary to set `irn_glob` and `jcn_glob` since all threads set the same value.

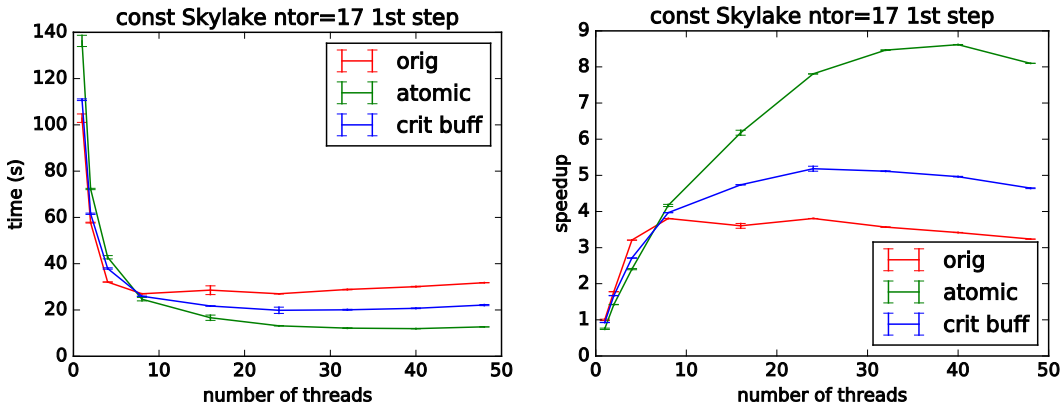

Fig. 4 Matrix construction execution time (left) and speedup (right).

The right side of Fig. 4 shows that replacing the critical section with an atomic directive improves the speedup of the matrix construction by more than a factor of two. Introducing an extra buffer for the critical updates does not help significantly.

The left hand side of the figure shows that using the atomic directive leads to larger execution times on a single core. This is also visible on the right hand side of the

figure (speedup), and it is later compensated by better scaling properties. This will be further discussed in Section 2.5.6.

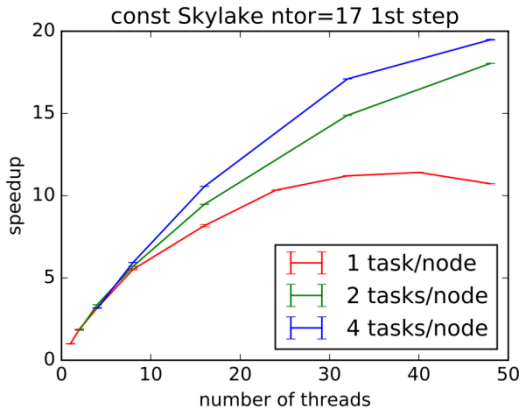

Fig. 5 Scaling of the matrix construction using different numbers of MPI tasks. The execution time is shown as a function of the total number of threads/node.

The scaling of the matrix construction degrades above 24 threads. To improve this, we can divide the threads among more MPI tasks. This leads to less OpenMP synchronization and faster matrix construction as shown in Fig. 5.

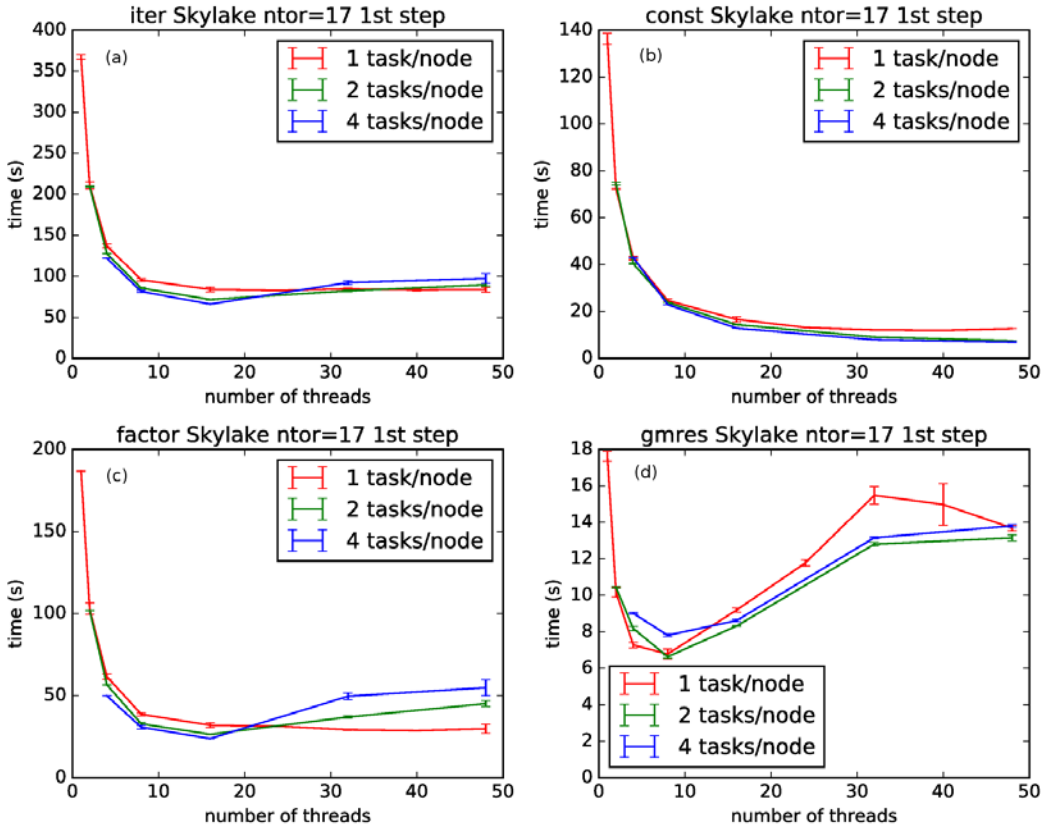

Fig. 6 Execution times during the iteration after restart. The *x*-axis represents the total number of threads per node. (a) Total time of the iteration, (b) matrix construction, (c) LU factorization, (d) iterative solver.

The LU factorization and GMRES steps do not necessarily improve when we increase the number of MPI tasks (see Fig. 6). In fact, when all the cores of the nodes are utilized for the calculation, then the fastest setting for the PaStiX library is one task per node (see Fig. 6(c)). But if we limit the total number of cores per node that are used, then the factorization and solve steps are fast even with 4 MPI tasks (see Fig. 6(c)–(d) between 8 and 16 threads). A switch was implemented in JOREK

which allows us to control the total number of threads used by PaStiX. We can define the following variable in the input file

```
pastix_maxthrd=16,
```

this would limit the total number of threads within a node that PaStiX can use to 16. The matrix construction keeps the value defined by *OMP_NUM_THREADS.* This leads to a factor of 1.7x speedup overall using 4 MPI tasks/node for the medium test case (72 MPI tasks in 18 nodes).

2.4.2. KNL performance

The modified code was tested on KNL nodes with different numbers of MPI tasks/node and different numbers of hyper-threads. The best configuration was found to be 4 MPI tasks per node, 17 threads/task for matrix construction, 4 threads/task for PaStiX and no hyper-threads. The execution times of the medium test case between KNL and Skylake are compared in Fig. 7. The code is 2.2–2.5 times faster on a Skylake node.

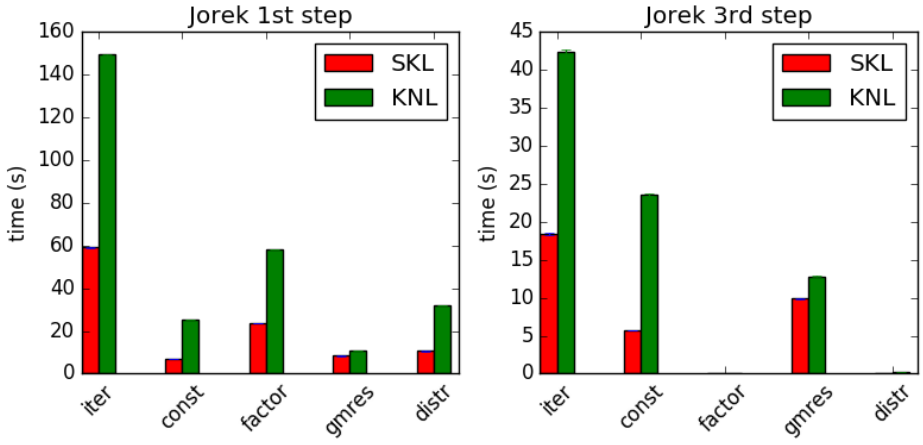

Fig. 7 Medium test case (*ntor*=17) execution times on KNL and SKL

A few issues were observed with the medium size test case: PaStiX by default tries to use the *hwloc* library to bind the threads to the cores. This fails in some cases, causing very long execution times. Introducing an upper limit for the threads helps to overcome this problem.

2.5. *Vectorization*

The PaStiX library uses MKL routines, which should already be properly vectorized for the target architectures. In contrast, the matrix construction consists of user code only, and is not vectorized. To improve the performance of the matrix construction, the single core performance of the *element_matrix_fft* subroutine was investigated.

It requires 18 nodes to test the whole code with the medium size test case (*ntor*=17). In this section we are interested in single core and single node performance, therefore a testbed was prepared to call the *element_matrix_fft* subroutine independently of the main code. This allows us to perform single node tests of the subroutine with the same parameters (*ntor*=17). Unless otherwise mentioned, the tests in this section are performed on the Skylake partition using the Intel Fortran compiler.

2.5.1. Matrix element calculation

Fig. 8 shows an outline of the matrix construction subroutine. In lines 1–14 (compute loops) we have seven nested loops that define the *ELM_p* temporary array, by first calculating the intermediate values *A, B, C*; and then the elements of the *ELM_p* array. Each of *A, B,* and *C* refer to a larger set of scalar and array variables. These loops are computationally intensive since calculating *A, B,* and *C* involves around a

thousand lines of arithmetic instructions. The iteration space of the individual loops is small, *n_gauss*, *n_vertex_max*, and *n_order+1* are all equal to four. The parameter *n_plane* is set in accordance with the physical model that we are calculating (Table 2). For the medium size test case *n_plane=32*.

In lines 15–21 in Fig. 8 (transform loops), the data elements from the *ELM_p* and three similar additional arrays are Fourier transformed and the result is stored in the *ELM* array. Considering the $O(n \log n)$ complexity of the fast Fourier transform with *n_plane=32* elements at a time, we can see that the arithmetic intensity of the transformation is only $\frac{32 \log(32)}{32 \times 8} \approx 0.6 \frac{\text{Flops}}{\text{byte}}$, which makes it very likely that this part of the computation will be limited by the memory bandwidth.

compute loops / transform loops

```
1. do ms=1, n_gauss
2.   do mt=1, n_gauss
3.     do mp=1,n_plane
4.       A(mp, ms, mt) = …
5.       do i=1,n_vertex_max
6.         do j=1,n_order+1
7.           B(i,j,ms,mt) = …
8.           do k=1,n_vertex_max
9.             do l=1,n_order+1
10.              C(k,l,ms,mt) = …
11.              ij = idx(i,j)
12.              kl = idx(k,l)
13.              ELM_p(mp,ij,kl) = A(…)*B(…)*C(…)
14. end do(s)
15. do i= 1,n_vertex_max*n_var*(n_order+1)
16.   do j= 1,n_vertex_max*n_var*(n_order+1)
17.     tmp=fft4(ELM_p(:,i,j))
18.     do k=1,n_tor/2
19.       do m=1,n_tor/2
20.         ELM(i*n_tor+k,j*n_tor+m) += tmp(k+m)
21. end do(s)
```

Fig. 8 Illustration of the matrix element calculation. Lines 1–14 calculate the *ELM_p* temporary array, lines 15–21 transform these and store the results in the *ELM* matrix.

2.5.2. Specifying the memory layout of arrays

We would like to rely on the auto vectorization of the compiler to improve the performance, but unfortunately the compiler is not able to vectorize the code in the original form. According to the optimization report (generated using the *-qopt-report=5* flag), the compiler does not have sufficient information about the following topics:

- aliasing of pointer arrays,
- array alignment,
- stride for multidimensional arrays,
- read/write dependency of temporary variables.

We will address the first three points in this section. In JOREK, a derived type is defined that wraps all the buffers which are needed during matrix element calculation:

```
type thread_buffer
  real, dimension(:,:), pointer :: ELM
  real, dimension(:,:,:), pointer :: ELM_p
  ! …
end type
```

These buffers are defined as pointers, which can decrease the performance because the compiler has to consider that the pointers could be aliased. Adding the

contiguous attribute to the variables improves the execution time, but it is better to avoid pointers altogether. The only rationale for using pointers would be that pointers keep the array bounds when they are passed as assumed shape dummy arguments, but this feature is not used in JOREK. Therefore, we can safely change them to allocatable arrays.

An array of *thread_buffer* objects is allocated in the variable *thread_struct*, and thread *tid* has its buffers in *thread_struct(tid).* To avoid writing long expressions, these buffers are renamed in the *element_matrix_fft* subroutine by introducing new pointers:

```
real, dimension(:,:,:), pointer :: ELM_p
ELM_p => thread_struct(tid)%ELM_p

! do calculation using ELM_p.
```

This again poses difficulties for the optimizer. An elegant solution to avoid pointers for renaming would be the associate construct from Fortran 2003. Unfortunately, it is a rarely used feature and the Intel compiler cannot optimize it properly. A third option is to pass the buffers as subroutine arguments

```
call element_matrix_fft(…, thread_buff(tid)%ELM_p)
```

and to choose a short dummy argument name for the buffer:

```
subroutine element_matrix_fft(…, ELM_p)
#define DIM1 n_vertex_max*n_var*(n_order+1)
   real, dimension(n_plane, DIM1, DIM1) :: ELM_p
end subroutine
```

Dummy argument renaming was already applied to the *ELM* array in the original code, now it has been extended to 16 more arrays.

Passing the buffers as dummy arguments allows us to specify the array shape explicitly. This information is known at compile time and it helps the compiler to reason about the stride between different array elements, and about the alignment of the different columns of the arrays.

In order to specify the alignment of the start of the arrays, we use the *–align array64byte* compiler flag, which will ensure that all local and module variables are aligned at 64-byte boundaries. This way all our arrays are aligned correctly. We still have to add the

```
!$DIR ASSUME_ALIGNED ELM_p:64
```

compiler directive to indicate that the dummy arguments are also aligned. This information is not known to the compiler, because it depends on how exactly the function is called from other compilation units.

The previous steps inform the compiler about array alignment, array size, the stride of memory access, and ensure that no arrays are overlapping. These changes improve the performance by roughly 20% as shown in Fig. 9. We should note that most of the code is still not vectorized, for which the main reason is the assumed dependency between different variables. In the next section, we will investigate this point.

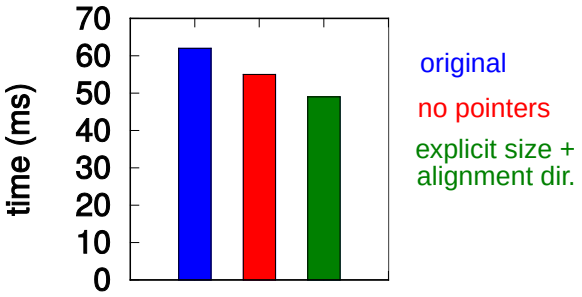

Fig. 9 The effect of different optimization techniques on the runtime of the subroutine *element_matrix_fft*. The red bar shows the performance when pointers are avoided. The green bar represents the results when the explicit array size and the alignment directive is specified for the temporary buffers (and no pointers are used).

After ensuring that the compiler has all the information about the memory layout of the arrays, we can now investigate the performance of the compute loops and the transform loops individually. Fig. 10 shows that two thirds of the execution time is spent in the compute loops part. The next section will focus on these loops.

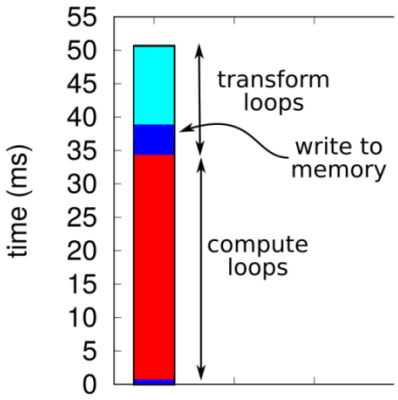

Fig. 10 The execution time of the two main groups of loops in the matrix element calculations subroutine.

2.5.3. Vectorization of the compute loops

The changes listed in the previous section can already help the compiler to vectorize simple kernels. Unfortunately, the *element_matrix_fft* subroutine is too complex for the compiler to vectorize. The usage of hundreds of temporary variables creates several assumed dependencies, which the compiler cannot resolve.

The dependency problem is easy to fix with the *SIMD* directive from the OpenMP standard. Using the *private* clause, we can define the temporary variables to be private for each vector lane. The *SIMD* directive also tells the compiler that there is no vector dependency inside the loop and it can be safely vectorized.

The function calls inside the loops can also prohibit vectorization. To evaluate different vectorization strategies, the function calls were temporarily switched off.

Several ways to optimize the compute loops were tried. Fig. 11 shows the execution time of these different variants. The most promising loop for vectorization is the *mp* loop, which has an iteration space of 32. This is an outer loop, and we first tried to transpose it so that it becomes an inner loop. This slightly increases the execution time (see the bar labelled as *mp1*), because certain variables are calculated redundantly. After adding the *SIMD* directive the execution time decreases significantly. Using temporary arrays, we can avoid some of the redundant calculation, but it did not improve the execution time (*SIMD mp2*). It is also possible to vectorize the *mp* loop as on outer loop using the SIMD directive (*SIMD mp3*). Alternatively, the loops over the Gaussian points (*SIMD ms mt*) can be also vectorized if we use the collapse clause of the SIMD directive (otherwise the iteration

count is too low). The last two variants have the best performance. Appendix 2.8 gives an overview of the different versions of the compute loops.

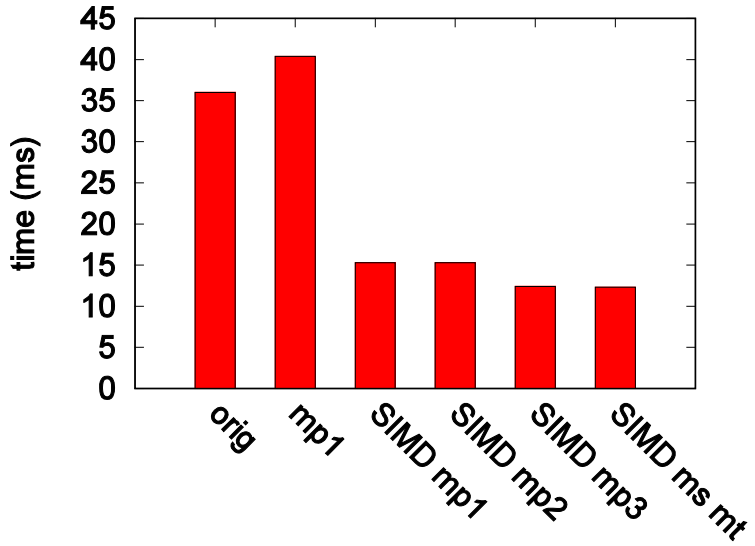

Fig. 11 Execution time of the compute loops (without function calls) using different vectorization options: *orig* and *mp1* have no vectorization, *SIMD mp1-mp2* have a vectorized inner *mp* loop, *mp3* is the outer loop vectorization. The last bar is the vectorized version of the loops over the Gaussian integration points. In Appendix 2.8, the different versions of the loops are listed.

Using the outer *mp* loop vectorization, the compute loop nest is completely vectorized, and the *mp* loop is completely unrolled. Still we achieve only a factor of three speedup instead of the expected factor of eight. The explanation of this discrepancy is that we are processing a large amount of data (see next section) and after vectorization, the cache bandwidth becomes the bottleneck. According to Intel vectorization advisor, the performance is somewhere between the L2 and L3 cache bandwidth limit.

The compute loop nest contains a small number of function calls. For the measurements presented in Fig. 10, these function calls were switched off. These calls can be vectorized using the OpenMP *declare simd* directive. We have to provide the additional *–vecabi=cmdtarget* compiler flag, so that the compiler generates AVX-512 vector code. For the functions *corr_neg_temp* and *corr_neg_dens* inlining further helps the performance. We place these subroutines in a file that is included into the same compilation unit where the *element_matrix_fft* subroutine is defined. This way the compiler can inline these functions. An alternative option would be the *–ipo* compiler flag, but that decreases the overall performance.

So far, we have only tested single core performance. In Fig. 12 we present the execution time of the optimized subroutine as we increase the number of threads (weak scaling, work/thread is kept constant). Due to the optimization discussed so far, the subroutine is approximately 2.3x faster than the original. The optimized compute loop (blue bars) takes roughly half of the execution time of the matrix element calculation (blue + green bars). While the capacity to perform flops increases linearly with the number of cores, the memory bandwidth does not. The L3 cache is shared among the cores as well. That is why the subroutine performance deviates from the ideal scaling, which would be a flat constant line in this graph.

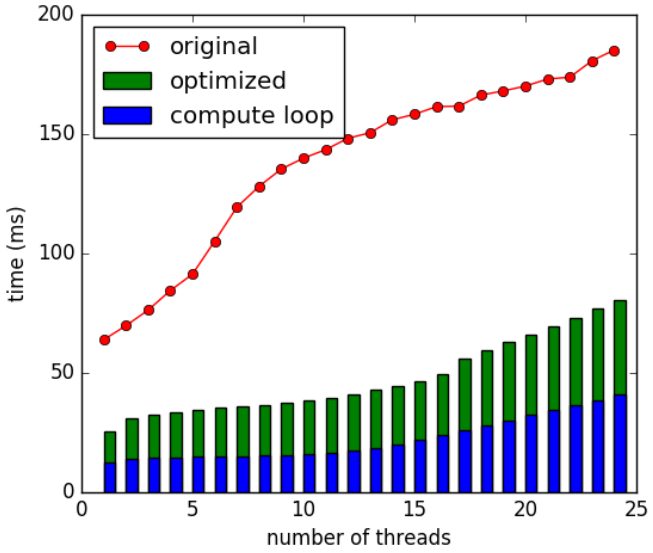

Fig. 12 Weak scaling of the *element_matrix_fft* subroutine: the amount of work/thread is kept fixed, no synchronization between the threads, one thread/core is used. The red curve shows the execution time of the original code without any optimization. The bars show the execution time with the optimizations in the compute loops: the blue part is the execution time of the compute loop itself and the green bars shows the execution time of the rest of the subroutine.

2.5.4. Memory and cache optimization

The compute loop (lines 1–14 in Fig. 8) calculates *ELM_p* and three more temporary arrays: *ELM_n, ELM_k, ELM_kn*. The total size of these arrays is 4*3 MiB. The second loop nest (lines 15–21 in Fig. 8) takes the temporary results, applies a Fourier transformation, and stores the results in the *ELM* array. The four temporary arrays ELM_p, ELM_n, ELM_k, ELM_kn are processed using four loop nests to add the contributions from these arrays to the single *ELM* output array, whose size is 27 MiB.

We should note that the Skylake processor has 1 MiB L2 cache and 33 MiB L3 cache shared among the 24 cores. On KNL, 1 MiB of L2 cache is shared between two cores in a tile. In our kernel each thread processes 12 MiB of temporary data, and generates 27 MiB of output data, therefore the memory bandwidth and cache usage becomes an important factor.

The dark blue patch in Fig. 10 shows the measured execution time of writing the results to the memory. It takes around 4 ms, which is close to the theoretical estimate of writing the data (a modified stream benchmark shows 7 GiB/sec single thread write bandwidth in the Skylake partition).

In order to improve data locality while writing the results, the four individual loop nests were merged for the *ELM_p*, ELM_n,… arrays. The *i* and *j* loops were transposed and the inner loops that scatter the data from *tmp* to *ELM* were also transposed. These changes lead to a few ms speedup.

To improve cache usage, instead of filling the *ELM_p(:,:,:)* array completely (3 MiB) before adding it to *ELM(:,:),* we filled only the *ELM_p(:,:,i)* slice (192 kiB) and added it to *ELM(:,:).* This reduces the data size that needs to be cached by a factor of 16. The cache optimized subroutine has much better weak scaling properties, as we can see in Fig. 13. Using a single multicore CPU with 24 threads, the optimized code has a factor of four speedup compared to the original. During these optimization steps, the last two dimensions of the *ELM_p* arrays were transposed to have a favorable access pattern in the compute loops.

This is just an initial step in optimizing cache usage. The scaling of the compute loops (Fig. 12) indicates that there is still more room for cache and memory optimization.

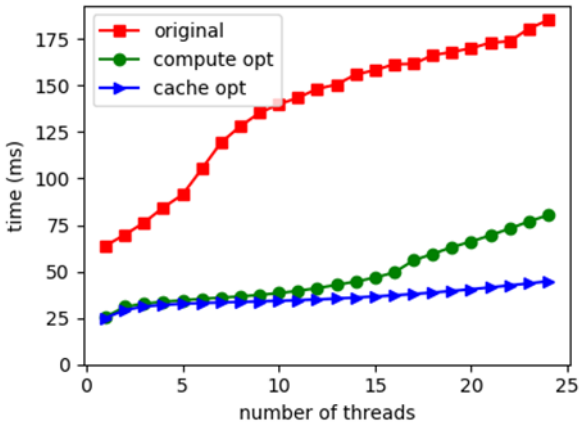

Fig. 13 Weak scaling of the element matrix construction subroutine. The red and green curves show the same data as in Fig. 12. The blue curve adds cache optimization on top of it, which improves the scaling.

2.5.5. Vectorization on KNL

The KNL and Skylake architectures both have the AVX-512 vector instruction set. The optimizations presented in the previous section also improve the performance on KNL by a factor of three, as we can see in Fig. 14. The most important change in this case is the compute loop optimization. The cache architecture of KNL is different than the one of Skylake, and unfortunately KNL does not benefit much from the simple cache optimization that was described in the previous section.

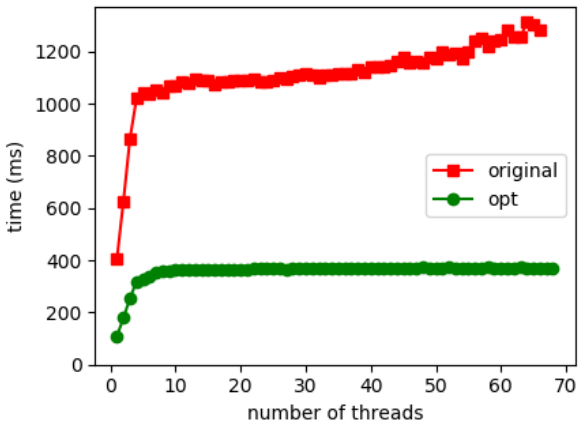

Fig. 14 Weak scaling of the *element_matrix_fft* subroutine: the execution time is shown for the original (red) and the optimized (green) version.

2.5.6. Integrating the changes into the main code

The results presented in the previous sections were measured using a separate test program. After integrating the changes into the JOREK code, we measured the execution time of the matrix construction in detail using VTune (Fig. 15). Comparing the purple bars between the left and right side, we can see that the execution time of *element_matrix_fft* is improved by a factor of 2.3x–3.6x (depending on the number of threads). The results from the *element_matrix_fft* subroutine are accumulated into the *A_glob* array. Updating *A_glob* needs synchronization among the threads. An atomic update (introduced in Section 2.4.1) has a large overhead even if we only have one thread (Fig. 15, left side). The execution time of the atomic synchronization is longer than the execution time of the optimized *element_matrix_fft* subroutine. Therefore we considered additional implementations of the synchronization.

M. Hölzl proposed a variant of the synchronization using critical section and a temporary buffer, similar to the one presented in Section 2.4.1, but better aligned with the loop structure in the *construct_matrix* subroutine. The new synchronization method has no overhead for a small number of threads, and although it has a very long waiting time when we use a large number of threads (see orange bars in the right side of Fig. 15), it is still better then previous implementations. By carefully choosing the number of MPI tasks and OpenMP threads, the new variant with a critical section becomes faster than the atomic synchronization. The best configuration for the *ntor*=17 test case on Skylake is to use 4 MPI tasks/node and 12 threads/MPI task. This way the synchronization overhead is minimized, and the matrix construction is faster by up to a factor of two (see Fig. 16).

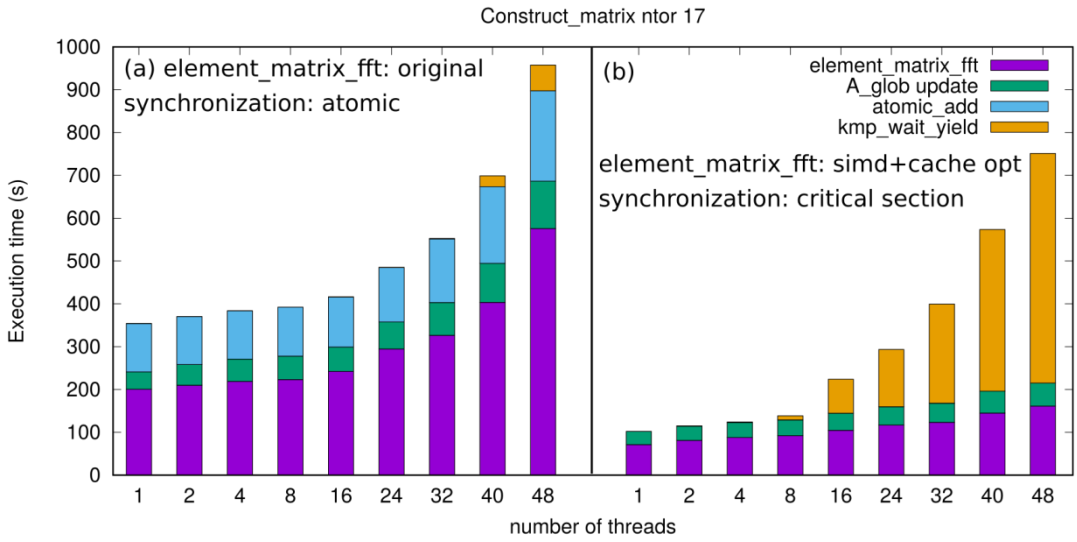

Fig. 15 Execution time of the matrix construction, the total execution time summed over all threads within a node is shown (*ntor*=17 test case, 1 MPI task/node). Left side: original version of the *element_matrix_fft* with atomic synchronization, right side: optimized *element_matrix_fft* with improved critical section synchronization.

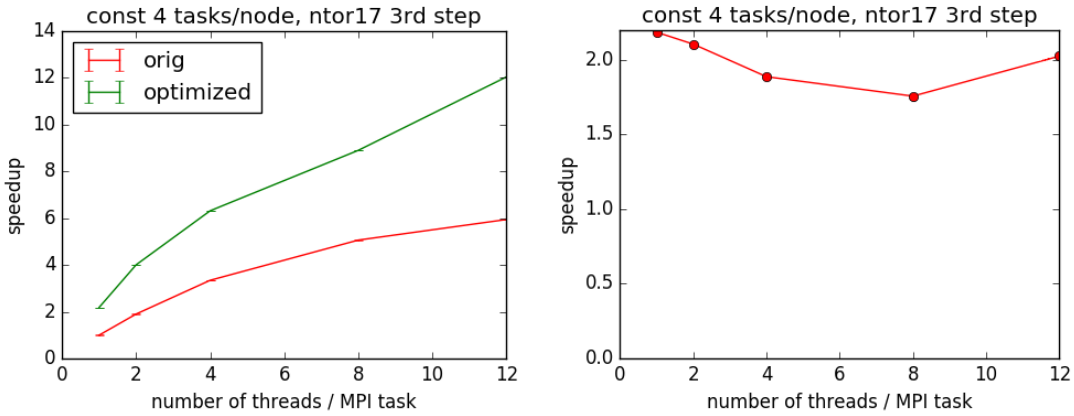

Fig. 16 Strong scaling of the matrix (*ntor*=17 test case, 4 MPI tasks/node). The left side shows the speedup with respect to the number of threads for the original and the optimised code. The right hand side presents the ratio between the green and red curves of the left hand side.

It might be possible to avoid synchronization during matrix construction altogether, since it is only the neighboring elements that write to the same memory locations. One could process the elements in several batches, with the batches set up in a way that their memory write patterns do not overlap. This idea is used in certain PIC codes during the charge assignment step (Moschuering, 2018).

2.6. *Summary*

The JOKLA project improved the performance of the JOREK code on the KNL and Skylake architectures. The OpenMP scaling of the matrix construction part of the code was investigated. Synchronization using atomic directives scales better for a large number of threads, but its overhead is comparable to the actual computation that needs to be synchronized. A modified version of the critical section synchronization together with a properly chosen hybrid MPI/OpenMP configuration gives the best performance.

It was found that the linear equation solver is also sensitive to the MPI/OpenMP execution configuration, and the best performance can be achieved if we limit the number of threads per node that is used by the solver.

The work continued by improving the vectorization of the matrix construction. In a separate test bed, the matrix construction subroutine was vectorized, and the data locality was improved, which lead to a 4x speedup for this subroutine on Skylake and a 3x speedup on KNL. After vectorization the code performance is limited by memory and cache bandwidth. The same changes, when integrated into the main code led up to a factor of two speedup of the matrix construction part. Future work should focus on further decreasing the synchronization overhead and the amount of copying the results between temporary buffers. The execution time of the JOREK code is generally 2.2–2.9 times longer on KNL than on Skylake in a node to node comparison.

2.7. *References*

Hölzl, M. (2018). ELM simulation setup. IPP.

Huysmans, G. T., & Czarny, O. (2007). MHD stability in X-point geometry: simulation of ELMs. *Nuclear Fusion, 47*(7), 659.

Moschuering, N. (2018, March). *Report on HLST project PICOPT.*

2.8. *Appendix*

Different methods of vectorizing the *element_matrix_fft* subroutines are listed here.

Name	Pseudo code	Remarks
MP1	**do** ms=1, n_gauss; **do** mt=1, n_gauss **do** i=1,n_vertex_max; **do** j=1,n_order+1 **do** k=1,n_vertex_max; **do** l=1,n_order+1 !$omp simd private(….) **do** mp=1,n_plane A(mp, ms, mt) = … B(i,j,ms,mt) = … rhs(mp,ij) = A(mp,ms,mt)*B(i,j,ms,mt) C(k,l,ms,mt) = … ELM(mp,ij,kl) = A*B*C **end do(s)** ...	Loops transposed: *mp* loop in the innermost position. Repeated calculation of *A* and *B*. More than 300 SIMD private variables.
MP2	**do** ms=1, n_gauss; **do** mt=1, n_gauss **do** mp=1,n_plane A(mp, ms, mt) = … **end do** **do** i=1,n_vertex_max; **do** j=1,n_order+1 B(i,j,ms,mt) = … **do** mp=1,n_plane rhs(mp,ij) = A(mp,ms,mt)*B(i,j,ms,mt) **end do** **do** k=1,n_vertex_max; **do** l=1,n_order+1 C(k,l,ms,mt) = … **do** mp=1,n_plane ELM(mp,ij,kl) = A*B*C **end do**(s)	The *mp* loop is split into smaller parts. Inner loop vectorization without repeated calculation. Larger memory usage: 147 temporary scalars replaced with temporary arrays of dimension(32). 170 SIMD private variables.
MP3	**do** ms=1, n_gauss; **do** mt=1, n_gauss !$omp simd private(...) **do** mp=1,n_plane A(mp, ms, mt) = ... **do** i=1,n_vertex_max; **do** j=1,n_order+1 B(i,j,ms,mt) = ... rhs(mp,ij) = A(mp,ms,mt)*B(i,j,ms,mt) **do** k=1,n_vertex_max; **do** l=1,n_order+1 C(k,l,ms,mt) = ... ELM(mp,ij,kl) = A*B*C **end do(s)**	Simple, original loop structure is kept. More than 300 SIMD private variables. Good performance.
MSMT	!$omp simd collapse(2) private(...) **do** ms=1, n_gauss; **do** mt=1, n_gauss **do** mp=1,n_plane A(mp, ms, mt) = ... **do** i=1,n_vertex_max; **do** j=1,n_order+1 B(i,j,ms,mt) = ... rhs(mp,ij) = A(mp,ms,mt)*B(i,j,ms,mt) **do** k=1,n_vertex_max; **do** l=1,n_order+1 C(k,l,ms,mt) = ... ELM(mp,ij,kl) = A*B*C **end do(s)**	Parallelization over the Gaussian integration points. Loops collapsed to increase iteration count. More than 300 SIMD private variables. Non-unit stride memory access. Good performance.

3. Final report on the AMGBOUT project

3.1. *Introduction*

First principles simulations of Scrape-Off Layer (SOL) turbulence provide important insight into the mechanisms governing plasma exhaust. Over the last decade, the BOUT++ code has proven itself able to tackle several aspects of the problem, thanks to its robust and user-friendly approach to defining models and geometry. However, to achieve full 3D simulations of saturated turbulence in realistic geometry, BOUT++ needs further performance optimizations.

In this project we consider an algebraic multigrid scheme in order to perform a pseudo 3D (2D+1D) Laplace inversion. The solver should be using BOUT++ conventions and data types, and should be flexible enough to allow a future parallelization. This is a natural extension of the implementation of multigrid algorithms in 2D solvers, performed by the HLST in 2015. The modularity of BOUT++ means that only the Laplace solver needs to be modified (only ~1000 lines out of ~85,000). Algebraic multigrid is an ideal tool for the required Laplace inversion, as its algorithm uses only information about the operator to be inverted, rather than information about the geometry of the problem. This means that the algebraic multigrid scheme is applicable in BOUT++'s toroidal geometry, and can be easily extended to work in more complicated geometries in the future.

Due to the more onerous nature of the pseudo 3D problem, it is important to extend the parallelization of the code to the "toroidal" direction which will ensure that the extra physics are implemented without a loss of performance. The absence of such a parallelization is a legacy design flaw, which could be removed by making use of the MPI protocols already implemented in the other two directions. In addition, parallelization in the third direction will enable full exploitation of the power of the multigrid approach in medium sized simulations which are important for parameter scans. For these cases, the present 2D approach saturates in efficiency at a relatively small number of processors. However, this 2D approach has difficult to handle realistic boundary condition. To overcome this difficulty, we consider a 3D solver without any modification which may have pure performance.

Developing an algebraic multigrid method requires difficult and time consuming research, which makes it reasonable to start with an available algebraic multigrid library. As a starting point, we will use a solver from the PETSc library, which allows us to choose from several direct and iterative solvers, including Krylov subspace methods with a multigrid preconditioner. We use the GAMG preconditioner from the PETSc library and investigate its performance with different settings to find the optimal one for BOUT++. In addition, we test other existing algebraic multigrid methods, namely BoomerAMG from hypre and ML from Trilinos, which can be used within the PETSc library. In the long term, we plan to develop our own algebraic multigrid algorithm. Based on the knowledge from our previous implementations of multigrid solvers, we expect to achieve a good performance of such a solver in BOUT++ as well.

3.2. *The BOUT++ code*

BOUT++ is a flexible framework for finding the solution of partial differential equations with differential operators and geometry relevant to plasmas. The code was developed as a toolbox for parallel computing in order to reduce duplication of effort and allow quick development and testing of new models. It is a collection of useful data types and associated routines, and occupies a middle ground between problem-specific codes and general libraries such as PETSc, Trilinos, Overtuen, Chombo, etc.

BOUT++ has the following features:

- Finite difference initial value code in 3D

- Implicit and explicit time integration methods
- Coordinate systems set in metric tensor components
- Handles topology on multiple X-points
- Written in C++, quite modular design
- Open source, available on github

The code has a wide user base and widespread applications, mostly oriented towards pedestal and Scrape-Off Layer physics. The main use at CCFE is related to SOL and divertor physics, in particular 3D turbulence and filament simulations in the framework of reduced drift fluid models. Further distinctive features of the code are the interchangeability of different physics models (which can go beyond fluid approximations) accessible via a user friendly modular interface, flexible mesh handling with the possibility of implementing two X-points, a closed field line region and two private flux regions, all while maintaining good computational performance. The code was used to perform experimental analysis of 3D filaments in the TORPEX machine and in MAST. A preliminary study of turbulent fluxes in ISTTOK has been carried out and future applications will extend this and also include COMPASS. The code could potentially be used to simulate 3D transport in the SOL of JET, AUG and TCV.

3.3. *Laplace inversion in the BOUT++ code*

To solve 3D tokamak-shaped problems in BOUT++, the domain is sliced in the poloidal direction and the second order PDEs are solved on each slice (*x*-*z* space) as shown in Fig. 17 (2D+1D decoupling). Each slice can be transformed by the conformal mapping g_y^{2D} to a reference domain (rectangular) with periodic boundary conditions along the *z*-direction as shown in Fig. 18. To develop the full 3D solver of the second order PDEs, the whole domain can be transformed by the conformal mappings g^{3D} to a reference domain (rectangular parallelepiped) with proper boundary conditions along each direction as shown in Fig. 18.

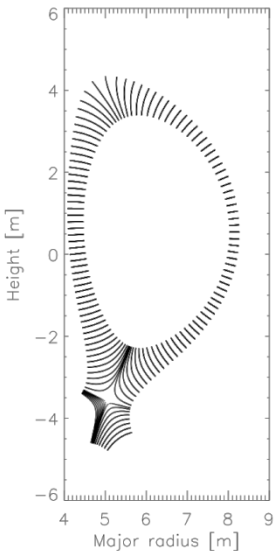

Fig. 17 The coordinates in the poloidal plane of a tokamak in the BOUT++ code. The *x*- and *y*- coordinates lie in the poloidal plane. The solid lines denote the *x*-coordinate lines.

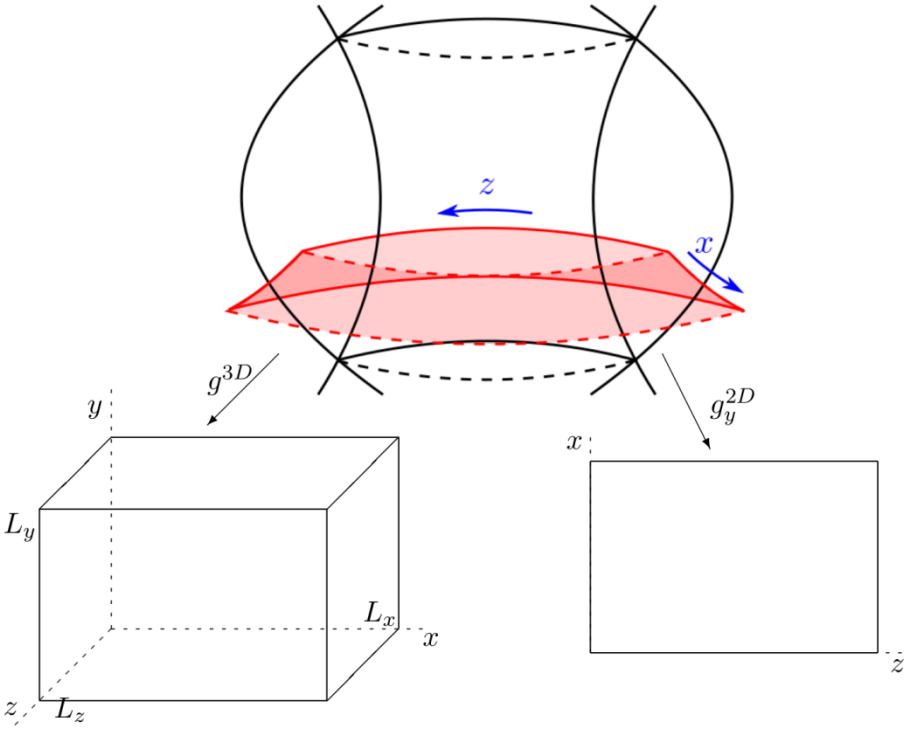

Fig. 18 Computational domain and its conformal mappings g^{3D} and g_y^{2D}.

Now we consider a discretization for the inversion of the second order PDEs on the reference domain (rectangular parallelepiped for 3D or rectangle for 2D) with the conformal transform g (g_y^{2D} or g^{3D}). The equation

$$d\nabla_\perp^2 f + \frac{1}{c_1}\nabla c_2 \cdot \nabla_\perp f + af = b$$

where a, c_1, c_2 and d are constants and b is a function, must be inverted for the potential f. The variables x, y and z are the flux coordinate, the poloidal coordinate and the toroidal coordinate, respectively, and the g terms are elements of the metric tensor. We then have the following expressions

$$\begin{aligned}\nabla_\perp^2 f = {} & G_1\frac{\partial f}{\partial x} + G_3\frac{\partial f}{\partial z} + g^{11}\frac{\partial^2}{\partial x^2} + 2g^{13}\frac{\partial^2 f}{\partial x \partial z} + g^{33}\frac{\partial^2 f}{\partial z^2} \\ & + \left(G_2 - \frac{1}{J}\frac{\partial}{\partial y}\left(\frac{J}{g_{22}}\right)\right)\frac{\partial f}{\partial y} + \left(g^{22} - \frac{1}{g^{22}}\right)\frac{\partial^2 f}{\partial y^2} + 2g^{12}\frac{\partial^2}{\partial x \partial y} + 2g^{23}\frac{\partial^2 f}{\partial y \partial z},\end{aligned}$$

$$\begin{aligned}\nabla c \cdot \nabla_\perp f = {} & \left(g^{11}\frac{\partial c}{\partial x} + g^{13}\frac{\partial c}{\partial z}\right)\frac{\partial f}{\partial x} + + \left(g^{13}\frac{\partial c}{\partial x} + g^{33}\frac{\partial c}{\partial z}\right)\frac{\partial f}{\partial z} \\ & + g^{12}\frac{\partial c}{\partial y}\frac{\partial f}{\partial x} + g^{23}\frac{\partial c}{\partial y}\frac{\partial f}{\partial z} + \left(g^{12}\frac{\partial c}{\partial x} + \left(g^{22} - \frac{1}{g^{22}}\right)\frac{\partial c}{\partial y} + g^{23}\frac{\partial c}{\partial z}\right)\frac{\partial f}{\partial y},\end{aligned}$$

where $G_k = \sum_{ij} g^{ij}\Gamma_{ij}^k$.

If we arbitrarily set the cross terms connected with the y derivatives g^{12} and g^{23} to zero, we have 2D expressions without the red terms of the original expressions, i.e., only the x and z derivatives remain. By using this approach – approximating the second order operator by removing the cross terms connected with the parallel derivatives – the inversion is performed on independent planes at constant poloidal angles (we call this a 2D-1D approach). This approach is robust because it solves several 2D problems instead of a 3D problem which would in general need more time

to be solved. However, this does not work close to the *X*-point because the values g^{12} and g^{23} are too large to be neglected. To overcome this difficulty and to keep the robustness of the 2D-1D approach, we can combine the 3D solver near the *X*-point with the 2D solver far away from the *X*-point.

In the original version of BOUT++, the solver was a direct solver: Fast Fourier Transforms in the toroidal direction reduced the problem to the inversion of a tridiagonal matrix, which was then done with the Thomas algorithm. The toroidal direction was not parallelized since FFTs are best performed on contiguous data, which would require communication over the entire grid in order to rearrange storage. The 2015 multigrid version inherits this constraint of the 2D solver. The incomplete formulation of the Laplacian problem – due to the lack of parallel derivatives – yields another important limitation of the current approach: the planes are decoupled. This implies, for example, that 2D solvers cannot implement parallel boundary conditions, so the sheath boundary conditions on the potential, which are critical to the correct description of the SOL, cannot be applied. Moreover, the integrability conditions on 2D solvers restrict the possible perpendicular boundary conditions; in particular they do not allow all-Neumann boundary conditions. This imposes spurious constraints on the parallel variation of the solution, which may then fail to be a good approximation to the solution of the 3D problem, introducing unphysical behavior. This issue is of paramount importance in the SOL, as the presence of sheath boundaries induces strong parallel variation of the plasma parameters which exposes the limitations of decoupled 2D solvers.

3.4. *Discretization of the 2nd order PDEs*

We consider the discretization of the 2nd order PDEs on a 2D and 3D reference domain. A cell-centered finite difference discretization is used in BOUT++ for 2D and 3D. We consider a parallelepiped domain as the 3D reference domain as shown in Fig. 19. In BOUT++, the mesh size along the *z*-direction is fixed as h_z and the mesh sizes along the other directions (h_x and h_y) vary. In the cell centered scheme, the values of the functions are defined at the center of each cell, with a size of $h_z \times h_x \times h_y$, and are denoted as *f(j,i,k).*

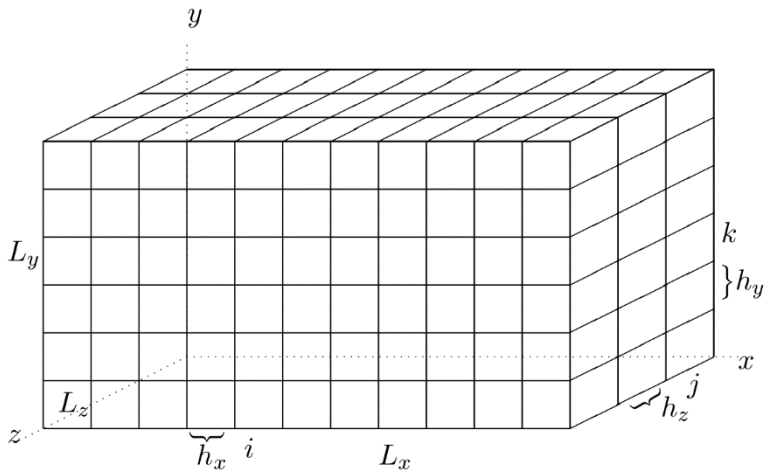

Fig. 19 The 3D reference domain.

The number of nonzero elements of the discretized system is 19 in 3D and 9 in 2D. To set these nonzero elements of the matrix for the discretized system, we consider a local numbering as shown in Fig. 20. The center cell has number 9 in the 3D case and number 4 in the 2D case.

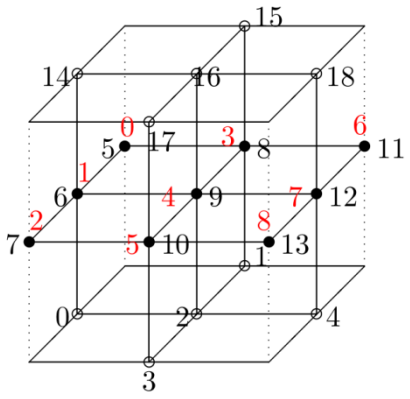

Fig. 20 Local numbering for the matrix for the discretized system near the center cell, which is number 9 in the 3D case and number 4 in the 2D case (in red).

For the cell-centered finite discretization, we have the following values for each element of the matrix for the 3D solver (A^2 for the 2D solver):

```
A0 = dydz*vol/4.0,                    A1 = dxdy*vol/4.0,
A2 = (ddy - dyd/2.0)*vol,
A3 = -dxdy*vol/4.0,                   A4 = -dydz*vol/4.0,
A5(A0^2) = dxdz*vol/4.0,              A6(A1^2) = (ddx - dxd/2.0)*vol,
A7(A2^2) = -dxdz*vol/4.0,             A8(A3^2) = (ddz - dzd/2.0)*vol,
A9(A4^2) = (a - 2.0*(ddx + ddy + ddz))*vol,
A10(A5^2) = (ddz + dzd/2.0)*vol,      A11(A6^2) = -dxdz*vol/4.0,
A12(A7^2) = (ddx+dxd/2.0)*vol,        A13(A8^2) = dxdz*vol/4.0
A14 = -dydz*vol/4.0,                  A15 = -dxdy*vol/4.0,
A16 = (ddy+dyd/2.0)*vol,
A17 = dxdy*vol/4.0,                   A18 = dydz*vol/4.0
```

where

```
vol = hx*hy*hz (hx*hy)
gt11 = g11,              gt12 = g12,    gt13 = g13,
gt22 = g22 - 1.0/g22,    gt23 = g23,    gt33 = g33
ddJ = [J(k2+1)/g22(k2+1) - J(k2-1)/g22(k2-1)]/2./hy/J,
ddx_C = (C2(i2+1)-C2(i2-1))/2./hx/C1
ddy_C = (C2(k2+1)-C2(k2-1))/2./hy/C1
ddz_C = (C2(j2+1)-C2(j2-1))/2./hz/C1
ddx = D*gt11/hx/hx,     dxdy = 2.0*D*gt12/hx/hy
ddy = D*gt22/hy/hy,     dxdz = 2.0*D*gt13/hx/hz
ddz = D*gt33/hz/hz,     dydz = 2.0*D*gt23/hy/hz
dxd = (D*G1+gt11*ddx_C + gt12*ddy_C + gt13*ddz_C)/hx
dyd = (D*(G2-ddJ)+gt12*ddx_C + gt22*ddy_C + gt23*ddz_C)/hy
dzd = (D*G3+gt33*ddz_C +gt13*ddx_C + gt23*ddy_C)/hz
```

3.5. *The PETSc library*

The first part of this section closely follows the PETSc User's Manual [1], p. 21.

The Portable, Extensible Toolkit for Scientific Computation (PETSc) has successfully demonstrated that the use of modern programming paradigms can ease the development of large-scale scientific application codes in Fortran, C, C++, and Python. Begun over 20 years ago, the software has evolved into a powerful set of

tools for the numerical solution of partial differential equations and related problems on high-performance computers.

PETSc consists of a variety of libraries (similar to classes in C++). Each library manipulates a particular family of objects (for instance, vectors) and the operations one would like to perform on those objects. The objects and operations in PETSc are derived from extensive experience with scientific computation. PETSc's modules deal, amongst other things, with

- index sets (IS), including permutations, for indexing into vectors, renumbering, etc;
- vectors (Vec);
- matrices (Mat) (generally sparse);
- over thirty Krylov subspace methods (KSP);
- dozens of preconditioners, including multigrid, block solvers, and sparse direct solvers (PC);
- managing interactions between mesh data structures, vectors and matrices (DM);
- nonlinear solvers (SNES);
- time steppers for solving time-dependent (nonlinear) PDEs, including support for differential algebraic equations, and the computation of adjoints (sensitivities/gradients of the solutions) (TS)

Each module consists of an abstract interface (a set of calling sequences) and one or more implementations using particular data structures. Thus, PETSc provides clean and effective codes for the various phases of solving PDEs, with a uniform approach for each class of problems. This design enables an easy comparison and use of different algorithms (for example, to enable experimentation with different Krylov subspace methods, preconditioners, or truncated Newton methods). Hence, PETSc provides a rich environment for modeling scientific applications as well as for rapid algorithm design and prototyping.

The libraries allow for easy customization and extension of both algorithms and implementations. This approach promotes code reuse and flexibility, and separates the issues of parallelism from the choice of algorithms. The PETSc infrastructure creates a foundation for building large-scale applications.

It is useful to consider the interrelationships among different pieces of PETSc. In Ref. [1], page 22 there is a diagram of some of these pieces. The figure illustrates the library's hierarchical organization, which enables users to employ the level of abstraction that is most appropriate for a particular problem.

Among the many components of the PETSc library, we need to use the solvers for linear systems, i.e., the Krylov Subspace Methods (KSP) component. In order to use this component, we also need to use the operators (Mat), vectors (Vec), and preconditioners (PC) libraries of PETSc. In this project, we focus on using the algebraic multigrid preconditioner for GMRES. Among the many algebraic multigrid methods, GAMG is PETSc's native AMG framework. Two good third party solvers, BoomerAMG from hypre and ML from Trilinos, can be called within PETSc. To use these third party solvers, one have to enable them with –download-hyper for BoomerAMG in Hypre and/or –download-ml for ML in Trilinos in configuring for installing PETSc. Furthermore, PETSc provides a general framework for the multigrid method (MG) for geometric and algebraic multigrid methods.

3.6. *Using the PETSc library in BOUT++*

We will use the parallel vector type of PETSc (which is MPI-based) and thus need to implement a conversion routine from the PETSc vector type to the BOUT++ vector type and vice versa. PETSc provides a variety of matrix implementations because a single matrix format will not be appropriate for all problems. Currently, PETSc supports dense storage and compressed sparse row storage (both in sequential and

parallel versions), as well as several specialized formats. We will use the parallel sparse matrices with the AIJ format as it is required for GAMG.

In order to configure BOUT++ with PETSc, one has to issue

```
$ ./configure --with-petsc
```

in the BOUT++ root directory and include the header file <petscksp.h>, which includes the other header files <petscpc.h>, <petscmat.h>, and <petscvec.h>.

In this project we focus on linear solvers which are found in the KSP module. It provides uniform and efficient access to all linear system solvers of the package, including parallel and sequential, as well as direct and iterative solvers. In order to solve a nonsingular system of the form

$$Ax = b,$$

where A denotes the matrix representation of a linear operator, b is the right-hand-side (RHS) vector, and x is the solution vector, we have to convert the vector data from the BOUT++ vector format to the PETSc vector format. We use the index set to assign the local ordering to the global ordering. Then we generate a matrix A in the PETSc (MPI) AIJ format which is the only format accepted by GAMG.

KSP uses the same calling sequence for both, the direct and the iterative solution of a linear system. We set the solver by choosing a combination of a Krylov subspace method and a preconditioner. To set the type of the solver and the preconditioner with KSP, we use KSPSetType(KSP **ksp**, KSPType **ksptype**) and PCSetType(PC **pc**, PCType **pctype**) after creating a KSP solver context **ksp** with KSPCreate(MPI Comm **comm**, KSP ***ksp**) and getting a PC preconditioner context **pc** with KSPGetPC(KSP **ksp**, PC ***pc**). Afterwards, we set the linear system with KSPSetOperators(KSP **ksp**, Mat **Amat**, Mat **Pmat**), where **Amat** represents the matrix of the linear system and **Pmat** represents the matrix for the preconditioner which may be the same as **Amat**.

Some solvers of the PETSc only can be used only serially or by using external libraries. One can use external libraries in the PETSc if they are linked during the installation of the PETSc. Because some users prefer to use pre-installed libraries often avoid using external libraries in the PETSc. we consider first the generic solvers of the PETSc. The LU factorization as the direct solver (**ksptype** = PREONLY and **pctype** = PCLU) and Restarted GMRES with the incomplete LU decomposition preconditioner (**ksptype** = KSPGMRES and **pctype** = PCILU) are very good candidates for verifying the correctness of the generation of the discretized linear system and the solvability of the system. Unfortunately, these methods cannot be used in parallel. Because of this we choose the parallel Restarted preconditioned GMRES (PGMRES) with the block Jacobi preconditioner (**ksptype** = KSPGMRES and **pctype** = PCBJACOBI). The block Jacobi preconditioner with one block on each core is the default preconditioner of the PGMRES.

For the BOUT++ problem with a parallelization along the *x*-direction only and a periodic boundary condition along the *z*-direction, the generated discretized matrix has the following form:

$$A = \begin{pmatrix} A_z^1 & B_z^1 & 0 & 0 & \cdots & 0 & 0 \\ C_z^1 & A_z^2 & B_z^2 & 0 & \cdots & 0 & 0 \\ 0 & C_z^2 & A_z^3 & B_z^3 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & A_z^{n-1} & B_z^{n-1} \\ 0 & 0 & 0 & 0 & \cdots & C_z^{n-1} & A_z^n \end{pmatrix}$$

where A_z^k, B_z^k, and C_z^k follow the nonzero element pattern

$$\begin{pmatrix} a_{11}^k & a_{12}^k & 0 & 0 & \cdots & 0 & a_{1m}^k \\ a_{21}^k & a_{22}^k & a_{23}^k & 0 & \cdots & 0 & 0 \\ 0 & a_{32}^k & a_{33}^k & a_{34}^k & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{(m-1)(m-1)}^k & a_{(m-1)m}^k \\ a_{m1}^k & 0 & 0 & 0 & \cdots & a_{m(m-1)}^k & a_{mm}^k \end{pmatrix}.$$

The inverse of the sub-matrice A_z^k is a dense matrix due to the nonzero elements a^k_{m1} and a^k_{1m}. If we reduce the block size to half the number of *z*-directional cells (*m*), the block matrix becomes a tri-diagonal matrix and the inverse of the block matrix is also a tri-diagonal matrix.

We use the GAMG preconditioner with **ksptype** = KSPGMRES and **pctype** = PCGAMG. Among many available options for the GAMG preconditioner, we test different number of smoothing for aggregation interpolation operators affects for convergence performance and the required number of iteration of the PGMRES. The default number of smoothings for aggregation is **one** and users can set other numbers with PCGAMGSetNSmooths(pc, number). We also use the ML in Trilinos with **pctype** = PCML. The usage of the BoomerAMG is slightly different with the ML because the Hypre library support several solvers including the BoomerAMG. To use the BoomerAMG solver in the Hypre, users have to set **pctype** = PCHYPRE and call PCHYPRESetType(pc, "boomeramg").

To run a program in the BOUT++, one sets options in BOUT.inp which is consisted several parts including mesh, coefficients of PDE, and solvers. We set the algebraic multigrid solver under **[laplace]** and **[petscamg]** as shown in Fig. 21. In here, we set solver and other options such as solvertype = hypre and multigridlevel= 6.

```
myg = 1
mxg = 1
solution = f(x,y,z); input = f(x,y,z)
[mesh]
symmetricGlobalX = true
nx = 34 ; ny = 4; nz = nx-2*mxg
dx = 0.0036363636363636364/(nx-2*mxg)
dy = 2.*pi/ny
dz = 2.*pi/nz/600
g11 = ; g22 = ; g33 = ; g12 = ; g13 = ; g23 = ; g_11 = ; g_22 = ;
g_33 = ; g_12 = ; g_13 = ; g_23 = ; J = ; Bxy = ; G1_11 = ; G1_22 = ;
G1_33 = ; G1_12 = ; G1_13 = ; G1_23 = ; G2_11 = ; G2_22 = ; G2_33 = ;
G2_12 = ; G2_13 = ; G2_23 = ; G3_11 = ; G3_22 = ; G3_33 = ; G3_12 = ;
G3_13 = ; G3_23 = ; G1 = ; G2 = ; G3 = ;
Lx = 0.0036363636363636364
Lz = 0.0209439510239320
[laplace]
type = petscamg
flags = 0
[petscamg]
solvertype = hypre
multigridlevel = 6
```

Fig. 21 An example of BOUT.inp to set options for BOUT++.

3.7. *Verification and numerical results*

J. Omotani from CCFE provided a test case with a rectangular domain as the reference domain, which has a uniform mesh in each direction with $nz = nx = 2^k$ ($k = 5, 6, \ldots, 15$) and $ny = 16$. First, we evaluated the discretization error in the l^2 and l^∞ norms and report the result in Fig. 22 for $k = 5, 6, \ldots, 10$.

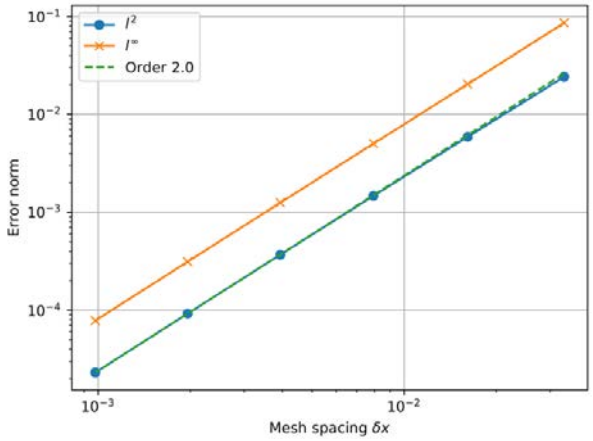

Fig. 22 Numerical discretization errors with respect to the mesh spacing δx.

The convergent factor of the discretization errors is two which is the theoretical value for the centered finite different discretization. It confirms the validity of the generation discretization system and the data conversion from BOUT++ to PETSc and vice versa.

From now on, we investigate the performance behavior of the solvers including a direct solver. First, we did tests on a single core on which one can use the LU decomposition as a direct solver (direct). We also tested PGMRES with a block Jacobi preconditioner (GMRES) and PGMRES with a GAMG preconditioner with default options (GAMG), **two** and **four** aggregation smoothing steps (GAMG2 and GAMG4), ML (ML), and BoomerAMG (Hypre). We report the average solution times and the ratio of the solution times of the consecutive levels of the multigrid method in Table 3 and plot the solution time with respect to the number of the DoF in Fig. 23.

Table 3 Average solution times and ratio of the solution times of consecutive levels of the multigrid method on a single core.

DoF	Direct	GMRES	GAMG	GAMG2	GAMG4	ML	Hypre
32^2	0.003	0.15	0.03	0.02	0.03	0.02	0.007
64^2	0.014	0.85	0.11	0.09	0.09	0.07	0.012
128^2	0.098	6.43	0.54	0.43	0.41	0.30	0.081
256^2	0.682	87.68	2.50	1.95	1.90	1.27	0.410
512^2	5.045	1275.77	14.01	10.20	10.16	6.69	2.337
1024^2	42.523	$*$	73.11	54.71	51.99	33.11	10.656
2048^2	$*$	$*$	329.75	251.34	251.49	147.66	45.155
Ratio	6.88	9.58	4.85	4.69	4.64	4.47	4.33

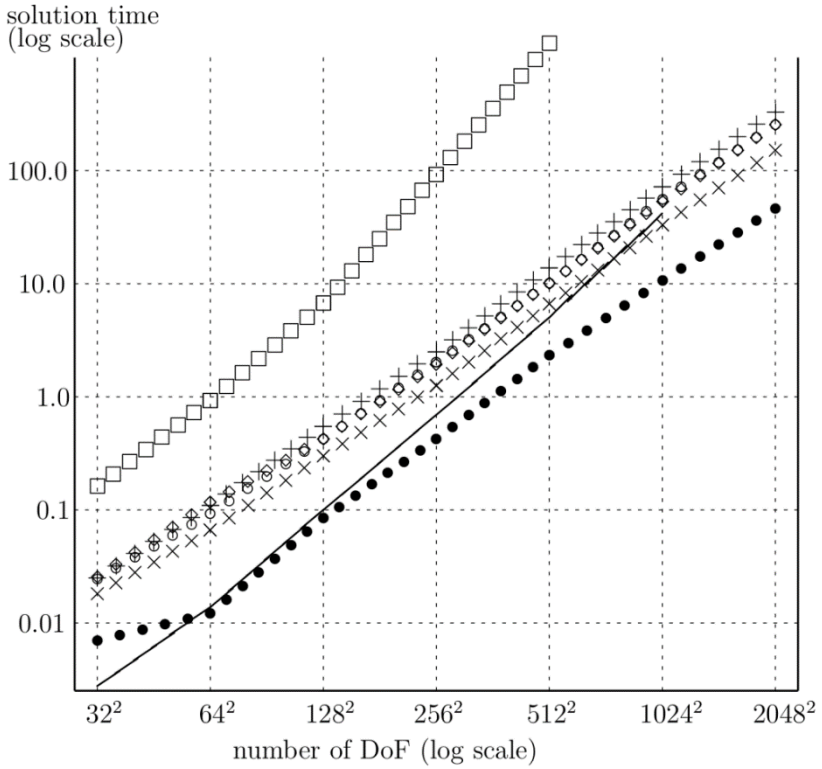

Fig. 23 Average solution times in seconds of several solvers on a single core as a function of the DoF. The solid line stands for a direct solver (LU decomposition), □ for GMRES with a block Jacobi preconditioner, PGMRES with a GAMG with one aggregation (+), GAMG with two aggregations (○), GAMG with four aggregations (◊), the ML solver from the Trilinos package (×), and the BoomerAMG solver from the Hypre package (•).

The direct method is the fastest solver for a problem of 32^2 DoF and it has a rapidly increasing solution time. Due to memory limitations, we cannot solve problems which have more than 1024^2 DoF by using a direct solver on a single core. The solution times of PGMRES with a block Jacobi preconditioner are also rapidly increasing. The solution times of PGMRES with an algebraic multigrid preconditioner are increasing moderately and are faster than the direct solver for large problems. PGMRES with the BoomerAMG preconditioner is the fastest solver for most of the problems (more than 256^2 DoF).

Next we considered the required number of iterations for PGMRES and report the results in Table 4. The required number of iterations for GMRES with a block Jacobi preconditioner requires more than the maximum number of iterations for large problem sizes with more than 512^2 DoF. The ratios of the number of iterations of PGMRES with an algebraic multigrid preconditioner are between 1.23 and 1.37. These ratios are typical values for a multigrid preconditioner (*log N*).

Table 4 The required number of iterations for PGMRES with several preconditioners and the ratios of the number of iterations of consecutive levels of the multigrid method on a single core.

DoF	GMRES	GAMG	GAMG2	GAMG4	ML	Hypre
32^2	838	69	54	42	73	10
64^2	1884	94	73	53	91	12
128^2	5370	120	84	58	109	23
256^2	23613	138	93	65	116	28
512^2	83777	158	106	74	132	31
1024^2	*	182	125	90	150	32
2048^2	*	206	141	111	167	33
Ratio	3.16	1.31	1.26	1.27	1.23	1.37

From now on, we consider the performance on parallel computers. To use parallel direct solvers in PETSc, one has to use third party solvers. However, some of them can be used in BOUT++ without PETSc. Therefore, we considered only parallel PGMRES with different algebraic multigrid preconditioners, i.e., GAMG4, ML, and BoomerAMG. We report the strong scaling property for different problems, with 1024^2 to 8192^2 DoF, for each preconditioner in Fig. 24 and Fig. 25. From these numerical results, we observe a good strong scaling behavior for all algebraic multigrid preconditioners.

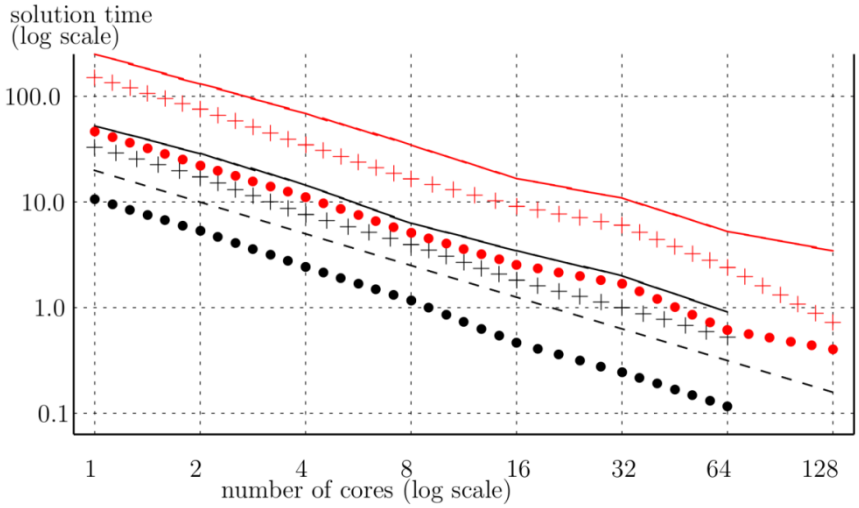

Fig. 24 Average solution time in seconds of parallel PGMRES with different preconditioners as a function of the number of MPI tasks to solve a fixed problem with 1024^2 DoF in black and 2048^2 DoF in red. A solid line stands for the GAMG4 solver , +++ for the ML solver , and ••• for the BoomerAMG solver. As a reference, we plotted the ideal case as - - -.

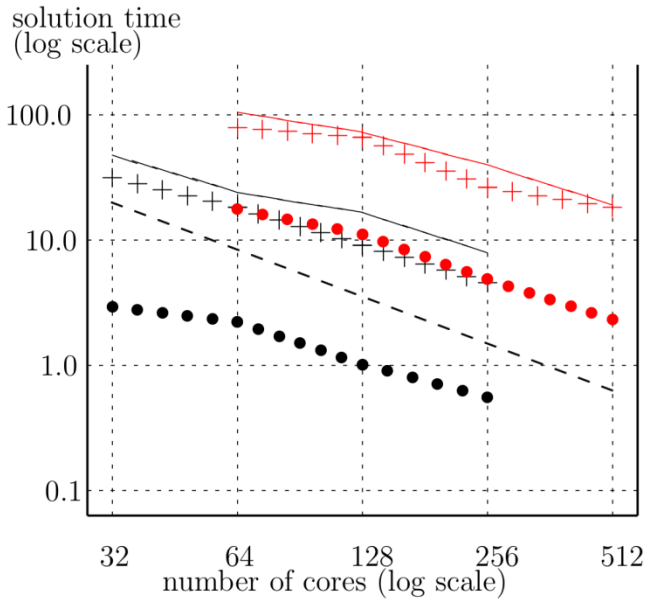

Fig. 25 Average solution time in seconds of parallel PGMRES with different preconditioners as a function of the number of MPI tasks to solve a fixed problem with 4096^2 DoF in black and 8192^2 DoF in red. A solid line stands for the GAMG4 solver, +++ for the ML solver, and ••• for the BoomerAMG solver. As a reference, we plotted the ideal case as - - -.

We also considered the (semi-)weak scaling behavior for the algebraic multigrid preconditioners. We selected four cases, 256^2 DoF per core, $512^2/2$ DoF per core, 512^2 DoF per core, and $1024^2/2$ DoF per core plotted with respect to the number of cores. The results are reported in Fig. 26 and in Fig. 27. They show relatively good scaling properties for each case, especially for the largest case with $1024^2/2$ DoF per core (Fig. 27 in red).

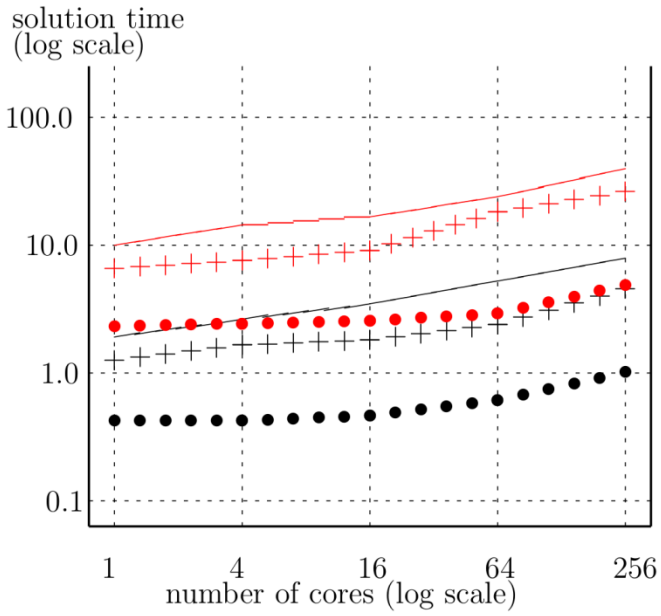

Fig. 26 Average solution time in seconds of parallel PGMRES with different preconditioners as a function of the number of MPI tasks to solve problems with the same number of DoF per MPI task, 256^2 DoF per MPI task in black and 512^2 DoF per MPI task in red. The solid line stands for the GAMG4 solver, +++ for the ML solver, and ••• for the BoomerAMG solver.

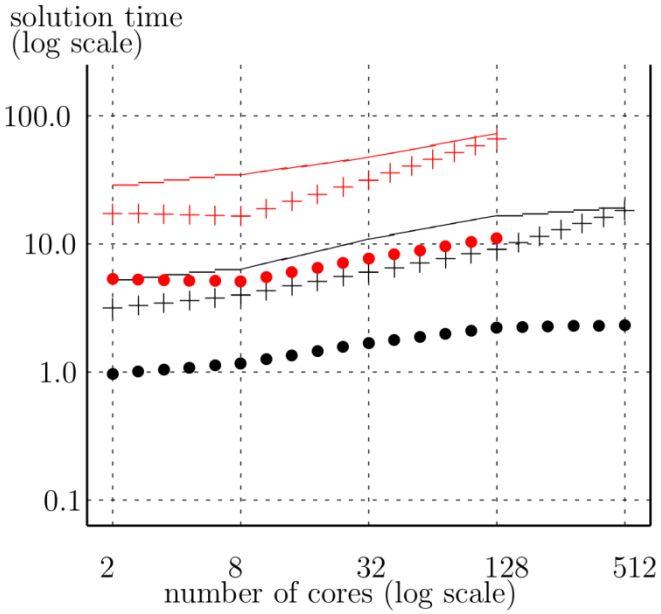

Fig. 27 Average solution time in seconds of parallel PGMRES with different preconditioners as a function of the number of MPI tasks to solve problems with the same number of DoF per MPI task, $512^2/2$ DoF per MPI task in black and $1024^2/2$ DoF per MPI task in red. The solid line stands for the GAMG4 solver, +++ for the MLsolver, and ••• for the BoomerAMG solver.

From the numerical results we conclude that the BoomerAMG solver from the Hypre package is the best solver. Therefore, we investigate the performance behavior of the BoomerAMG solver in more detail and report the results in Fig. 28 and Fig. 29. The numerical experiments show that the BoomerAMG solver has good strong and weak scaling properties. The solution time for large problem sizes running on large numbers of MPI tasks are acceptable for the BOUT++ developers.

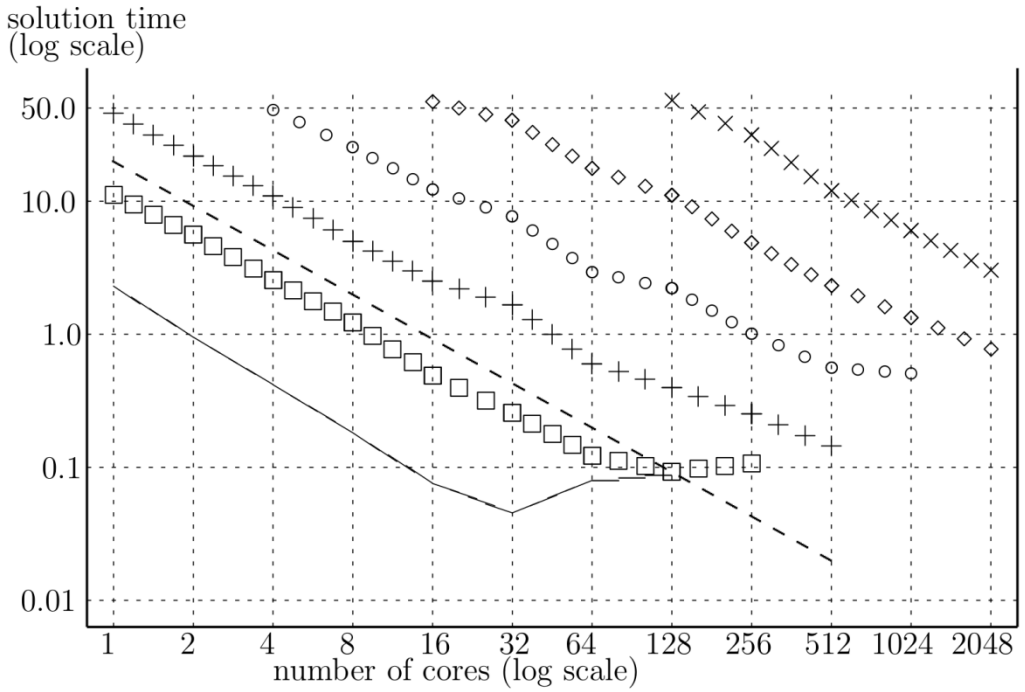

Fig. 28 Average solution time in seconds of parallel PGMRES with the BoomerAMG preconditioner as a function of the number of MPI tasks to solve a fixed problem size, $(2^k)^2$ DoF for *k = 9, 10, 11, 12, 13, 14*. The solid line stands for *k= 9*, □□□ for *k = 10*, +++ for *k = 11*, ○○○ for *k = 12*, ◊◊◊ for *k=13*, and ••• for *k = 14.* The dashed line - - - is for the ideal case.

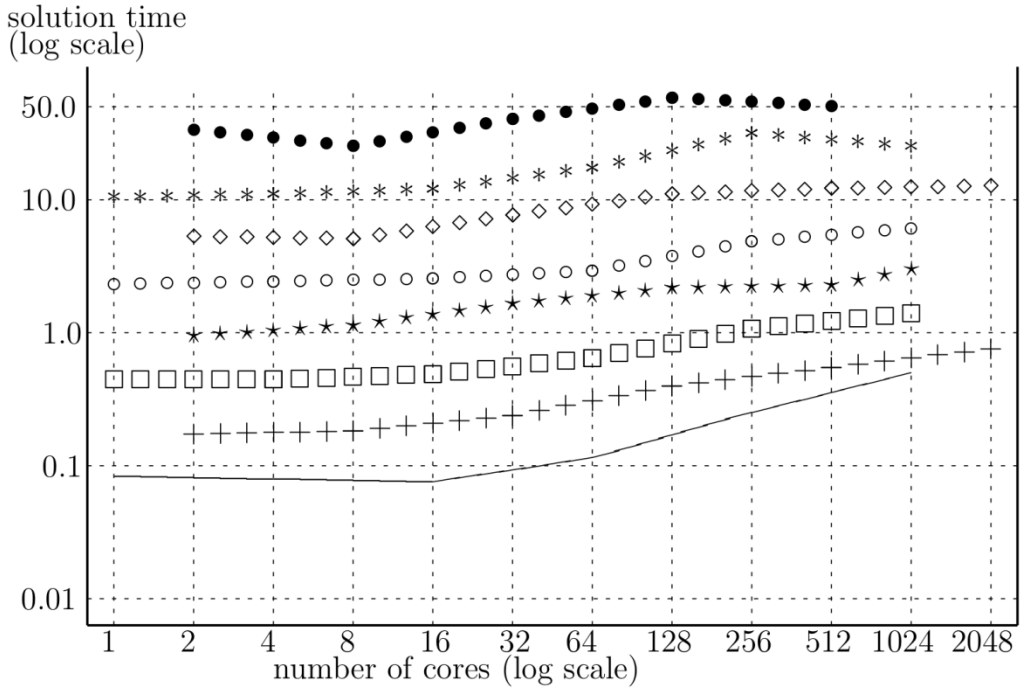

Fig. 29 Average solution time in seconds of parallel PGMRES with the BoomerAMG preconditioner as a function of the number of MPI tasks to solve problems with the same number of DoF per MPI task, 2^k DoF per MPI task for *k = 14, 15, …, 21*. The solid line stands for *k = 14*, +++ for *k = 15*, □□□ for *k = 16*, *** for *k = 17*, ○○○ for *k = 18*, ◊◊◊ for *k=19*, ⋆⋆⋆ for *k = 20*, and ••• for *k = 21*.

3.8. ***Summary***

An algebraic multigrid solver branch (petscamg) concerning the BOUT++ code was set up in github and installed on the Marconi machine. As an initial step K.S. Kang installed the PETSc library on the Marconi machine to use two third party solvers, the

ML solver from the Trilinos package and the BoomerAMG solver from the Hypre package.

J. Omotani from CCFE prepared a realistic test case so we could verify the data conversion from BOUT++ to the PETSc library (and vice versa) with several numerical tests. We achieved numerical performance results for different solvers: a direct solver on a single core and the PGMRES solver with different preconditioners: the block Jacobi, the GAMG for various aggregation smoothing steps, the ML from Trilinos, and the BoomerAMG from Hypre. The PGMRES solver was executed on a single core and on multiple cores. We concluded from the results that the BoomerAMG solver from Hypre is the fastest solver. It was accepted by the BOUT++ developers.

In addition, we prepared a 3D solver with the algebraic multigrid algorithm in the framework of the PETSc library. It is planned that the project coordinator will debug and finalize it.

3.9. *Reference*

[1] Balay, S., S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, and H. Zhang (2017). PETSc Users Manual. TechnicalReport ANL-95/11 - Revision 3.8, Argonne National Laboratory

4. Final Report on HLST project CINCOMP3 – Part 1

The CINCOMP3 project is a continuation of the CINCOMP and CINCOMP2 projects [1, 2, 3] and is dedicated to provide support for European scientists who use the Marconi supercomputer located at CINECA. The Marconi supercomputer was launched in July 2016 and its official production phase started in mid-October 2016. The fusion community has access to two partitions of Marconi named A2 and A3. A2 is based on the latest and final generation [4] of the Intel Xeon Phi product family (*Knights Landing*). The A3 partition is equipped with Intel Xeon 8160 processors (*Skylake*). In the framework of this project, both the hardware and the software of the A2 and A3 partitions were tested using a variety of benchmarks. Multiple issues were found on both partitions and subsequently reported to the Marconi support team via the ticket system.

4.1. *The Marconi supercomputer architecture*

The Marconi supercomputer is located in Bologna at the largest Italian computing centre named CINECA. It currently consists of two partitions named A2 and A3. The A2 partition, which is equipped with processors of the Intel Xeon Phi product family (*Knights Landing*), provides a computational power of about 11 Pflop/s. The A3 partition is equipped with the latest Intel Xeon 8160 processors (*Skylake*) and provides 7.4 Pflop/s. One can find a detailed description of the CPU architecture and the partition structure in our previous reports [1, 2, 3] or on the official Marconi user's guide web page [5]. The European fusion community only has access to the so-called Marconi-Fusion part. This part includes 5 Pflop/s of the A3 and 1 Pflop/s of the A2 partition. The tests in this report will focus on the Marconi-Fusion part of Marconi.

4.2. *Revision of old issues*

We start this project with a revision of open tickets and test them again using the new Intel 2018 compiler and Intel 2018 MPI library.

4.3. *Marconi network performance*

The PingPong test from the Intel MPI Benchmark (IMB) suite [6] was used to test the Marconi network performance. In this test an *N* byte message is sent from process one to process two by means of the functions *MPI_Send* and *MPI_Recv*. When the message is received, process two sends the same data back to process one. The total communication time is calculated as $\Delta t/2$, where Δt is the time consumed by the whole process. The test is repeated 100–1000 times and an average value is computed. Latency is defined as the time spent to send a zero byte message.

Unexpected results were obtained from the intra-node test. The bandwidth drops with increasing message sizes from ~15 GB/s (for a message size of 256 KB) to ~9 GB/s (for a message size of 4 MB). Moreover, the bandwidth between two distinct nodes was higher in comparison to the bandwidth obtained inside one node. These results were not expected and a ticket with the Marconi support team was opened. In response to the ticket the Intel support suggested to use a new option of the IMB Benchmark version 3.1 [7] named *-off_cache.* This flag allows avoiding cache effects and lets the PingPong test use a message buffer, which does very likely not reside in cache. Without this option the communication buffer is the same for all repetitions of a particular message size sample and the bandwidth cannot be measured accurately. The flag has two parameters: the cache size and the cache line size. The default (recommended) value is -1.

Fig. 30 shows the bandwidth measured with the PingPong test using the Intel 2018 compiler with and without the *off_cache* flag inside one SKL node on one CPU. The flag has an important influence on the obtained bandwidth for messages between 10 kiB and 16 MiB. Without the flag the bandwidth continuously decreases after a message size of 256 kiB until ~4.2 GiB/s. With the flag and the recommended default

parameter value of -1 the bandwidth saturates with a value of ~4.2 GB/s and the previously detected bandwidth drop disappears. This means that the bandwidth drop problem can be attributed to caching effects.

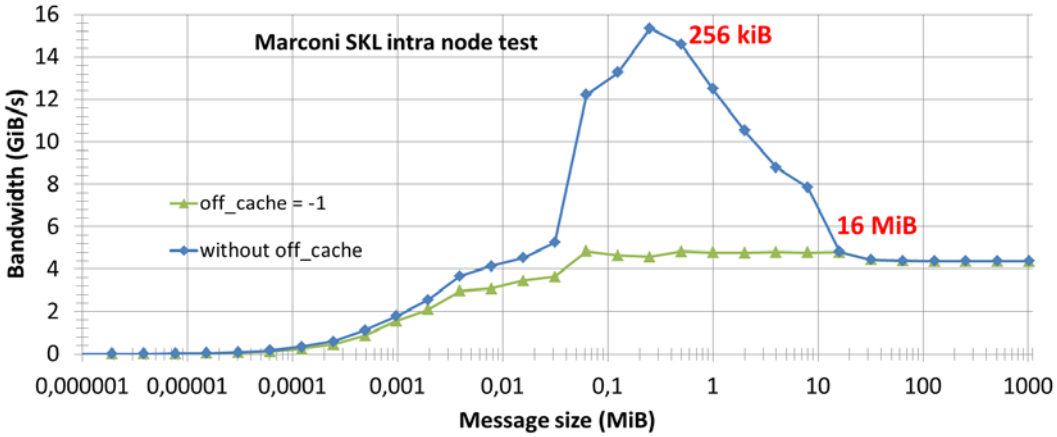

Fig. 30 Comparison of the memory bandwidth for the intra-node PingPong test on the Marconi A3 partition with and without the *off_cache* flag.

We repeated the PingPong test with the *off_cache* flag for the inter-node bandwidth test as well. The results are shown in Fig. 31. The old issue is still remaining even with the new Intel 2018 compiler and with the new *off_cache* flag. The memory bandwidth between two nodes (red line) is higher for large message sizes (>200 kB) compared to the memory bandwidth inside one node (blue line) and even within one CPU (green line). This issue was escalated to the Intel support team. The explanation for this was obtained from the COBRA supercomputer support team [29]. Intel Omni-Path (OPA) uses a protocol from the OpenFabrics Alliance [30] which promotes remote direct memory access (RDMA) switched fabric technologies. RDMA supports zero-copy networking by enabling the network adapter to transfer data directly to or from application memory, eliminating the need to copy data between application memory and the data buffers in the operating system. These transfers require no work to be done by CPUs, caches, or context switches, and transfers continue in parallel with other system operations. When an application performs an RDMA read or write request, the application data is delivered directly to the network, reducing latency and enabling fast message transfer. The RDMA of the OPA is faster than the intra-node communication via the SHMEM fabric if only a single pair of ranks running the PingPong is used. But when using multiple pairs of ranks the SHM fabric is more efficient. Moreover, the SHM fabric uses an additional memory copy operation during the message transfer (“copy in” and “copy out”) – which is not very efficient for large message sizes (> cache size).

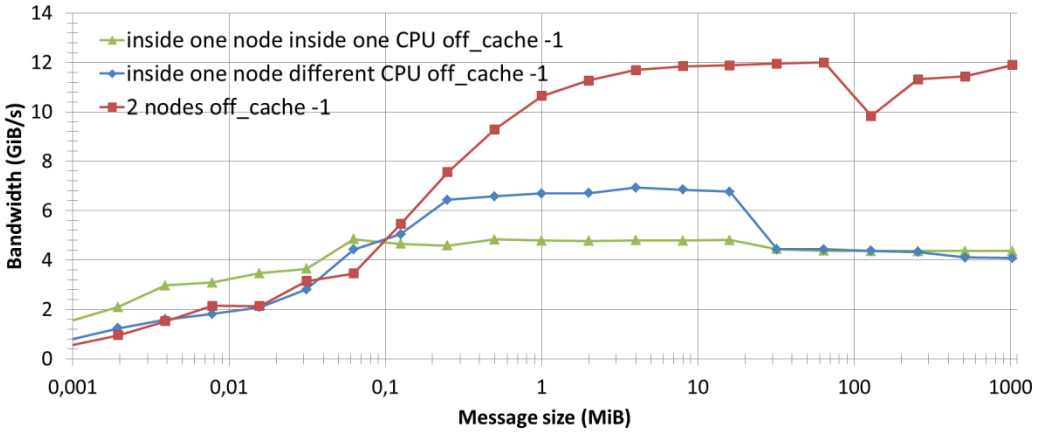

Fig. 31 Comparison of the memory bandwidth for the inter- and intra-node PingPong test on the Marconi A3 partition with the default parameter -1 for the *off_cache* flag.

4.3.1. Issue with Intel MPI 3.0 library

One important limitation of the MPI subroutines *MPI_File_write_at_all* and *MPI_File_read_at_all* was found during the development of the parallel MPI I/O for the HLST project JORSTAR2 [8]. 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 case of the Intel MPI implementation of these subroutines, the variable *count* (the amount of elements to be read) is multiplied by the size in bytes of 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/write the maximum possible number of elements (2147483647 a four bytes integer) for a double precision data type the code crashes with a segmentation fault due to an integer overflow. This means that one MPI task can only read a maximum data chunk of two GiB for a double precision array.

We investigated this issue again using our own test code. The code was compiled with the latest Intel 2018 compiler and the Intel 2018 MPI library. The issue appears to be unresolved. This bug was reported to the Marconi support team again, who forwarded it to the Intel support team. The Intel support was able to reproduce the problem and indicated that it is a bug. They proposed a temporary workaround that solves the problem and announced that the problem will be completely resolved by the Intel MPI version 2019 Update 1.

4.4. *Saturation of the Omni-Path interconnect*

Significant fluctuations in the performance of the interconnect were detected during scalability tests of the VIRIATO code during the HLST project VIRIATO2 [9]. These tests consist of varying the number of MPI tasks being involved in a communication with a corresponding variation of the message size. We assume that the memory bandwidth of the Intel Omni-Path inter node connect (100 Gb/s = 12.5 GB/s of bi-directional bandwidth) [14] on Marconi is not fully saturated by two MPI tasks. Thus, we would need more communication pairs in order to benefit from the full bandwidth.

The previously described IMB PingPong test provides measurements between only two MPI tasks. Therefore, we decided to develop our own test code, which behaves similarly to the PingPong test but has the option of a concurrent message transfer between multiple MPI pairs. We tested this code by comparing it with the IMB PingPong test using only two MPI tasks (one pair). The results were in very good agreement.

For the next step we executed our code with multiple MPI task pairs. During this test messages were sent only in one direction. This makes sure that we test the one directional Omni-Path bandwidth on Marconi. Fig. 32 shows the memory bandwidth obtained on the SKL A3 partition versus the message size. Additionally, the number of MPI pairs was varied. First of all we compare the results of two MPI tasks (diamond blue line) with the output from the IMB (violet line). Both measurements, as discussed above, are in very good agreement giving a maximum memory bandwidth of ~11.5 GB/s.

One can also see that all curves (using different numbers of MPI pairs) are on top of each other. This means that the bandwidth is already saturated with two MPI tasks (one pair) giving a bandwidth of 11.5 GB/s. This result matches the vendor specification of a maximum Omni-Path bandwidth of 12.5 GB/s as our test reaches 92% of the theoretical bandwidth.

For an additional confirmation we used another bandwidth test from the OSU Micro-Benchmarks [21] (Fig. 32 crossed blue line). This test achieved a maximum bandwidth value of 12.05 GB/s. This is in very good agreement with our previous tests and with the IMB benchmark.

However, these results are in disagreement with the observation from the VIRIATO2 [9] project discussed above and the bandwidth measurements for the

Infini-Band interconnect done by other groups [10]. Further investigations of the VIRIATO code have shown that the problem appears in message transfers using shared memory windows. The detailed description of this problem can be found in [27].

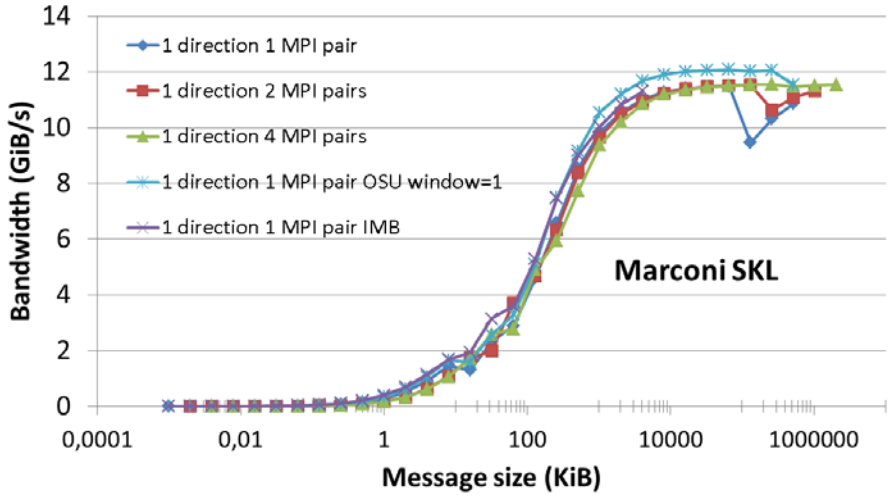

Fig. 32 Bandwidth versus message size obtained from a custom PingPong pair test code using different numbers of MPI tasks on the Marconi A3 SKL partition compared with the IMB and the OSU Micro-Benchmark.

4.5. *Issue with the memory bandwidth for large messages on the Marconi KNL partition*

We repeated the three benchmarks described in the previous section (the IMB, the OSU Micro-Benchmark and our custom MPI pairs bandwidth test) on the Marconi A2 KNL partition (cache mode). The results are presented in Fig. 33. The obtained results are very similar to what we measured on the Marconi SKL partition (Fig. 32). This means that two MPI tasks saturate the bandwidth on the KNL partition as well. However, we detected an important issue. The bandwidth drastically drops from ~10–11 GB/s to about three GB/s for large message sizes (>64 MB) for all tests. All tests were done with the Intel 2018 compiler and the Intel 2018 MPI library. We repeated the test with the GNU compiler and the OpenMPI library. The results were very similar to the observations presented in Fig. 33. This issue was reported to the Marconi support team who escalated it to the Intel support team. The Intel support advised us to use huge pages in order to resolve the problem. The huge pages tests will be performed by the Marconi support team in the near future.

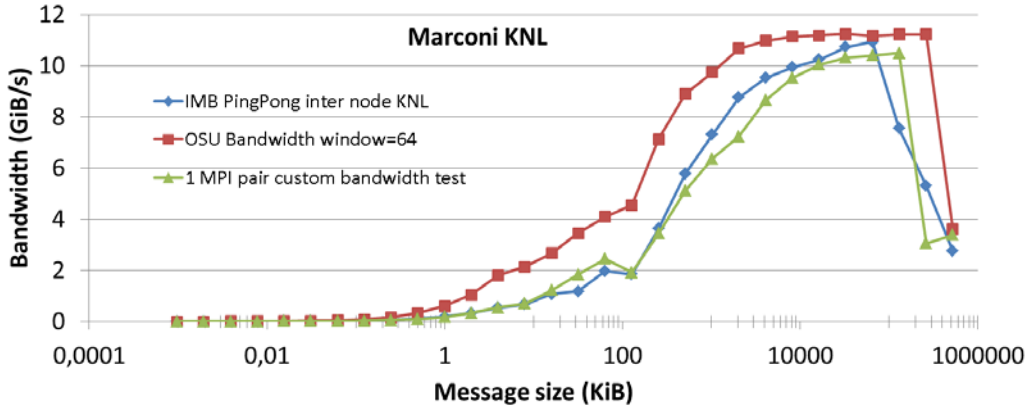

Fig. 33 Bandwidth versus message size obtained from three different benchmarks: (i) blue line – the PingPong test from the IMB; (ii) red line – the uni-directional bandwidth test from the OSU Micro-Benchmark and (iii) green line – our custom MPI pair test code.

4.6. *Check of the bandwidth for symmetric access to the Omni-Path interconnect*

We also used the custom benchmark, described above, to test if there is a difference in the memory bandwidth between two nodes for different placements of the MPI tasks. We ran this test on the Marconi A3 SKL partition using two MPI processes (one pair). The obtained results are presented in Fig. 34. The blue line shows the results for the case where the first task was assigned to CPU one of the first node (pinning number 0) and the second MPI task was pinned to CPU one of the second node (pinning number 0). The green line represents the test where the first task was assigned as before on CPU one of the first node (pinning number 0) but the other MPI task was pinned on CPU two of the second node (pinning number 24). The violet line shows the test results where the first MPI task was assigned on CPU two of the first node (pinning number 24) and the other MPI task was pinned on CPU one of the second node (pinning number 0). All three tests provide very similar results. Therefore, there is no "symmetry" bandwidth problem on Marconi. This problem was previously detected on the first generation of the Intel Xeon Phi processor (Knights Corner) [13] on the HELIOS supercomputer.

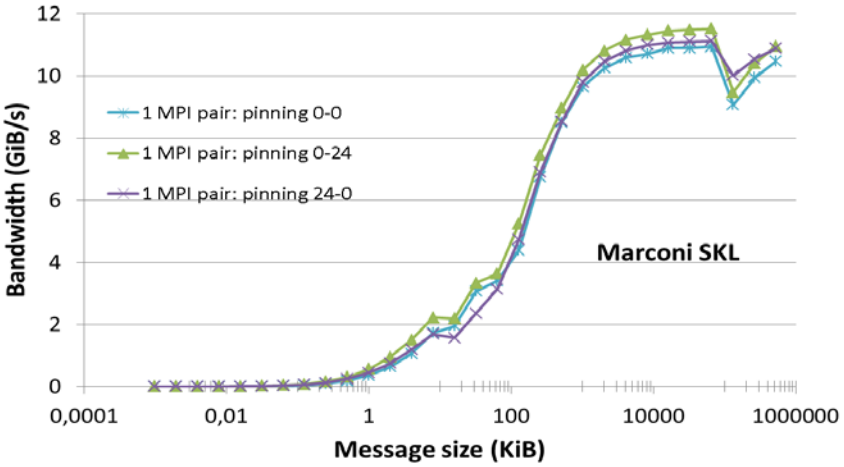

Fig. 34 The bandwidth results (symmetry check) obtained from our custom pair test code using two MPI tasks (one pair) with different pinnings on the Marconi A3 partition.

4.7. *Check of the bi-directional inter-node bandwidth of the Omni–Path interconnect*

We discussed above that Marconi is equipped with the newest Intel Omni–Path inter-node connect, which has a bi-directional memory bandwidth of 25 GB/s (12.5 GB/s in each direction) as specified by the vendor [15, 16]. In the previous sections we measured that the memory bandwidth can reach values of ~11.5 GB/s on both Marconi A2 and A3 partitions which corresponds to >90 % of the theoretical value. However, all these tests were done by sending the message in only one direction. This procedure is illustrated in Fig. 35 a) for 8 MPI tasks (four pairs). In the first step, four MPI tasks (located on the same node) send a message to receivers which are pinned to the second node. In step two all previous receivers simultaneously send a message back to the previous senders. The bi-directional transfer is shown in Fig. 35 b). The main difference in comparison with the one-directional transfer is that during each step a message is sent and received in both directions.

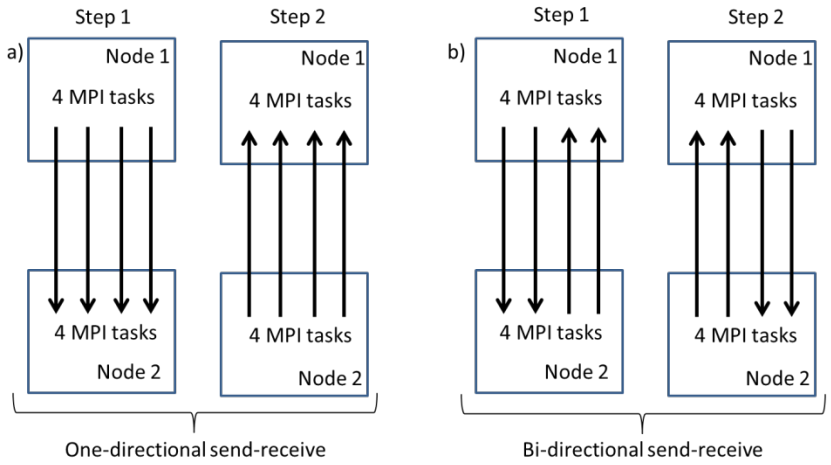

Fig. 35 Schematic of the one- and bi-directional message transfer between two nodes.

We modified our custom MPI pair code in order to reproduce the bi-directional transfer illustrated in Fig. 35 b). The results are presented in Fig. 36: the blue line stands for four MPI tasks (two pairs) and the red line stands for eight MPI tasks (four pairs). Additionally, the *Sendrecv* test from the IMB was executed (Fig. 36, light blue line). As stated by Intel [22] this test should reproduce the full bi-directional memory bandwidth. Finally, the bi-directional bandwidth test of the OSU Micro-Benchmark (Fig. 36, green line) was executed as well. All four tests provide a very similar bandwidth with a maximum value of 22.9–24 GB/s which corresponds to 92–96 % of the theoretical peak value.

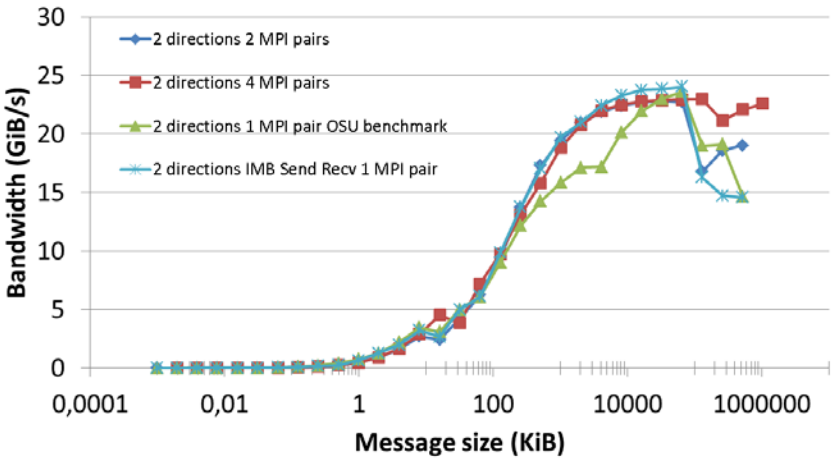

Fig. 36 Bi-directional bandwidth versus message size obtained using three different benchmarks.

4.8. *RAM bandwidth test*

The STREAM benchmark [17] is one of the most popular benchmarks in the high performance computing community for measuring the memory bandwidth between CPU and RAM. Fig. 37 presents the results obtained with the STREAM benchmark. It was launched on a single node (two CPUs) of the Marconi SKL partition. The STREAM code was compiled with the newest Intel 2018 compiler. All four benchmarks (*add*, *scale*, *copy* and *triad*) have a similar behavior with a slightly higher bandwidth yielded for the *add* and *triad* tests. We also tested two different thread pinning methods named *compact* and *scatter* [1, 3]. Both of them provide a typical distribution for a NUMA shared memory system [18].

A maximum bandwidth of ~95 GB/s can be reached on a single socket (one CPU) while ~190 GB/s can be reached on a single node (two CPUs). These results are about 80 % of the theoretical peak memory bandwidth specified by the vendor (238 GB/s) [19], which is a typical fraction for such a system.

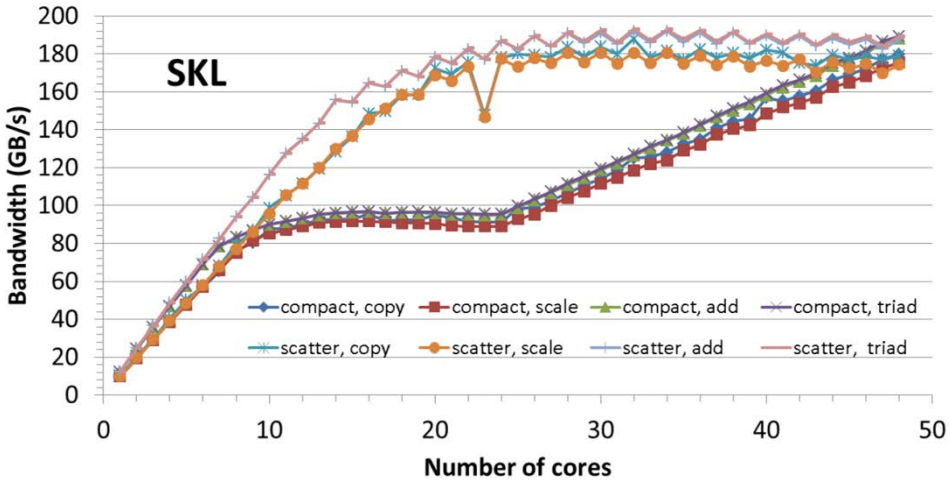

Fig. 37 Results of the STREAM benchmark on a single Marconi computing node from the A3 SKL partition.

The Marconi support team detected that the STREAM benchmark sometimes returns a lower bandwidth than expected. We investigated this problem and found that compiling the code with the following two flags *-xCORE-AVX512* and *-mtune=skylake* causes the bandwidth to drop for the *copy* test (Fig. 38). Using only one of these two flags results in the *copy* test bandwidth to return to its higher value. This is a very important issue as the wiki web page of the Marconi supercomputer suggests to compile code which should run on the SKL partition with both of these flags for better performance [20].

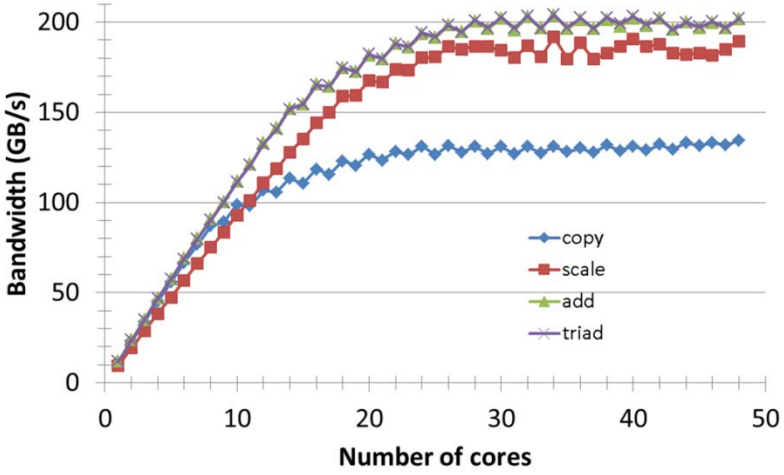

Fig. 38 Results of the STREAM benchmark on a single Marconi computing node from the A3 SKL partition using the Intel 2018 compiler and the *-xCORE-AVX512 -mtune=skylake* flags. Full compilation command line: *icc -xCORE-AVX512 -mtune=skylake -O3 -qopenmp -mcmodel=medium -DSTREAM_ARRAY_SIZE=400000000 -DVERBOSE -DNTIMES=50 stream.c -o stream_skl.x*

We also tried the STREAM benchmark, compiled with the two aforementioned flags, but using the Intel 2017 compiler. The results are shown in Fig. 39. In this case, the bandwidth drop for the *copy* test disappears. We informed the Marconi support team about this issue and they submitted a ticket to the Intel support team. A possible explanation was reported in the Bits & Bytes Computer Bulletin of the MPCDF [28], where it was explained that with the *AVX512* instruction set and with the compiler option *-qopt-zmm-usage=high* that enables the usage of the *zmm* registers, the CPU frequency goes significantly down. However, why it happened with the Intel 2018 compiler and not with the Intel 2017 is still unclear. Moreover, why the usage of both compiler flags (*-xCORE-AVX512* and *-mtune=skylake*) is necessary to cause such a

decrease but each option separately does not produce it, is still not understood and under investigation of the Marconi support team.

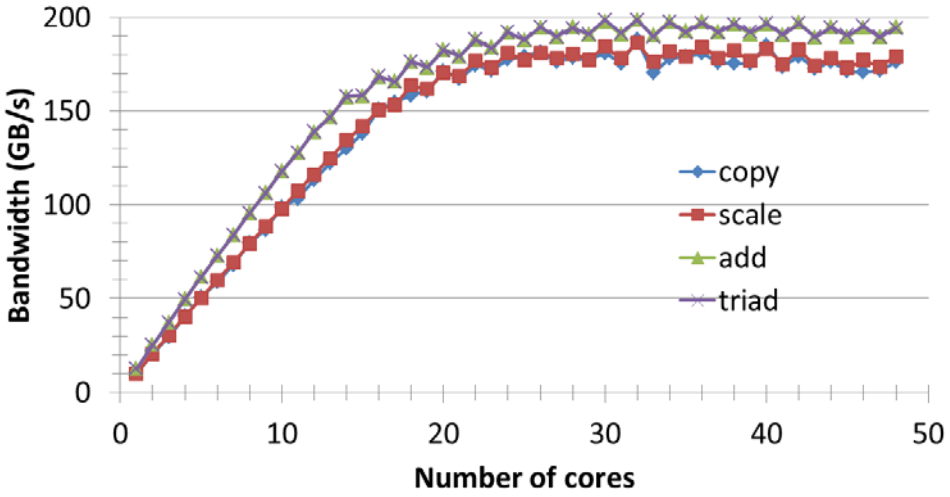

Fig. 39 Results of the STREAM benchmark on a single Marconi computing node from the A3 SKL partition using the Intel 2017 compiler and the *-xCORE-AVX512 -mtune=skylake* flags. Full compilation command line: *icc -xCORE-AVX512 -mtune=skylake -O3 -qopenmp -mcmodel=medium -DSTREAM_ARRAY_SIZE=400000000 -DVERBOSE -DNTIMES=50 stream.c -o stream_skl.x*

Finally, we performed the STREAM benchmark for different array sizes while running on only one thread. Fig. 40 shows how the bandwidth decreases with the growth of the array size. The bandwidth reaches values of ~68 GB/s if the data is located in the first level cache (L1). It decreases to ~45 GB/s for the level two cache (L2) and ~20 GB/s for the level three cache (L3). The bandwidth saturates when reaching the main memory and then stays almost flat with a value of ~11 GB/s. These results were expected as they are the typical behavior of a CPU with three cache levels.

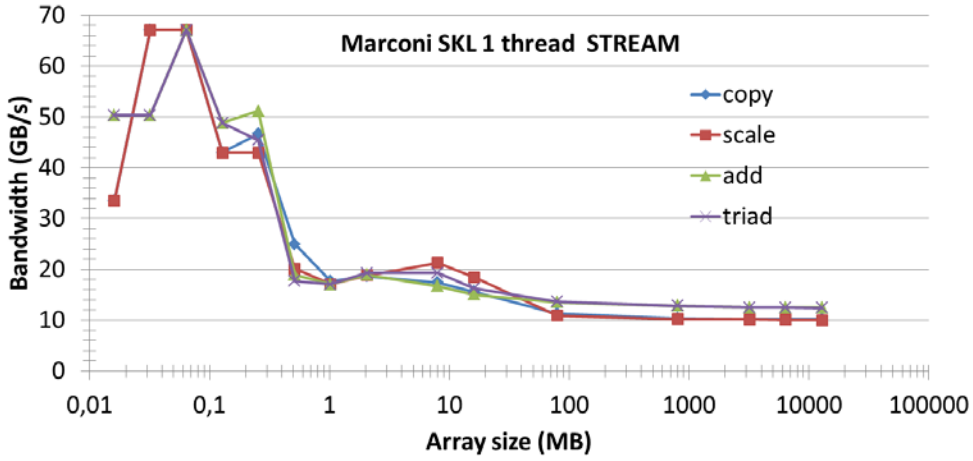

Fig. 40 Results of the STREAM benchmark on a single Marconi computing node from the A3 SKL partition using one OpenMP thread for different array sizes.

4.9. *Marconi performance stability test*

The operation committee of EUROfusion for the Marconi supercomputer decided to perform a regular check of the performance stability. Three codes (GENE, STARWALL and EUTERPE) are run each month and the run times are compared. The STARWALL code is a pure MPI code which uses the external ScaLAPACK library [23, 24]. The GENE code is a hybrid MPI+OpenMP code that uses the LAPACK, FFTW, SLEPc and HDF5 libraries [25]. The EUTERPE code is an MPI code that uses the PETSc [26] library. The codes were launched on both the SKL and KNL partitions. In order to increase the statistical significance each run was

repeated at least three times and the average, minimum and maximum values were calculated from the results.

Fig. 41 shows the average wall clock time, together with its minimum and maximum indicated as an error bar, of the STARWALL code, which was run on the SKL partition using 64 nodes. For an ideally stable supercomputer operation the execution time should be identical for each run i.e. we would get a straight line without any error bars. In our case, the wall clock time significantly fluctuates during tests within one day (one campaign) and also during the whole measurement (all campaigns). For example, on the 5th of April 2018 we launched three instances of the code. All of them provided different execution times: test1=14.19 minutes; test2=22.62 minutes and test3=32.82 minutes. During the whole examination period, three tests finished after about 14 minutes and five after more than 30 minutes. The reason for these differences is still unclear. One of the explanations could be an ailing node as discussed in details in [2]. During this benchmark, a significant amount of jobs also failed due to different other problems on Marconi.

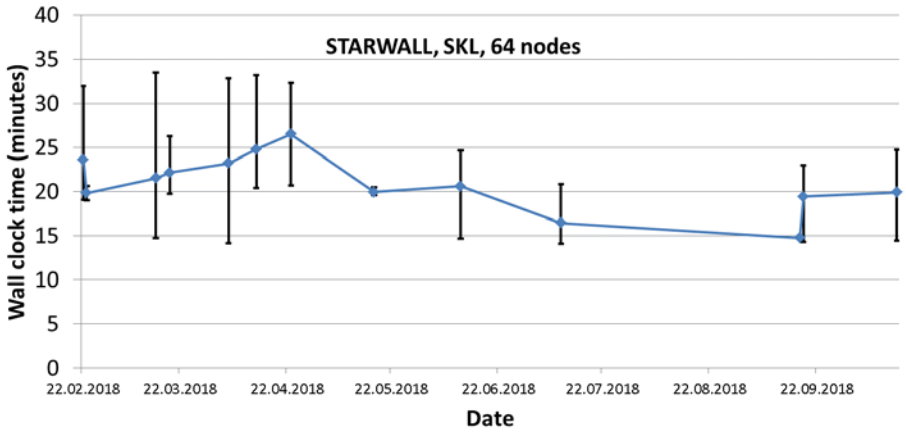

Fig. 41 The execution time in minutes of the STARWALL code which was run on the SKL partition using 64 nodes. Minimum and maximum values are indicated with error bars.

The wall clock time of the GENE code which was launched on the SKL partition using 128 nodes is presented in Fig. 42. One can see that the execution time strongly fluctuates for each single test campaign and also for the whole period in a way similar to the case of the STARWALL code. However, the results are sometimes very similar, for example on the 17th of May 2018 test1=29.52 minutes, test2=30.27 minutes and test3=29.44 minutes.

The fluctuations of the execution time are less pronounced when a code is launched on the KNL partition. Fig. 43 shows the wall clock time of the EUTERPE code, which was executed on the KNL partition using 128 nodes. The average wall clock time fluctuates similarly to the two previous cases of the STARWALL and GENE codes launched on the SKL partition. However, minimums and maximums stay much closer to the average value, i.e. the standard deviation is relatively small.

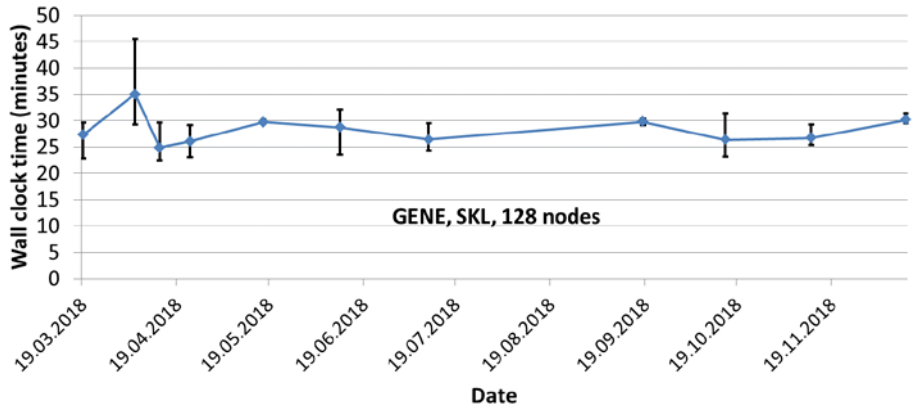

Fig. 42 The execution time in minutes of the GENE code, which was run on the SKL partition using 128 nodes. Minimum and maximum values are indicated with the error bar.

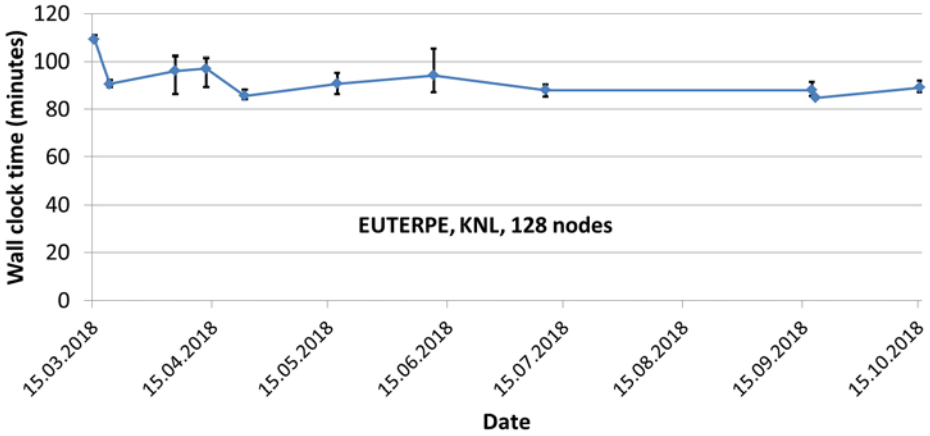

Fig. 43 The execution time in minutes of the EUTERPE code, which was run on the KNL partition using 128 nodes. Minimum and maximum values are indicated with the error bar.

We investigated the detected fluctuations of the wall clock time presented above in more detail. The STARWALL code was also launched on the COBRA supercomputer at the MPCDF [29], equipped with Intel Xeon 6148 Skylake nodes (40 cores with 2.4 GHz) and on the JUWELS supercomputer at the Forschungszentrum Jülich [32] equipped with Intel Xeon Platinum 8168 Skylake nodes (48 cores with 2.7 GHz). We executed 15 identical runs of the STARWALL code on Marconi and COBRA and 10 runs on JUWELS. The results are shown in Fig. 44. The wall clock time fluctuates on both COBRA and JUWELS. However, the fluctuation amplitude on JUWELS is lower in comparison to Marconi and COBRA. One can also see that the code performance on JUWELS is the fastest with a base line of about 13 minutes. On Marconi the wall clock base line is about 16 minutes and on COBRA ~19 minutes. The reasons for this are the different numbers of cores per node and different CPU frequencies on the machines.

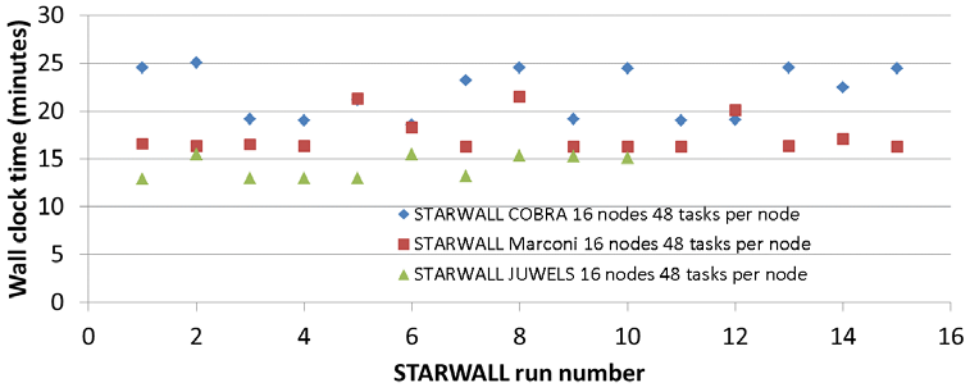

Fig. 44 The execution time in minutes of the STARWALL code which was run on the SKL partition of Marconi (red squares), COBRA (blue diamonds) and JUWELS (green triangles) using 16 nodes.

We also launched 10 STARWALL copies on Marconi and COBRA but now using 30 MPI tasks per node (15 tasks per CPU). The results are shown in Fig. 45. The computational time became very similar on both machines and the fluctuations almost disappeared. The reason for the fluctuations of the wall clock time could be interrupts from Omni-Path as it is being handled by the CPUs within the nodes. If a code is launched using all cores on a node, there are no free resources left for interrupt handling. Then, on one core two tasks (the user code and the interrupt) are executed simultaneously resulting in a decrease of the performance. If we leave some cores on a node empty, the interrupt handler can use these free resources, thus preventing a disruption of the user code execution. Therefore, the fluctuations vanish when we use only 15 MPI tasks per CPU.

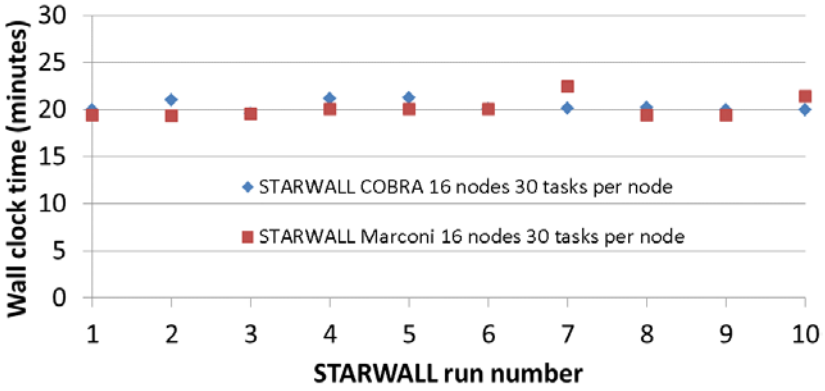

Fig. 45 The execution time in minutes of the STARWALL code which was run on the SKL partition of Marconi (red line) and COBRA (blue line) using 16 nodes and 30 MPI tasks per node, leaving ten and 18 cores free.

We tried to detect where the code spends the extra time during long execution times. With the Intel environment variable I_MPI_STATS one can separate the pure computational time and the time the application spends for MPI communications. For these tests we again used the full node i.e. 40 MPI tasks per node on COBRA and 48 MPI tasks per node on Marconi and JUWELS. Fig. 46 shows a histogram of different timings of the STARWALL code on (a) Marconi, (b) COBRA and (c) JUWELS. The blue column represents the total run time; the red column is the time the code spends in MPI communication and the green column indicates the pure computational time (without communication). Both histograms show that the computational time (green column) always stays constant. Execution time fluctuations only affect the MPI communication time. These results give one more hint that the fluctuation problem is related to the MPI communication rather than to the Intel Omni-Path interconnect.

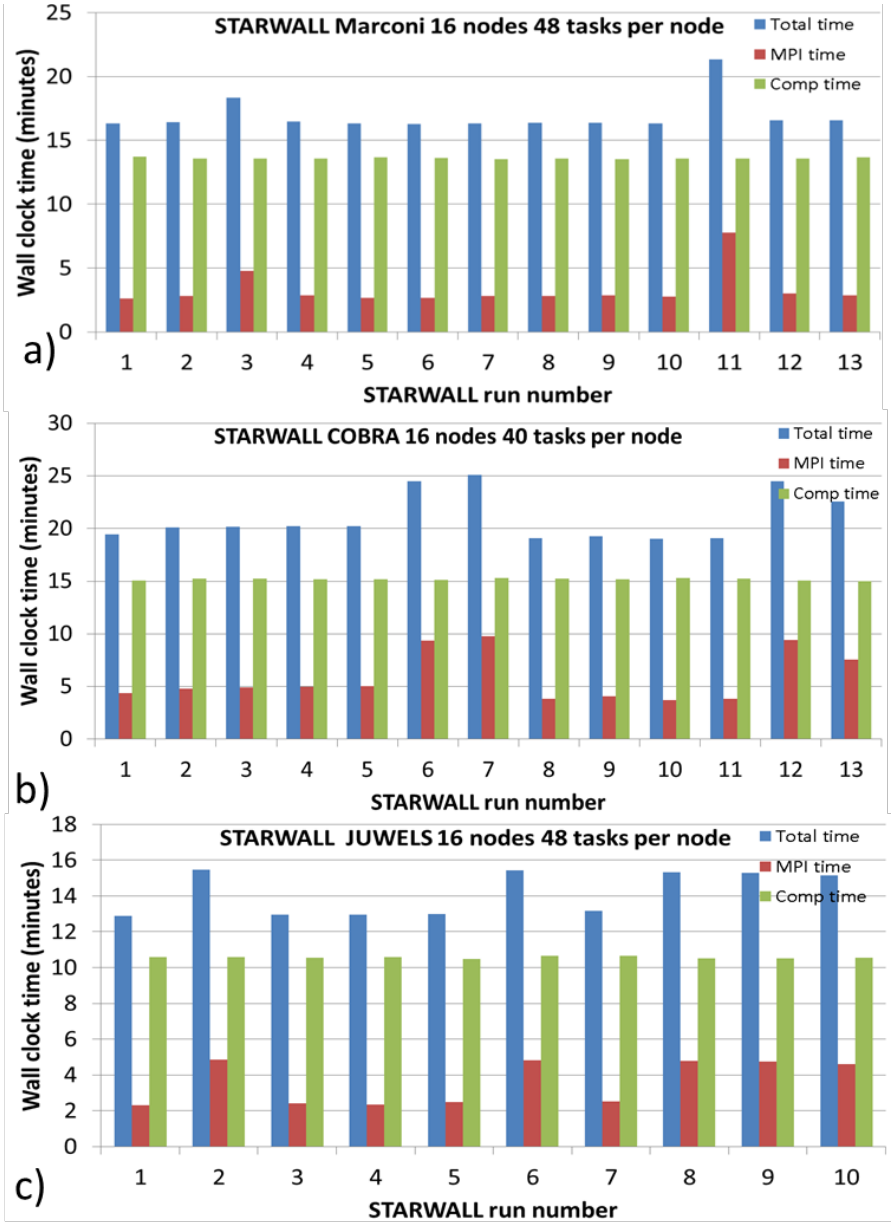

Fig. 46 Histograms of the execution time in minutes of the STARWALL code which was executed on the SKL partition of (a) Marconi, (b) COBRA and (c) JUWELS using 16 nodes. The blue column depicts the total run time; the red column is the time the code spent in the MPI communication and the green column indicates the pure computational time (without communication).

4.10. *Marconi resources usage analysis*

We performed an analysis of the Marconi SKL partition usage during October 2018. Fig. 47 shows a histogram of the jobs that were launched on Marconi characterized by the number of requested nodes (different colors). Most of the submissions (10939 – 76,6 %) were small jobs occupying one to nine nodes. Only five jobs (0.04 %) used 130–256 nodes. There were no jobs using more than 256 nodes.

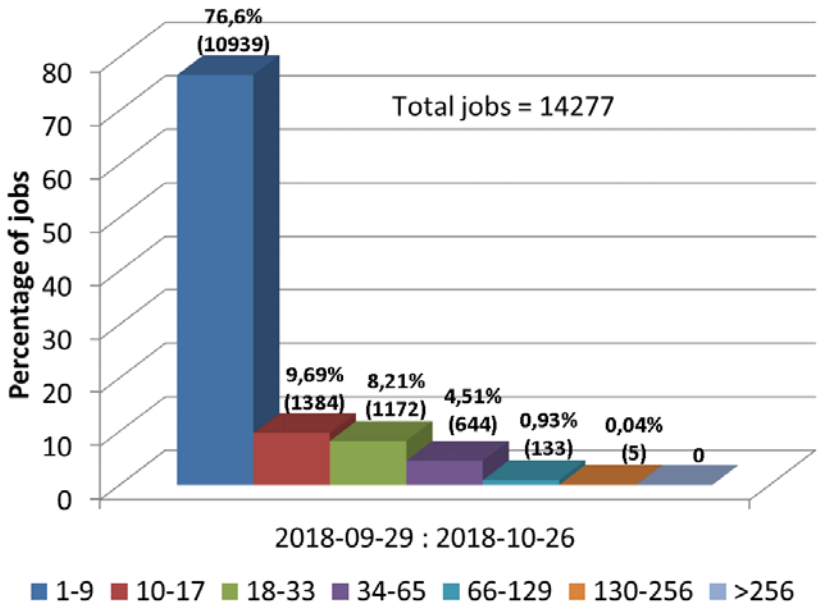

Fig. 47 Histogram of the percentage of jobs occupying a certain number of nodes (different colors), launched on Marconi during October 2018.

If we weight data of the histogram presented in Fig. 47 with the node-hours we will see where Marconi SKL partition spent its resources. Fig. 48 shows a histogram of the consumed node-hours, distinguishing between the requested number of nodes (different colors) that were launched on the Marconi SKL partition during October 2018. Most of the resources (~73 %) were used by jobs with an average degree of parallelism (from 18 to 128 nodes). 25 % of the jobs were of low-parallelism (<33 nodes). Finally, only 0.83 % of the resources were directed to high-parallelism jobs (>128 nodes).

The histogram in Fig. 48 was constructed by using the data of allocated nodes for each job assuming that all cores of each node were involved in the computation. However, sometimes users do not use all cores of a node (48 cores per node in the SKL partition). There can be different reasons for such a behavior. Some algorithms can only run on power of two and others might need more memory per core. In Fig. 48 the white numbers inside each bar indicate the ratio of the allocated resources to the actually used by a code. One can see that only 71.56 % of the available allocated resources were used by jobs in the range of 18 to 33 nodes (green bar). We will continue to investigate the usage of the resources on Marconi for the upcoming months.

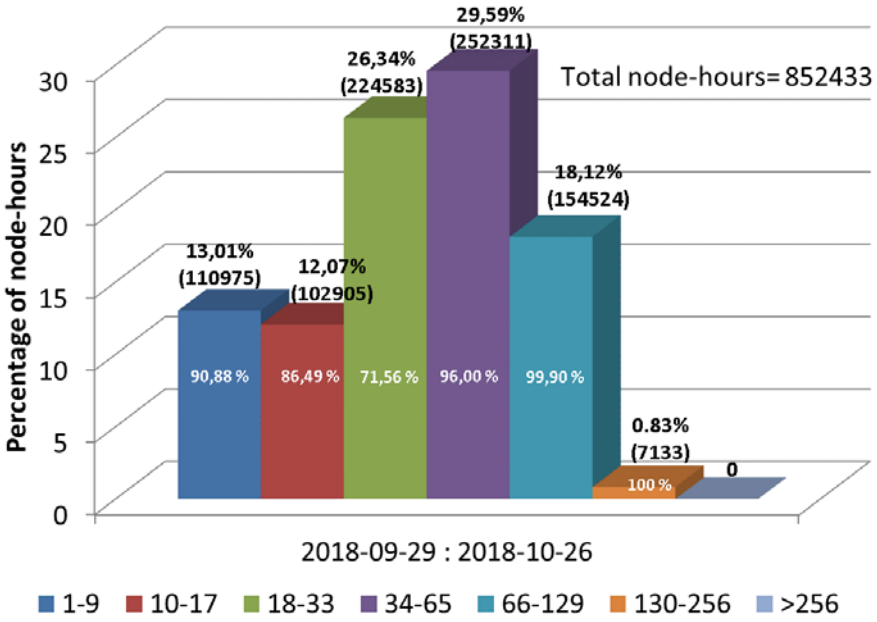

Fig. 48 Histogram of consumed node-hours versus number of nodes (different colors) used on Marconi during October 2018.

4.11. ***Summary***

Different benchmarks and tests were made in order to determine the performance of the A2 (*KNL*) and A3 (*Skylake*) partitions of Marconi. Issues were found that significantly limit their use. Some of them were resolved by the Marconi support team, others however are still under investigation.

Old issues were revised using the new Intel 2018 compiler and Intel 2018 MPI library. The problem with the intra-node network performance and MPI subroutines *MPI_File_write_at_all* and *MPI_File_read_at_all* still persists.

The inter-node benchmarks on the A2 partition revealed a drastic bandwidth drop for message sizes larger than 64 MB.

Three different benchmarks were used to test the Intel Omni-Path interconnect. It was found that the bandwidth could be saturated by using two MPI tasks (reaching ~92 % of the theoretical peak value). The bi-directional bandwidth rate was checked as well. It works properly providing ~92–95 % of the theoretical value.

The STREAM benchmark shows an intra-node memory bandwidth drop for the *copy* test using the Intel 2018 compiler and the *-xCORE-AVX512* and *-mtune=skylake* compilation flags. This is a significant issue as the Marconi wiki pages give the advice to always use both of these flags to achieve the best performance.

The so-called “three code benchmark” was executed regularly in order to check the stability of Marconi in terms of the execution time for real production codes. It was found that the wall clock time of identical codes can fluctuate significantly from one run to the next. It was also detected that these fluctuations appear only in the MPI communication, the computation time always stays constant.

Fluctuations in the execution time of more than 20 % for a run of a specific code were detected. It was found that the fluctuation problem is related to the MPI communication rather then to the Intel Omni-Path interconnect.

4.12. ***References***

[1] S. Mochalskyy, HLST annual report 2016
[2] S. Mochalskyy, HLST annual report 2017
[3] N. Moschuering, HLST annual report 2017
[4] https://itpeernetwork.intel.com/unleashing-high-performance-computing/

[5] https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide
[6] https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[7] https://software.intel.com/en-us/imb-user-guide
[8] S. Mochalskyy et al, "Parallelization of JOREK-STARWALL for non-linear MHD simulations including resistive walls (Report of the EUROfusion High Level Support Team Projects JORSTAR/JORSTAR2)", (2018).
[9] T. Ribeiro, HLST annual report 2017
[10] W. Gropp et al, "Modeling MPI Communication Performance on SMP Nodes: Is it Time to Retire the Ping Pong Test", EuroMPI 2016 Proceedings of the 23rd European MPI Users' Group Meeting, Pages 41-50.
[12] http://www.mpcdf.mpg.de/services/computing/hydra
[13] S. Mochalskyy, HLST annual report 2015
[14] https://www.intel.la/content/www/xl/es/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html
[15] https://www.intel.com/content/www/us/en/products/network-io/high-performance-fabrics/omni-path-host-fabric-interface-adapters.html
[16] https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/transforming-economics-hpc-fabrics-opa-brief.pdf
[17] https://www.cs.virginia.edu/stream/
[18] https://en.wikipedia.org/wiki/Non-uniform_memory_access
[19] https://ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2_10-GHz
[20] https://wiki.u-gov.it/confluence/display/SCAIUS/MARCONI-SKL+%28Marconi-A3%29+environment
[21] http://mvapich.cse.ohio-state.edu/benchmarks/
[22] https://software.intel.com/en-us/imb-user-guide
[23] Merkel P. and Sempf M. 2006 Proc. 21st IAEA Fusion Energy Conf. (Chengdu, China) TH/P3-8
[24] Merkel P., Strumberger E., Linear MHD stability studies with the STARWALL code arXiv:150804911 (2015)
[25] http://genecode.org/
[26] http://fusionwiki.ciemat.es/wiki/EUTERPE
[27] T. Ribeiro, HLST annual report 2018
[28] http://www.mpcdf.mpg.de/about-mpcdf/publications/bits-n-bytes?BB-View=198&BB-Document=198
[29] https://www.mpcdf.mpg.de/services/computing/COBRA
[30] https://en.wikipedia.org/wiki/OpenFabrics_Alliance
[31] https://en.wikipedia.org/wiki/Remote_direct_memory_access
[32]http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html;jsessionid=03F99C7DEBB56E2875A0F6B294E50EC4

5. Final report on HLST project SPICE

5.1. *The SPICE2 and SPICE3 codes*

The SPICE (Sheath Particle In CEll) package includes two codes: SPICE2 (2D3V) and SPICE3 (3D3V) [1, 2]. These codes are dedicated to perform simulations of magnetized plasmas in contact with solid objects and have been successfully used for the study of plasma deposition near castellated plasma-facing components (PFCs) [3, 4]. In order to resolve the virtual cathode (formed at the surface due to the thermionic emission) the size of the PIC cells has to be relatively small (one tenth of the Debye length). Preliminary 3D simulations with a cell size of one quarter of the Debye length have shown important modifications of the escaping thermionic current arising from a localized hotspot. In order to perform complete 3D simulations without any simplifications, substantial improvements in the scaling and parallelization of the code have to be done. In particular, the Poisson solver which is still completely sequential has to be parallelized.

5.2. *Status of the code*

SPICE3 is written in Fortran 90 and provides its output in the Matlab MAT binary format. The code is parallelized using domain decomposition. All internal routines are parallel except for the Poisson solver. During the simulation, each process handles the movement of the particles. Particles, which are crossing the sub–domain boundaries, are being transferred via MPI calls. Each process generates a fraction of the charge density vector. These fractions are send via MPI to the master process, which creates the complete (global) charge density vector and executes the Poisson solver. Afterwards, the master task splits the resulting potential vector into chunks for each subdomain and scatters them back to each core.

The code scales up to only 64 cores. It is notorious for its memory requirements, which is typically 4–8 GB per core. The Poisson equation is solved via a sequential multigrid solver, which employs the UMFPACK library. A new parallel Poisson solver should decrease the memory consumption by avoiding the storage of the global matrix on each core. The parallelization of the solver should also speed up the code and therefore allow to simulate a more detailed structure of the plasma close to the boundary emission region.

5.3. *PETSc library*

The Portable Extensible Toolkit for Scientific Computation (PETSc) is a library of data structures and routines developed by the Argonne National Laboratory for computing the scalable (parallel) solution of scientific applications mostly for partial differential equations and sparse matrix computations [5]. The library includes a variety of solvers with different preconditioners for the Poisson equation ($\nabla^2\varphi = -\frac{\rho}{\varepsilon_0}$, where φ is the electrostatic potential, ρ is the charge density and ε_0 is the vacuum permittivity). Among them, there are parallel and sequential, direct and iterative, linear and nonlinear solvers. We will concentrate here on parallel iterative linear solvers and their preconditioners, as they are the most appropriate for the current project. The best solver for such a problem is KSPCG — the Preconditioned Conjugate Gradient (PCG) iterative method.

The advantage of the PETSc library is that the users need to only prepare the input data (matrices and vectors) and then chose a solver. The library provides all the necessary code to calculate the solution, which means that all solvers can be easily tested for the current problem in order to find the fastest one. Switching the solver can simply be done by using one PETSc command: *KSPSetType(KSP ksp, KSPType method),* where *KSPType method* is one of the solvers.

5.4. *Implementation of the PETSc Poisson solver*

In order to use a PETSc Poisson solver, an input equation matrix and a right hand side (RHS) vector must be prepared. The equations matrix should also include all boundary conditions. The SPICE3 code simulates a volume with six boundary planes. Four of them are periodic and two have a Dirichlet boundary condition. In a common simulation scenario, the upper boundary has a fixed potential and serves as a particles injector, while the bottom boundary has a fixed potential as well but serves as a wall.

Inside the simulation domain of the SPICE3 code any arbitrary combination of basic shapes, such as blocks, spheres, cylinders, cones etc., define the internal geometry. These objects should also be treated as walls with a Dirichlet boundary condition and therefore lead to a modification of the equation matrix.

5.4.1. **Objects free PETSc Poisson solver**

The equations matrix which includes all internal objects and boundary conditions should be provided by the project coordinator and is still in preparation. Therefore, in a first step, we have decided to develop our own matrix using the aforementioned boundary conditions (four periodic and two Dirichlet) but ignoring any internal objects. We have used the spare matrix distributed format (AIJ) for our parallel calculations.

Fig. 49 shows the middle plane of the potential distribution obtained from the newly developed Poisson solver, which uses the PETSc library. For this test, we used the typical grid size of the SPICE3 code (129×129×129 PIC nodes). A periodic boundary condition was applied in the *x*- and *y*-directions, while a Dirichlet boundary condition was used in the *z*-direction with *z*(0) = 500 V and *z*(129) = 0 V. We can see in Fig. 49 that the obtained potential distribution decreases linearly from the top boundary *z*(0) to the bottom one *z*(129) as expected. Due to the periodic boundary condition, the potential at each constant *z* is identical for all *y*. This simple test confirms the correctness of our solver in a vacuum (without any charge density i.e. with a zero RHS vector).

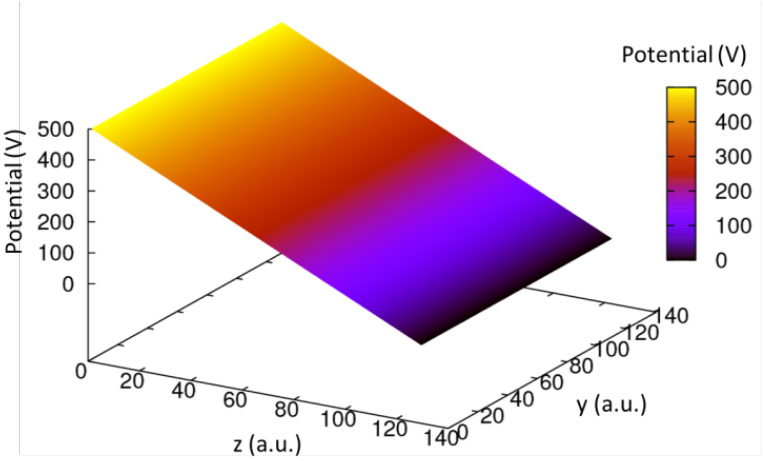

Fig. 49 Potential distribution of the middle (x = 64) *z*-*y* plane obtained from the parallel PETSc Poisson solver using a 129×129×129 PIC grid, periodic boundary conditions in the *x*- and *y*-directions and a Dirichlet boundary condition in the *z*-direction respectively (*z*(0) = 500 V, *z*(129) = 0 V).

The previous test used a zero RHS vector i.e. there was no charge and it simply resolved the equation $\nabla^2\varphi = 0$. In the test presented in Fig. 50, we set two charges: one positive and one negative at equal distance from each other and from the boundaries. We used homogeneous Dirichlet boundary conditions in this test. The potential isolines are distributed over the domain. The 0 V line (black) is located in

the middle of the domain exactly between the two charges. This test confirms the correctness of our Poisson solver for a non vanishing right hand side.

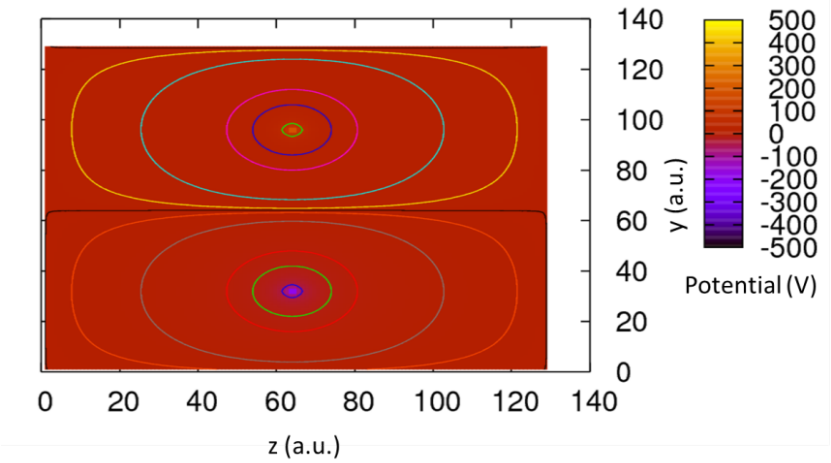

Fig. 50 Potential distribution of the middle (x=64) z-y plane obtained from the parallel PETSc Poisson solver using a 129×129×129 PIC grid, periodic boundary conditions in the x- and y-directions and a Dirichlet boundary condition in the z-direction ($z(0) = z(129) = 0$ V).

With the test presented in Fig. 51, we want to validate the periodic boundary conditions of our simulation domain. For this test, we include only one positive charge in the vicinity of one of the periodic boundaries. In Fig. 51, we can see that the potential isolines have a closed structure and that the periodicity is preserved.

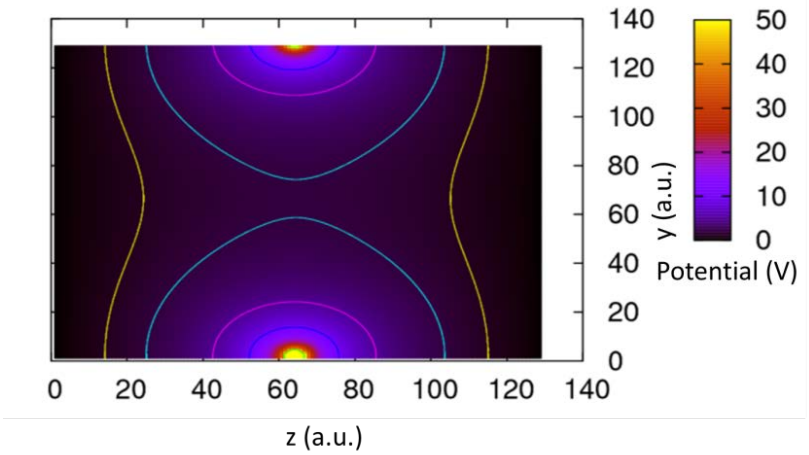

Fig. 51 Potential distribution of the middle (x = 64) z-y plane obtained from the parallel PETSc Poisson solver using a 129×129×129 PIC grid, periodic boundary conditions in the x- and y-directions and a Dirichlet boundary condition in the z-direction ($z(0) = z(129) = 0$ V).

Finally, we combine all our tests together. As in all previous test cases the periodic boundary condition was applied in the x- and y-directions and the Dirichlet boundary condition was again set to $z(0) = 500$ V and $z(129) = 0$ V. One positive charge was put close to the plane $z(129)$ with a 0 V potential. Fig. 52 shows the potential obtained from this test. We can see that the potential decreases linearly similar to test number one. The potential peak from the positive charge is close to the 0 V plane. In summary, no tests of this section are contradicting the correctness of our PETSc Poisson solver.

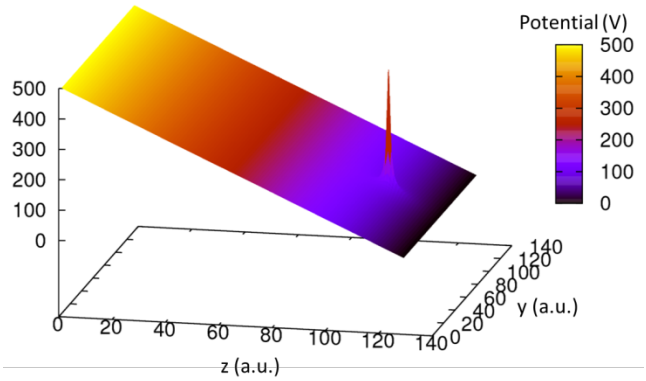

Fig. 52 Potential distribution of the middle (*x*=64) *z*-*y* plane obtained from the parallel PETSc Poisson solver using a 129×129×129 PIC grid, periodic boundary conditions in the *x*- and *y*-directions and a Dirichlet boundary condition in the *z*-direction (*z*(0) = 500 V, *z*(129) = 0 V).

In the last test, we compared the results from the PETSc solver with an exact analytical solution. For this test, we chose the following function $f(x) = x \cdot \cos(x)$ as exact solution. Then the exact right hand side has the following form $f''(x) = -2 \cdot \sin(x) - x \cdot \cos(x)$. We implemented this RHS vector in our solver and calculated the solution. The results from the solver (red line) together with the exact solution (green line) are shown in Fig. 53. As one can see both results are on top of each other. Therefore, this test confirms again the correctness of our Poisson solver.

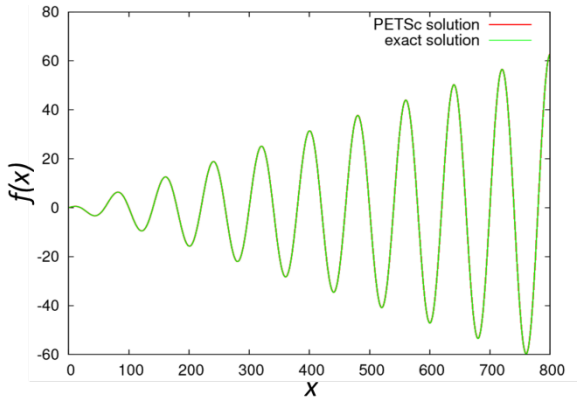

Fig. 53 The exact (green line) and the PETSc (red line) solution of the function $\mathbf{f(x) = x \cdot \cos(x)}$.

5.4.2. **Scaling of the PETSc Poisson solver**

After the Poisson solver has been validated we now measure its scaling properties. For this test, a 128×128×128 PIC grid was used. Fig. 54 shows the wallclock time versus the number of MPI tasks. One can see that our solver scales almost linearly. The total computational time using 512 MPI tasks is about 0.3 seconds. In the final version of the solver, the computational time should decrease even more. For the current solver we used the default preconditioner. In addition, the approximate solution vector was set to 0 V. For the real application, the solution vector from the previous iteration of the time loop will be used to improve the convergence of the solver.

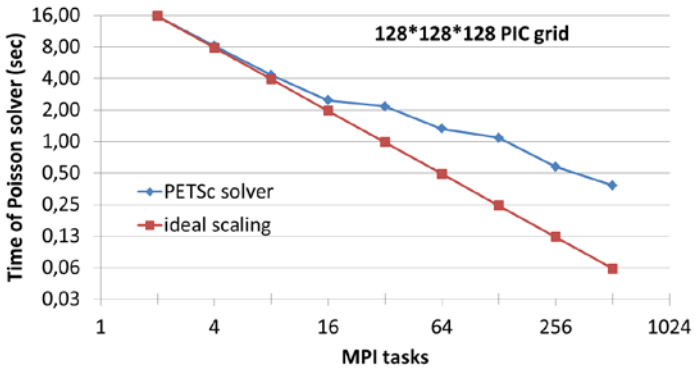

Fig. 54 Scaling of the total wallclock time of the PETSc Poisson solver versus the number of MPI tasks.

5.4.3. Test on a large PIC grid

Up to now, all tests were done for a typical SPICE3 grid that is in the range of 128×128×128 PIC nodes. However, this project is dedicated to develop a Poisson solver which scales to large numbers of cores and which can be used on a large grid. Therefore, we repeated the test presented in Fig. 52 but with a much larger grid: 550×550×550 PIC nodes. The potential obtained from this test is shown in Fig. 55. The distribution is identical to the one shown in Fig. 52. The potential decreases linearly from the $z(0)$ plane with 500 V to the plane with 0 V $z(550)$. The positive charge close to the $z(550)$ plane can also be seen in Fig. 55. This shows that our solver works correctly on a large-scale grid.

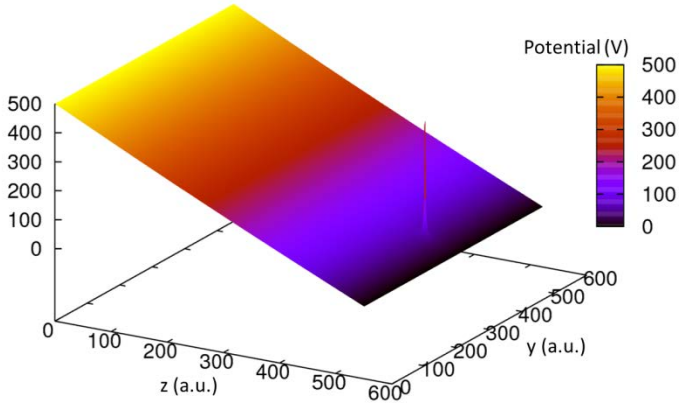

Fig. 55 Potential distribution of the middle (x=64) z-y plane obtained from the parallel PETSc Poisson solver using a 550×550×550 PIC grid, periodic boundary conditions in the x- and y-directions and a Dirichlet boundary condition in the z-direction ($z(0)$ = 500 V, $z(129)$ = 0 V).

5.5. *Implementation of the real right hand side vector*

After all synthetic tests were successfully performed, we implemented in our solver the right hand side (rho) vector from the real SPICE3 “internal object free” testcase. Fig. 56 shows the obtained potential distribution from both the SPICE3 multigrid sequential solver (blue line) and from the PETSc GMRES parallel solver (red line) in two different planes: a) in the middle of the simulation domain at x=64 and y=64 and b) in the plane of the periodic boundary at x=1 and y=64. In this test the periodic boundary conditions were used in the x- and y-directions and a Dirichlet boundary condition in the z-direction ($z(0)$ = -2.8 V, $z(129)$ = 0 V). The obtained results from both solvers are on top of each other for both figures. Therefore, such results confirm again the correctness of our solver. It provides identical results in comparison to the original SPICE3 multigrid solver for a moderate problem size (128×128×129 PIC grid) and it works for a large grid as it was shown in Fig. 55.

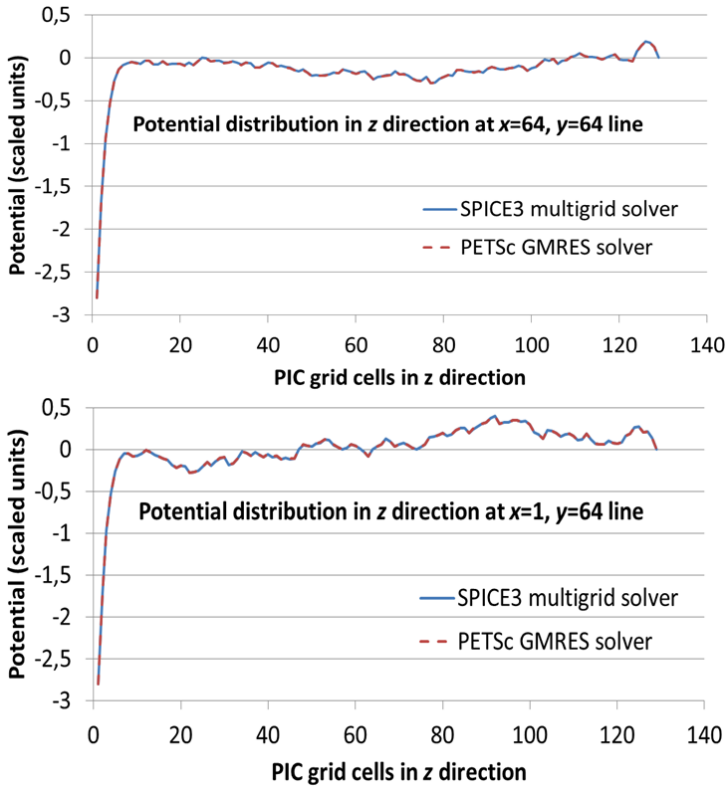

Fig. 56 Potential distribution in the *z*-direction at *x*=64, *y*=64 (a) and *x*=1, *y*=64 (b) planes obtained from the parallel PETSc GMRES Poisson solver (red line) and SPICE3 Poisson solver (blue line) using the real right hand side data. Periodic boundary conditions were used in the *x*- and *y*-directions and a Dirichlet boundary condition in the *z*-direction (*z*(0) = -2.8 V, *z*(129) = 0 V). The simulation domain was distributed among 16 MPI tasks in *x*- and *y*-direction.

In Fig. 56, we shown comparison of the PETSc solver with the SPICE3 multigrid solver. In this run the simulation domain of the SPICE3 code was distributed among 16 MPI tasks (four in the *x*-, four in the *y*- and one in the *z*-direction, respectively). As a next step we wanted to verify if our solver works properly when the distribution is made also in the vertical *z*-direction. Fig. 57 shows the obtained potential distribution as in the previous test from both the SPICE3 multigrid sequential solver (blue line) and from the PETSc GMRES parallel solver (red line) for a test case of 64 MPI tasks (four in the *x*-, four in the *y*- and four in the *z*-direction, respectively). A comparison at four different cuts was done. The first one was chosen at *x*=33 – a plane of the domain decomposition in the *x*-direction. The second comparison was done for a plane at *y*=33 – a plane of the domain decomposition in the *y*-direction. The third plane was chosen in the vertical *z*-direction (*x*=50 and *y*=33). The last comparison was done at the periodic boundary plane (*x*=129). The obtained results from both solvers as in the previous test are on top of each other for all four figures. Therefore, such results again confirm the correctness of our solver. It provides identical results in comparison to the original SPICE3 multigrid solver for any domain decomposition in all directions.

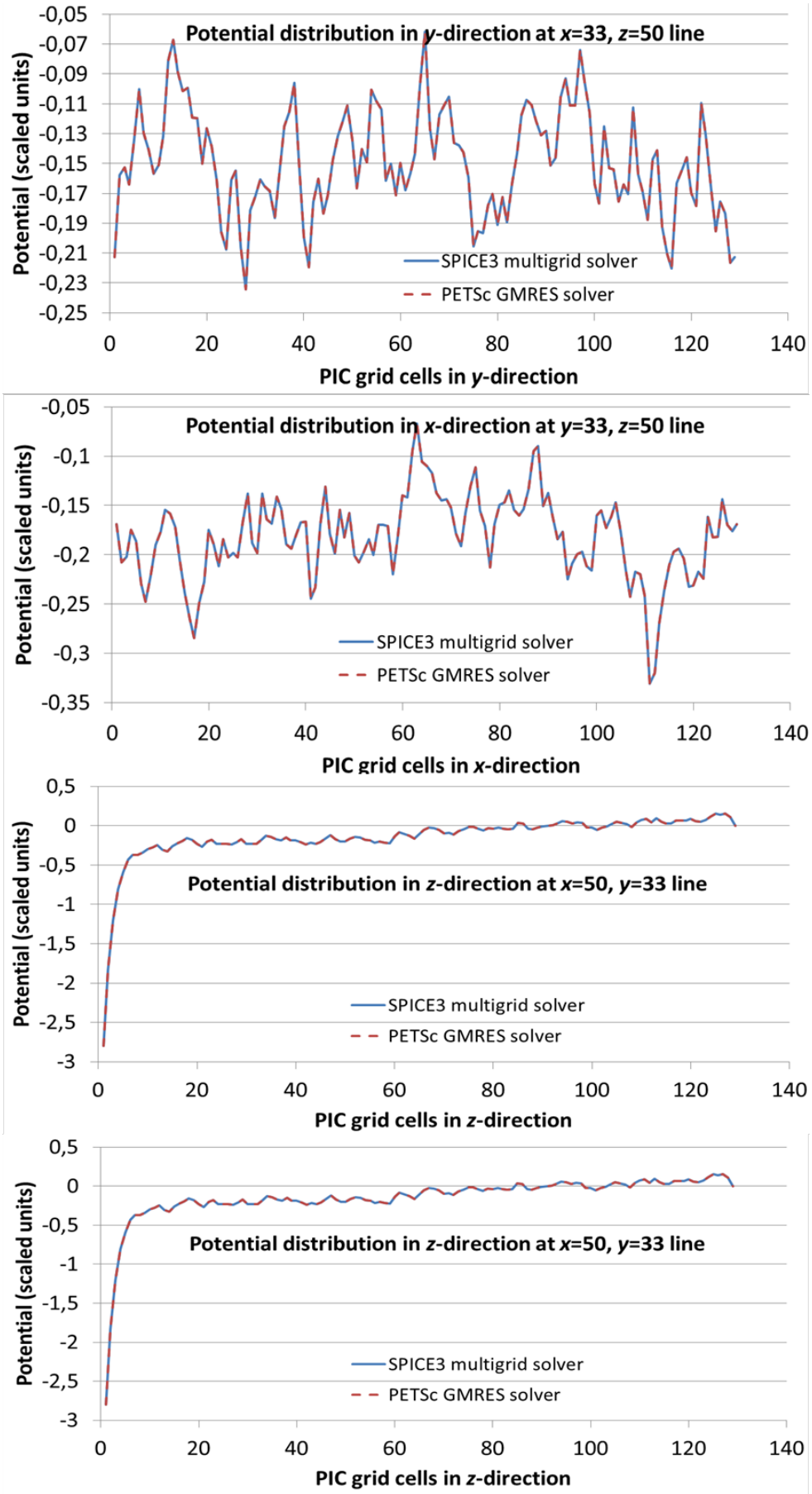

Fig. 57 Potential distribution in different directions obtained from the parallel PETSc GMRES Poisson solver (red line) and SPICE3 Poisson solver (blue line) using the real right hand side data. Periodic boundary conditions were used in the *x*- and *y*-directions and a Dirichlet boundary condition in the *z*-direction ($z(0) = -2.8$ V, $z(129) = 0$ V). The simulation domain was distributed among 64 MPI tasks in all three directions.

5.6. *Integration of the library into the SPICE code*

After the solver was successfully tested in the stand-alone regime we integrated it into the SPICE3 code. Two identical runs were executed: one with the serial multigrid solver, and another one using the parallel PETSc solver. As in all previous tests, we used a simulation domain without any internal objects inside. Periodic boundary conditions were used in the *x*- and *y*-directions and a Dirichlet boundary condition in the *z*-direction (z(0) = -2.8 V, z(129) = 0 V). Very good agreement between the two solvers was found. Fig. 58 shows that the absolute potential difference between these solvers is in the range of 10^{-6} kT_e. Therefore, we conclude that the new PETSc parallel solver was successfully implemented into the SPICE package.

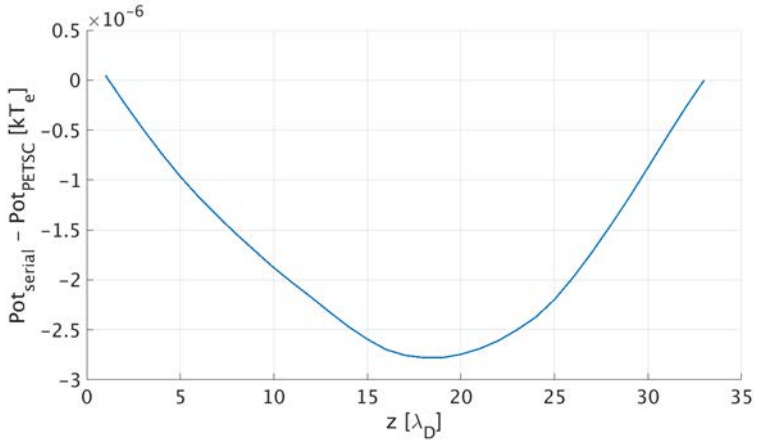

Fig. 58 Absolute potential difference in the *z*-plane between the serial multigrid solver and the parallel PETSc solver obtained from the output of the SPICE code.

5.7. *Vacuum Poisson solver with internal objects*

Our previous results were obtained using the PETSc Poisson solver that does not include any internal object inside the simulation domain. Now we modify the equation matrix using the data concerning the internal object boundaries from the SPICE3 code. This allows us to calculate the potential distribution along the simulation domain that includes any complex objects inside. Fig. 59 shows the potential distribution in vacuum (the RHS charge density vector was set to 0) from a test case with one rectangular object and a small aperture inside it. A -2.8 V potential is applied at the top boundary of the simulation domain (*z*(0)) and a 0 V potential is used at the object boundary. The object is clearly visible in the figure together with the aperture inside. The potential distribution is as we would expect.

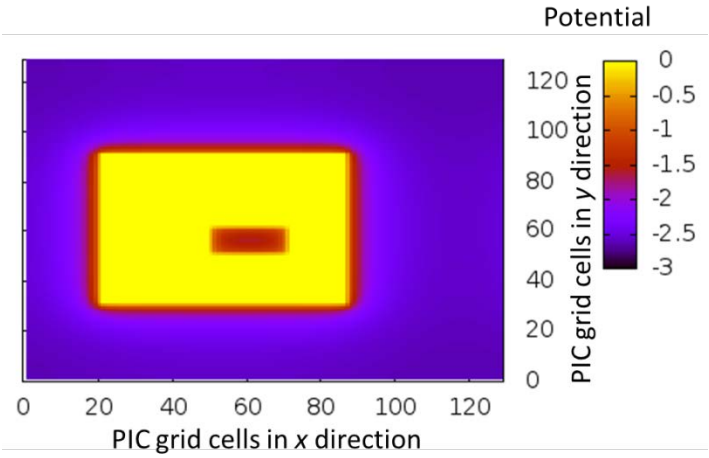

Fig. 59 Potential distribution in vacuum at the *x*-*y* plane (*z* = 5) obtained from the parallel PETSc Poisson solver using a 128×128×129 PIC grid, periodic boundary conditions in the *x*-

and *y*-directions and a Dirichlet boundary condition in the *z*-direction, respectively (*z*(0) = -2.8 V, *z*(129) = 0 V).

In order to prove the correctness of our PETSc solver with internal objects we compare our potential distribution with the results obtained from the serial multigrid solver for the same test case. Fig. 60 shows the vacuum potential distribution for a similar test case presented in the Fig. 59 but with an object boundary potential of -2.8 V. The obtained results from both solvers are on top of each other. Therefore, the results confirm the correctness of our solver for the vacuum testcase including internal objects.

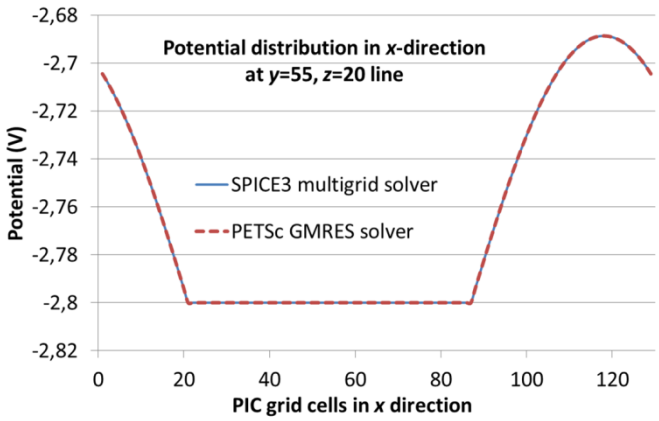

Fig. 60 Potential distribution in the *x*-direction obtained from the parallel PETSc GMRES Poisson solver (red line) and the SPICE3 multigrid Poisson solver (blue line) using one rectangular object inside the domain. Periodic boundary conditions were used in the *x*- and *y*-directions and a Dirichlet boundary condition was used in the *z*-direction (*z*(0) = -2.8 V, *z*(129) = 0 V). A potential of -2.8 V was applied at the boundary of the object. The simulation domain was distributed among 64 MPI tasks in all three directions.

5.8. *Poisson solver with internal objects and charge density vector*

The construction of the RHS vector was modified to consider internal objects. It includes a constant potential at the objects boundaries and modifications of the charge density matrix near the objects. Fig. 61 shows the obtained potential distribution in two planes from both the SPICE3 multigrid sequential solver (blue line) and from the PETSc GMRES parallel solver (red line) for a test case with 64 MPI tasks (four in the *x*-, four in the *y*- and four in the *z*- direction, respectively) and with three rectangular internal objects. The obtained results from both solvers are on top of each other for both figures. Therefore, these results confirm the correctness of our solver with internal objects. It provides identical results in comparison to the original SPICE3 multigrid solver.

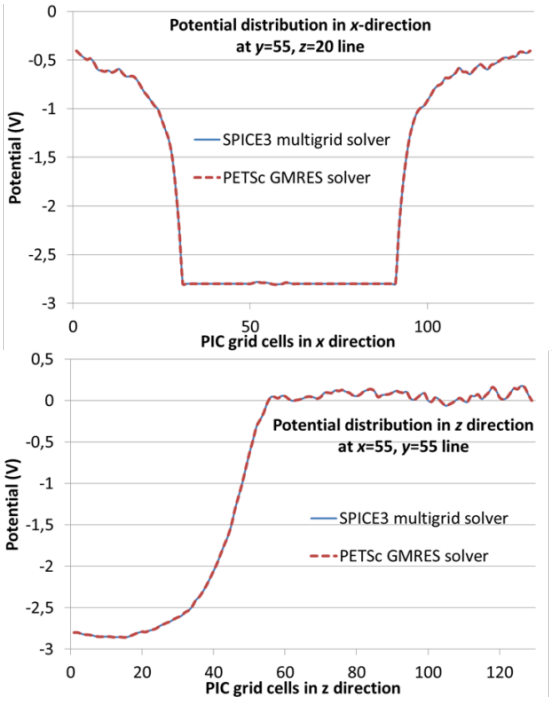

Fig. 61 Potential distribution in different directions obtained from the parallel PETSc GMRES Poisson solver (red line) and SPICE3 Poisson solver (blue line) using the right hand side data from a real test case with internal objects inside the simulation domain. Periodic boundary conditions were used in the *x*- and *y*-directions and a Dirichlet boundary condition was used in the *z*-direction ($z(0) = -2.8$ V, $z(129) = 0$ V). The simulation domain was distributed among 64 MPI tasks in all three directions.

A possibility to apply different potential at each internal object was developed in the end of the project. This potential can be changed at each time step, as it is required in the SPICE code. The project coordinator tested the modified subroutine and it provides identical results in comparison to the original multigrid solver.

5.9. *Test of different PETSc solvers*

The PETSc library has the possibility to change the solver type via one command: *KSPSetType* (discussed in Sec. 1.3). The default solver, which was used in all previous tests, was *KSPGMRES* (Generalized Minimal Residual). We know from [5] that the *GMRES* solver is not the optimal one if a symmetric positive definite matrix (SPICE case) is involved in the computation. Therefore, we executed a performance study of different solvers. Three solvers were compared: *KSPCG* (Conjugate Gradient), *KSPBICG* (BiConjugate Gradien) and *KSPGMRES* (Generalized Minimal Residual). For the comparison, a testcase with a 128×128×129 grid, 64 MPI tasks and internal objects was used. As we expected the default *KSPGMRES* solver provides the worse result (1 s). The computation time of *KSPBICG* was measured of 0.75 s. The best timing was obtained from the *KSPCG* solver (0.4 s). Therefore, the *KSPCG* solver was set as the default in our module. However, we left also the possibility to use the other solvers, which might have better timings for other test cases.

5.10. *Performance of the complete SPICE code*

The execution time of the complete PIC cycle of the SPICE code with both the old and the project coordinator measured the new solver. The results are presented in Fig. 62. Using 64 MPI tasks together with the old serial multigrid solver the total execution time per iteration was about 0.9 s. With the new PETSc solver the execution time decreases by a factor of two to 0.48 s. Further improvement of other parts of the code will be worked on in the next HSLT project.

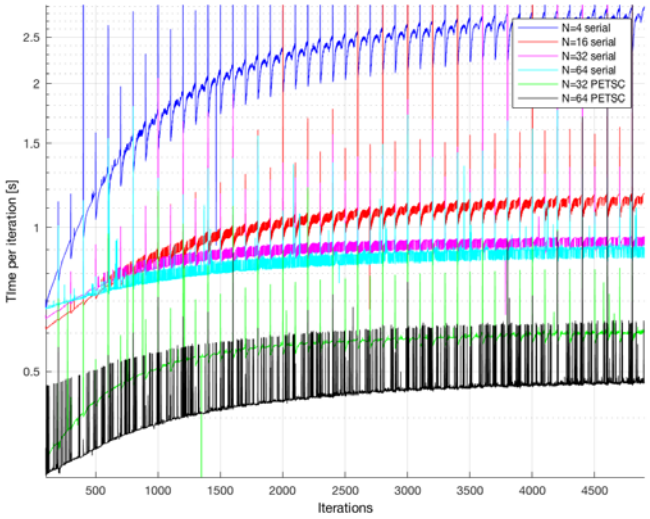

Fig. 62 Wall clock time per complete PIC iteration.

5.11. *Summary*

During the project a 3D PETSc parallel Poisson solver was developed. The solver was validated using a variety of tests using synthetic and real data. The code scales linearly up to 128 MPI tasks and provides good performance. The solver works correctly using both the typical SPICE3 grid (128×128×129) and a larger grid (550×550×550). This is the main goal of this project. The solver was also tested on real data from the SPICE3 code providing identical results to the serial multigrid solver for test cases that include internal objects inside the simulation domain and different potential at each object. The solver has been integrated into the SPICE code and production runs was performed.

5.12. *References*

[1] M. Komm, PhD thesis, “Studium okrajového plazmatu Tokamaku a jeho interakce s první stěnou”, Praha 2011
[2] Z. Pekarek, PhD thesis, “Advanced techniques of computer modelling in low- and high-temperature plasma physics”, Prague 2012
[3] M. Komm et al., Nucl. Fusion 57 (2017)126047
[4] J. Coenen et al., Nucl. Fusion 55 (2015) 23010
[5] https://www.mcs.anl.gov/petsc/

6. Report on HLST project CINCOMP3 – Part2

6.1. *Introduction*

The CINCOMP3 project is the continuation of the CINCOMP and CINCOMP2 projects. These projects are dedicated to provide support for specific issues on the supercomputers of the EUROfusion consortium.

The HLST currently investigates performance fluctuations of production codes. The fluctuations gave rise to the idea that the previously examined fluctuations in the MPI_Barrier completion times might still be an issue and that they might lead to these performance fluctuations. The reason for making this connection is mainly rooted in the fact that the detected fluctuations are not Gaussian distributed. The run time of the simulations has a very frequently occurring baseline, which is sporadically disturbed by runs which take more than 20% additional time on SKL. This seems to fit the MPI_Barrier measurements made during the CINCOMP2 project, which showed a very stable mean sample time, interspersed with few very large outliers.

In order to make the results more meaningful it was decided to benchmark the MPI_Allreduce function instead of the MPI_Barrier function, since the offending production codes use this function a lot. MPI_Allreduce should also be unaffected by any process skew as its runtime is orders of magnitude larger than the run time of MPI_Barrier (which introduces the process skew). A message size of 1 MiB in combination with the MPI_SUM operation will be used for all tests which employ the MPI_Allreduce function.

6.2. *Runtime Spread for MPI_Allreduce*

The standard benchmark configuration for all tests in this chapter is the following:

- MPI_Allreduce function using 1 MiB messages and the MPI_SUM operation
- Process skew elimination disabled
- For each sample, the maximum time over all ranks is used.

A schematic of the general test structure, using this configuration, can be found in Listing 1.

```
for (int i = 0; i < 1000000; i++) {
  MPI_Barrier(comm);
  start = time();
  MPI_Allreduce(sendb, recvb, 1048576, MPI_UNSIGNED_CHAR, MPI_SUM, comm);
  end = time()
  MPI_Reduce(end - start, &time_max, 1, MPI_LONG_LONG_INT, MPI_MAX, 0, comm);
}
```

Listing 1 MPI_Allreduce benchmark structure.

Fig. 63 and all subsequent figures in this chapter use the `time_max` value, as detailed in Listing 1, and plot them in a double logarithmic histogram. The y-axis is scaled to give the percentage per bucket instead of the total amount of samples in order to facilitate a comparison to other tests. The dashed vertical bars show the location of the mean value. A Gaussian fit is superimposed onto the histogram (if it converged) and the legend includes the percentage of samples under this fitting function. The legend also includes the number of samples and, for the orange plot only, the factor between the mean run times, which is equal to the factor between the total run times of both benchmarks normalized to the number of samples.

6.2.1. Marconi performance for different configurations

The first couple of benchmarks are concerned with the performance of Marconi for different configurations.

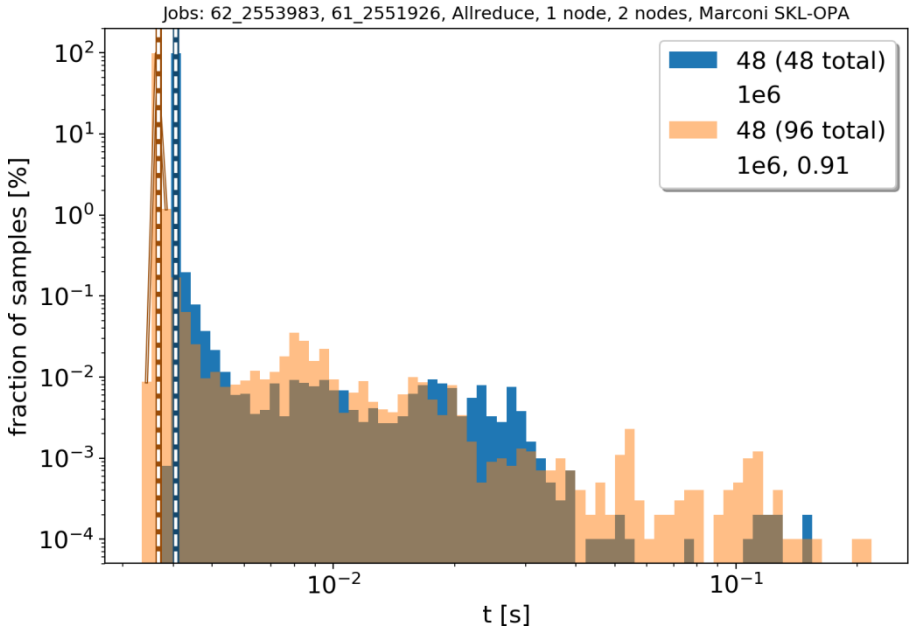

Fig. 63 Marconi SKL performance: Comparison between one full node and two full nodes.

Fig. 63 shows a comparison between one and two nodes on Marconi SKL. The difference is minimal. The single node run finishes slightly faster, but both runs exhibit large fluctuations in the sample times. This is very comparable to the previously investigated (CINCOMP2) MPI_Barrier behavior. The fact that the results do not change meaningfully when using a single node only, suggests that the interconnect and possible associated interrupts might not be the culprit. If the interconnect bombards the node with interrupts even if it has no messages for this specific node, it might still be at fault.

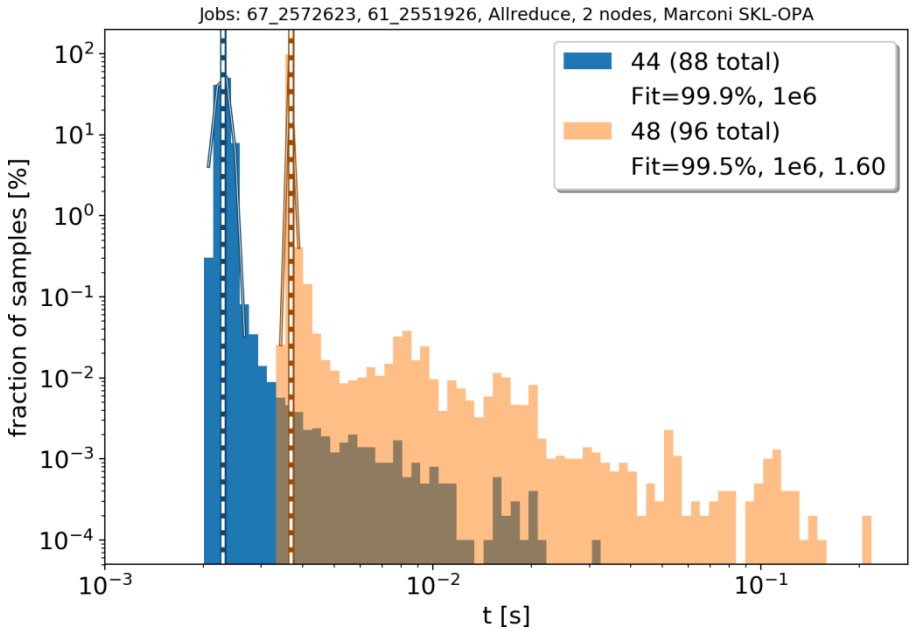

Fig. 64 Marconi SKL performance: Comparison between using 44 cores of each of two nodes and using the full 48 cores of each of two nodes. The free cores are distributed equally over the two sockets.

Apart from a one to two nodes comparison we also look at the performance if we leave multiple cores free on each socket of a node. If hardware interrupts are the reason for the fluctuations, leaving cores free should ameliorate the situation.

Fig. 64 and Fig. 65 show the performance implications of using only 44 or 32 cores per node, respectively. The free cores are always distributed equally over all sockets. The benchmarks use two nodes. The fluctuations are gradually reduced when using

fewer cores per node. For 32 cores, we measure a significant speed-up, which could also be related to this number being a power of two.

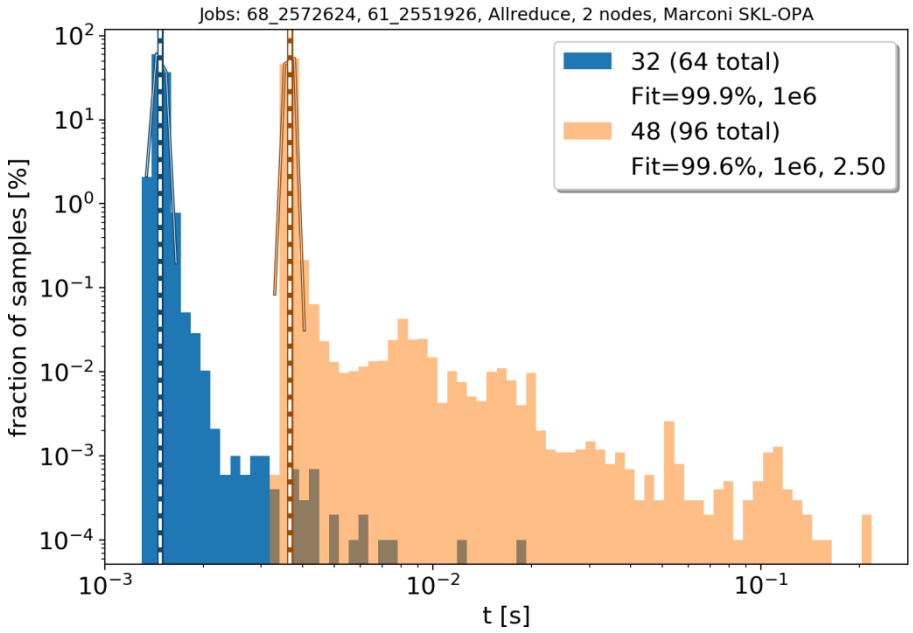

Fig. 65 Marconi SKL performance: Comparison between using 32 cores of each of two nodes and using the full 48 cores of each of two nodes. The free cores are distributed equally over the two sockets.

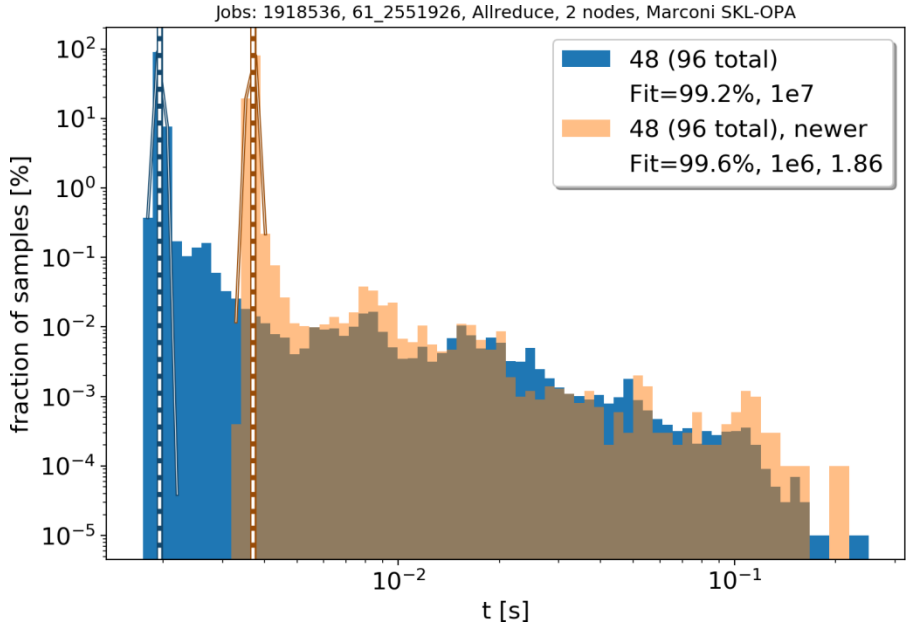

Fig. 66 Marconi SKL performance: Comparison of two complete nodes for two different points in time. The older run finished on the 31st of August, while the new run finished on the 8th of November. A large software update was performed at the beginning of November.

The data for the Marconi benchmarks was taken during the week of the 5th–9th of November. We have earlier data as well. This earlier data was recorded before some significant software updates. There are some strange differences between the old data and the new data, which might be useful to deduce the source of the fluctuations.

Fig. 66 shows a comparison between the results of these two different benchmarking dates using two complete nodes. Apparently, the update made the mean slower by a factor of 1.86. Interestingly, it did not affect the structure of the fluctuations. They still start at the, newly shifted, mean, but their height is the same height as the previous,

older, results. In other words, the fluctuations are not merely shifted with the shift of the mean. They are reduced in width and their height fits the old structure.

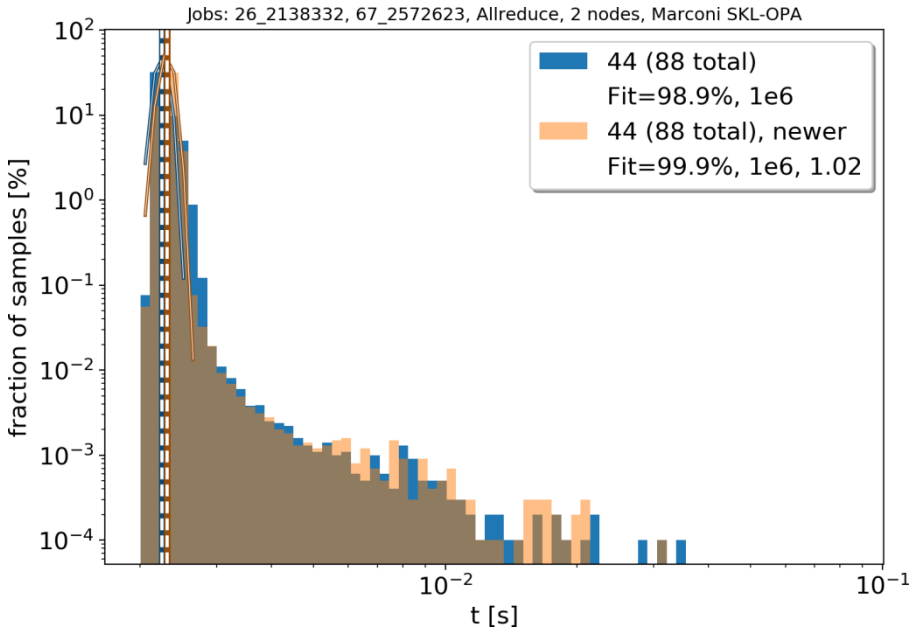

Fig. 67 Marconi SKL performance: Comparison of two nodes using 44 ranks per node for two different points in time. The older run finished on the 27th of September, while the new run finished on the 9th of November. A large software update was performed at the beginning of November. The free cores are distributed equally over the two sockets.

This indicates that there are two different mechanisms at play, one mechanism, which is responsible for setting the mean and another, unrelated mechanism, which is responsible for the generation of the fluctuations.

Fig. 67 shows the comparison between the earlier and the later data for simulations leaving 2 cores per socket free. In this case, the mean did not shift meaningfully. This is a very weird behavior as well.

However, it actually solves a previously unexplained oddity. The old benchmarks had the strange result that the mean performance did not strictly increase with decreasing participating cores. It was actually faster to use 48 cores than to use 44 cores. This was fixed with the software update by making the mean, when using 48 cores, slower.

6.2.2. Marconi Performance in comparison to other systems

This chapter contains benchmark results from six different supercomputers around the world. These benchmarks are compared to the benchmarks of Marconi. These systems offer different hardware configurations, which may help a lot in determining the source of the unexplained fluctuations. We also made sure to have the software configuration on the different systems to be as similar as possible. This includes the MPI_Allreduce algorithm choice configuration, which was set to the Intel default value of

```
1: 0-255 & 0-2147483647
1: 256-511 & 0-63
1: 256-511 & 256-511
2: 512-1048575 & 16-511
2: 256-2097151 & 64-255
2: 1024-2147483647 & 256-2147483647
5: 256-1023 & 512-2147483647
3: 0-2147483647 & 0-2147483647
```

on all systems, except the Japanese machine, where we could not yet verify the configuration. The number before the colon signifies the algorithm, to be cross referenced with [1]. The first range of numbers stands for the amount of bytes in the

messages which are in the communication, and the second range, after the ampersand, stands for the number of ranks participating in the function call. The lines are read from top to bottom until a suitable configuration for the current function call is found. This configuration means that algorithm 2, Rabenseifner's algorithm, corresponding to the fourth line, is used for all our benchmarks.

If hardware interrupts are the source of the fluctuations, they may be isolated to different architectures and interconnects. If that is the case, we should be able to pinpoint specific hardware configurations and request improvements for these specific systems.

The first system under investigation is the Draco supercomputer at the MPCDF in Garching. It is Haswell (Xeon E5-2698) based and uses an InfiniBand FDR14 interconnect. Fig. 68 shows no large fluctuations and slower mean performance. This would indicate that InfiniBand is not affected by hardware fluctuations.

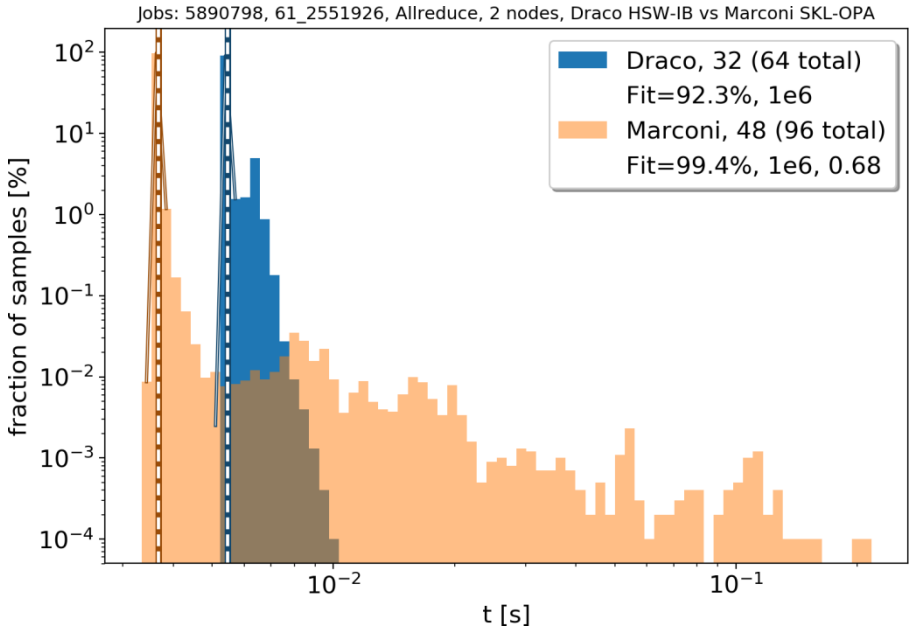

Fig. 68 Comparison between Marconi (SKL, Omni-Path) and Draco (HSW, InfiniBand) at MPCDF, Germany

The second system to compare with Marconi is the MareNostrum 4 supercomputer of the Barcelona Supercomputing Centre. It is equipped with Intel Xeon Platinum 8160 processors and an Intel Omni-Path (OPA) interconnect. Fig. 69 shows that this machine also exhibits large fluctuations, albeit with a slightly smaller width. The mean performance is slightly better as well. This still conforms to the theory that SKL-OPA has an issue with hardware interrupts.

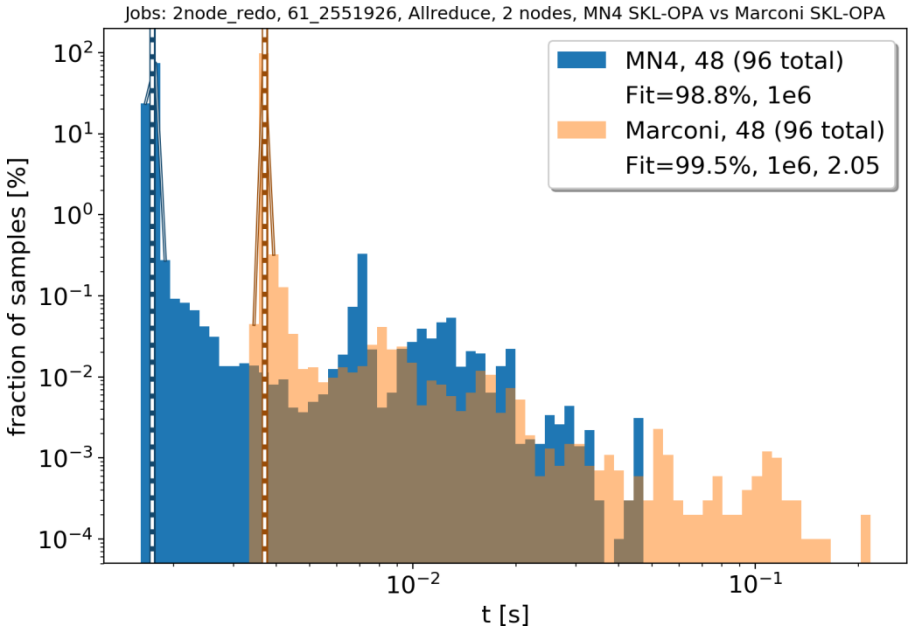

Fig. 69 Comparison between Marconi (SKL, Omni-Path) and MareNostrum 4 (SKL, Omni-Path) at BSC, Spain

The third comparison system is the JUWELS machine at the Juelich Supercomputing Centre. It uses Intel Xeon Platinum 8168 CPUs and the Intel EDR-InfiniBand (Connect-X4) interconnect. According to the data in Fig. 70, this system confirms our theory, as InfiniBand does not seem to suffer from increased fluctuations.

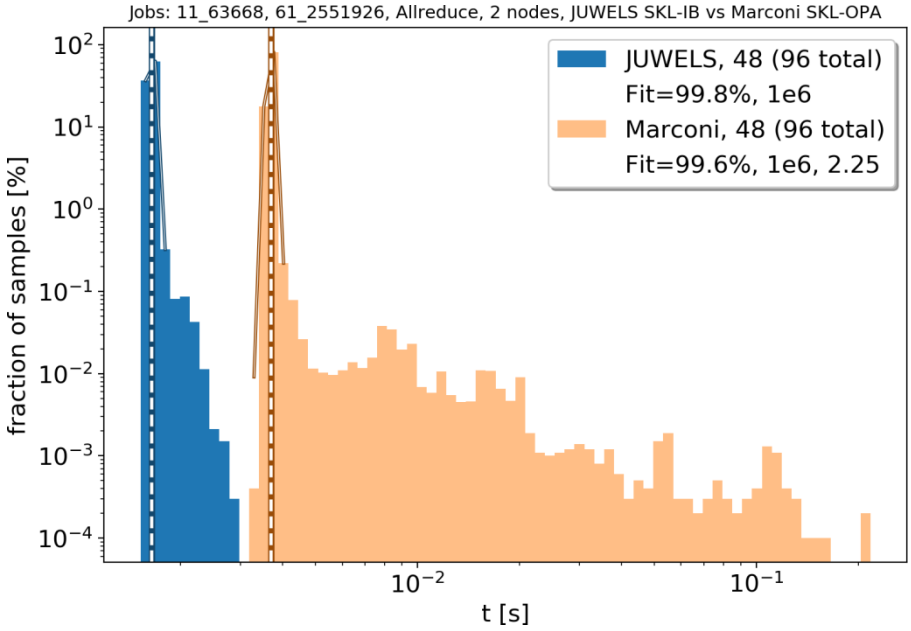

Fig. 70 Comparison between Marconi (SKL, Omni-Path) and JUWELS (SKL, EDR-InfiniBand, Connect-X4) at JSC, Germany

Unfortunately, the trend of confirming our hardware interrupt theory stops with the next two results. We performed our benchmarks on the Cobra system at MPCDF (SKL, Omni-Path) and the Irene machine at TGCC in Bruyères-le-Châtel (SKL 8168, InfiniBand EDR). The resulting histograms can be found in Fig. 71 and Fig. 72, respectively. They both contradict our theory. The Cobra system is an Omni-Path system without fluctuations while the Irene system is an InfiniBand system with fluctuations.

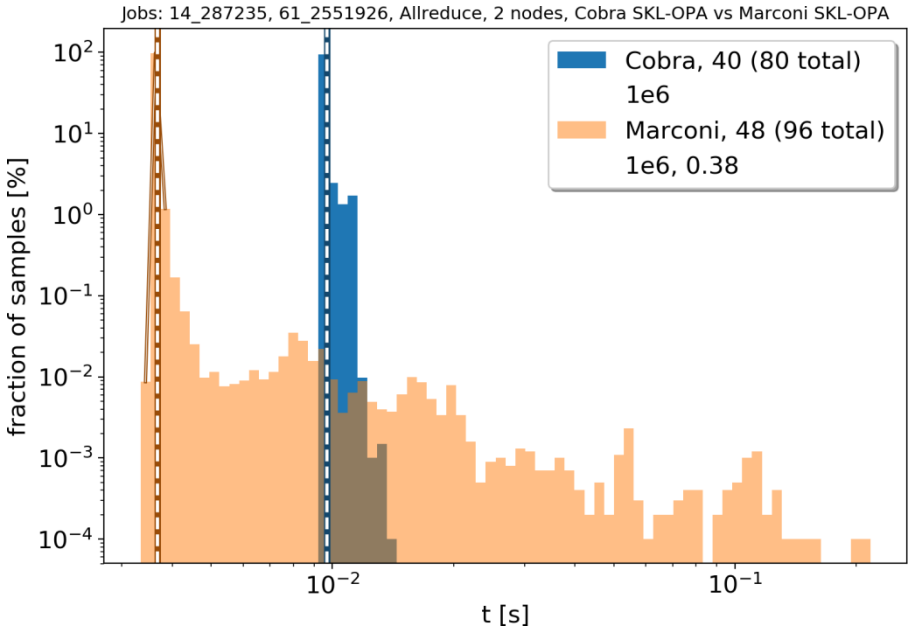

Fig. 71 Comparison between Marconi (SKL, Omni-Path) and Cobra (SKL, Omni-Path) at MPCDF, Germany

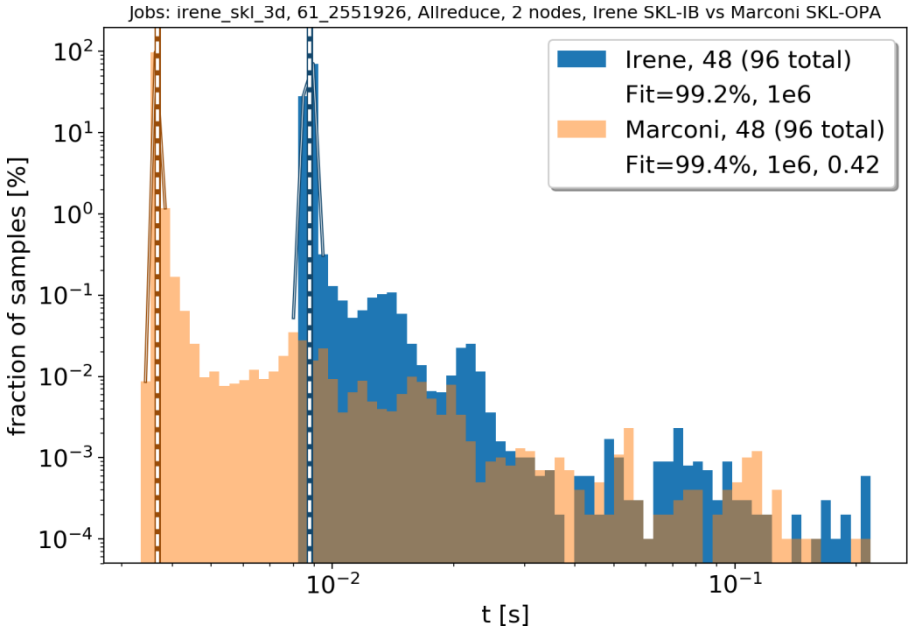

Fig. 72 Comparison between Marconi (SKL, Omni-Path) and Irene (SKL, InfiniBand) at TGCC, France

Finally, we were able to run our benchmark on a system with a non-Intel interconnect, namely the Cray SKL Aries machine at QST, Rokkasho Fusion Institute. Fig. 73 shows that this system performs well and does not have problems with increased fluctuations.

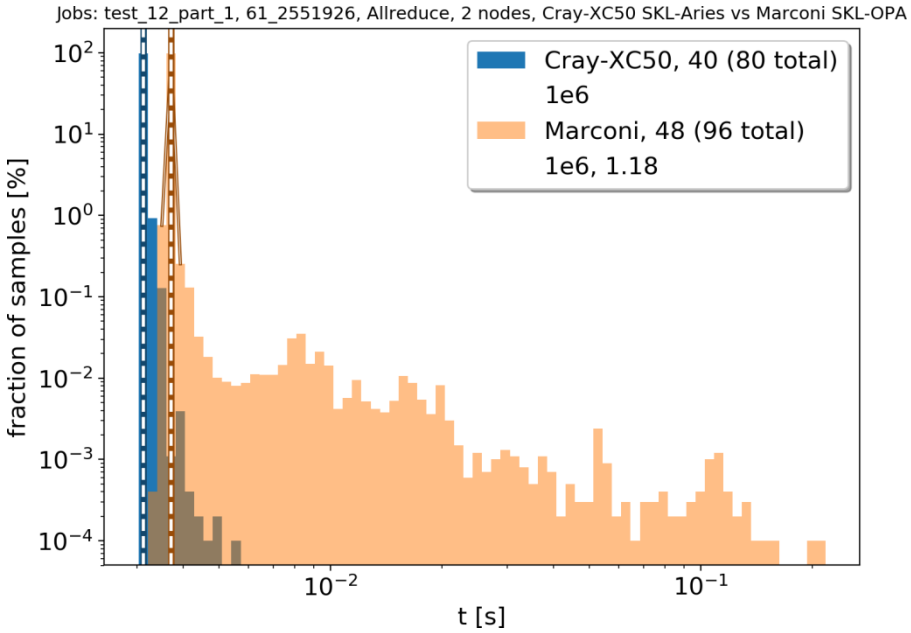

Fig. 73 Comparison between Marconi (SKL, Omni-Path) and Cray-XC50 (SKL, Aries) at QST in Rokkasho, Japan

After going over the detailed results of the different systems, Fig. 74 presents a coarser but summarized plot of the data. It shows that the behavior is very different across the different systems.

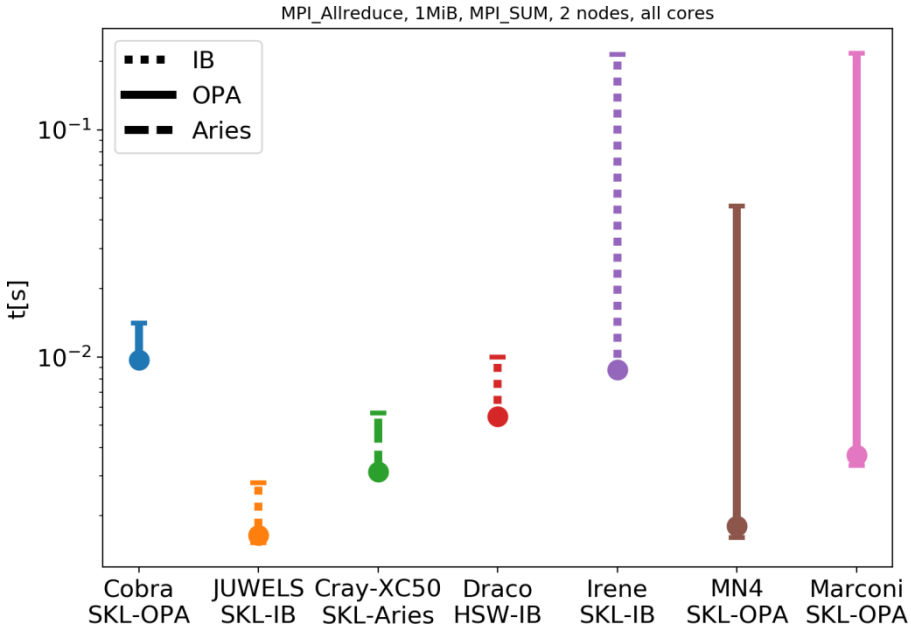

Fig. 74 Comparison between all investigated systems. The colored circles give the mean completion time of the MPI_Allreduce benchmark for each system. The error bars are made via finding the minimum and maximum sample completion time for each specific benchmark. This is the same as the time value of the leftmost and rightmost bar in the previously shown figures for the respective system. This error bar represents the width of the fluctuations. The bars are arranged from left to right in order of increasing width. Please note that the length of the error bar is given by the ratio of the lower and upper boundary of it, not the difference. This is due to the logarithmic axis. The error bar therefore gives the relative error, not the absolute error.

Comparing the mean values is not completely fair, as the investigated systems have very different characteristics. The amount of cores per node varies between 32 for the Draco system and 48 for most of the other systems. The different interconnects have strongly varying dates of entering the market as well. Despite this, a comparison between the fluctuations should still be perfectly viable. Fig. 74 shows

that the fluctuations do not depend on the used interconnect or the employed architecture. This suggests that there is room for improvement for the Marconi machine as it exhibits the largest fluctuations of all investigated systems.

It may be helpful to take note of the used version of the fabric interface library. Marconi and MareNostrum4 use an OFI version of 10.5 (obtained via `rpm -qa | grep opa`), while Cobra uses the more recent version 10.8. Incidentally, Cobra is also the one Omni-Path machine which does not experience increased fluctuations. Unfortunately the newer version will not be available on Marconi or MareNostrum4 for at least three more months, as the GPFS file system, these machines use, is not yet compatible.

6.3. *Summary*

The performance fluctuations of the MPI_Allreduce function were investigated using a custom code. This code measures the run time of single function calls in order to provide data for histograms detailing the mean run time and the standard deviation of the run time (or run time fluctuation).

Analyzing the Marconi A3 partition with this tool while varying several parameters, it was found that it exhibits run time fluctuations of several orders of magnitude in all cases. A very strong effect was found when varying the amount of cores per node. Using fewer cores per node decreases the fluctuations decisively. This fits the hardware interrupt explanation.

In order to identify the specific element, responsible for the fluctuations, we performed the benchmark on six other supercomputers. We can therefore test the architecture and the interconnect independently. The obtained results are very dissimilar. The different machines have mean run times, which differ in the range of half an order of magnitude. The fluctuations are even more diverse and differ on the order of multiple orders of magnitude. Unfortunately, the obtained data does not single out specific architectures or interconnects. There are slow and fast Skylake systems, slow and fast Omni-Path machines, and the fluctuations range from small to large for both parameters. Other interconnects behave in this erratic manner as well.

In any case, the Marconi machine exhibits the largest fluctuations and its mean run time is not the fastest (but not the slowest as well). The investigations were expanded to include several other parameters, like the software version and other configuration parameters. No conclusive result has been obtained so far.

6.4. *Reference*

[1] Intel Documentation. (n.d.). Retrieved from https://software.intel.com/en-us/node/528906

7. Report on HLST project PICOPT

7.1. *Introduction*

The new HLST project PICOPT aims at helping developers of gyrokinetic Particle-In-Cell (GK PIC) codes to leverage the performance benefits of modern supercomputer architectures. To this end, it envisions to provide optimized implementation strategies for common GK PIC algorithms. GK PIC codes have very special capabilities and demands. Recent computer architectures (KNL, SKL) bring special constraints to optimization efforts as well, with vectorization being the major one.

PICOPT was proposed with the hope that the narrow algorithmic definition of GK PIC codes can be exploited in order to provide optimizations for many current simulation codes. Since refactoring and optimization are large-scale code changes which demand very careful examination of the code followed by implementation and intensive testing stages, GK PIC codes are especially suitable and worthwhile for this task as they are a code class which consumes proportionally high amounts of computer resources.

The first step for this project is a thorough research into the optimization strategies of different PIC codes, which are used in the community at the moment. Through the examination of these codes, we hope to get a good understanding of possible optimization venues.

From the analysis of existing GK PIC codes, a best-practice guide for optimizing GK PIC codes can be prepared. It should comprise the most efficient way to do GK PIC simulations by leveraging the learned methods as well as further tweaking and improving them. Using this guide, other existing gyrokinetic GK PIC codes can then be optimized.

7.2. *Common PIC algorithm*

PIC codes aim to solve the relativistic or non-relativistic Vlasov-Maxwell system of equations. They can be extended to include collisions, which enable them to solve the Boltzmann-Maxwell system of equations. The problem can therefore be fully described by the following equations. The Vlasov equation, which can be derived from the Liouville equation, is given by

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} f + q(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) \cdot \frac{\partial f}{\partial \boldsymbol{p}} = 0$$

where $f(\boldsymbol{r}, \boldsymbol{p}, t)$ is the particle density distribution function, with $\boldsymbol{r}$ being the spatial coordinate, $\boldsymbol{p}$ being the momentum and t the time, q is the particle charge, $\boldsymbol{v}$ is the velocity of a characteristic, $\boldsymbol{E}$ is the electric field and $\boldsymbol{B}$ is the magnetic field. Vector values are written in bold face. Maxwell's equations are given by

$$\boldsymbol{\nabla} \cdot \boldsymbol{E} = \frac{\rho}{\epsilon_0}\,,\ \boldsymbol{\nabla} \cdot \boldsymbol{B} = \boldsymbol{0},$$

$$\boldsymbol{\nabla} \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t} \text{ and } \boldsymbol{\nabla} \times \boldsymbol{B} = \mu_0 \left(\boldsymbol{J} + \boldsymbol{\epsilon_0} \frac{\partial \boldsymbol{E}}{\partial t} \right),$$

where ρ is the charge density, ϵ_0 is the permittivity of the vacuum, μ_0 is the permeability of free space and $\boldsymbol{J}$ is the current density.

These form a system of equations since the charge and the current density need to be derived from the particle density distribution function and since the electric- and magnetic fields are needed to solve the Vlasov equation.

PIC codes employ so-called markers or quasi-particles in order to solve the Vlasov equation kinetically using a Monte Carlo approach. The advantage, in comparison with solving the full fluid problem, is the fact, that the complexity of the problem is reduced. A full fluid code needs a 6D grid to represent the available particle phase-space density whereas we only need 3D grids when using the PIC scheme. The

computational demand of a PIC simulation does not scale with the dimensionality of the problem since the markers inhabit the phase space without fully spanning it. It also offers an automatic adaption of the resolution as the particles converge and disperse. This is the main reason for the success of the PIC method for fully kinetic simulations.

Maxwell's equations are commonly solved using a finite-difference or finite-element algorithm. The update loop then commonly looks like this:

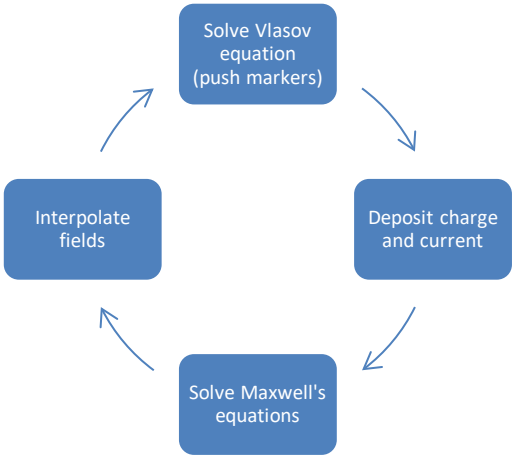

Fig. 75 Illustration of a standard PIC code update step.

7.3. *Common GK PIC algorithm*

Gyrokinetic codes solve the Vlasov-Maxwell system as well, but make use of the special properties of particles in very strong external magnetic fields to perform the computations much more efficiently.

The three main exploited properties are:

1. The huge separation of scale between the fast gyro-rotation around the magnetic field lines and the dynamics of the gyro-center. This is the most important property and is explained in detail further on.
2. The magnetic field restricts the movement of charged particles in a significant manner, which makes it possible to ignore one velocity dimension.
3. The electron gyro-radiuses are often much smaller than the electromagnetic mode lengths, which enables a significantly simplified treatment of electron dynamics.

The first property is the most important property for the subsequently explained optimization efforts. Providing the temporal resolution to accurately represent the fast gyro-rotation increases the computational demand by approximately a factor of one million in comparison to the optimized version.

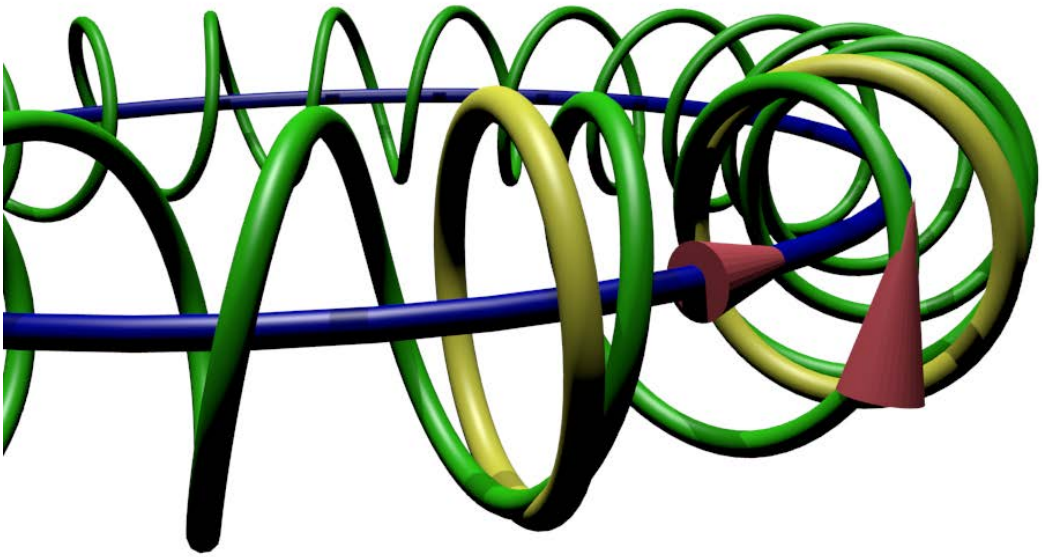

Fig. 76 Sketch of trajectories in a strong magnetic field. An approximation to the actual particle trajectory is shown in green. The gyro-radius approximation uses markers, which follow the blue guiding-center trajectory, while interacting with the electro-magnetic fields as a charged ring. Two examples of the charged ring, or gyro-ring, positions are shown in yellow. The full approximating trajectory for the charged rings would be a hollow torus around the guiding-center. The red arrows show the movement direction of the particles and the approximating markers.

In order to get rid of the vastly different scales between the gyro-rotation and the guiding-center motion, gyrokinetic PIC encapsulates the gyro-movement of the particles in the analytic description of the problem. In order to do this it introduces so-called gyro-rings. These rings represent the fast particle movement in an approximate manner and are used instead of ordinary charged particle markers when doing charge deposition and field interpolation. A sketch of the correct particle trajectory and the approximation can be found in Fig. 76. The particle charge is smeared out over the ring into multiple charge fractions. This is shown in Fig. 77. The modifications to the formulas make sure that the behavior stays consistent with the model.

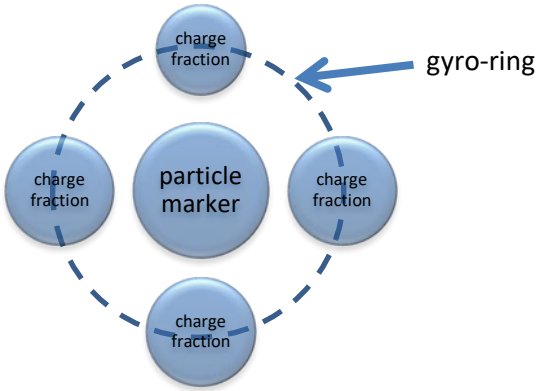

Fig. 77 Particle marker with charge fractions (gyro-points) arranged in a circle around it. The distance of the fractions to the marker is defined by the gyro-radius $r = \frac{mv_{\perp}}{|q|B}$. The number of charge fractions may be adapted to the size of the gyro-ring. The charge fractions are used instead of the marker only in the charge deposition and field interpolation step. The circles in the diagram do not refer to actual circles as the charge fractions and particle markers are treated as point-like in the algorithm.

There are also many other optimizations to this standard algorithm which are not explained in detail in this report as they were not changed in the examined codes.

7.4. *SIMD and vectorization*

The PICOPT project has the objective of finding common optimization patterns to make GK PIC codes perform well on the current and future generation of computing hardware. Due to the stagnating processor frequency, forced by the power consumption and physical limits on the structure size of integrated circuit production, hardware development has switched its focus to a different advancement category.

Starting 2009, Intel as well as competitors made architectural changes, which aimed at incorporating vector processing capabilities into their products. Vector processing is a parallelism paradigm, which targets the latest stages of code execution. Single Instruction Multiple Data (SIMD) patterns in the code base may be recognized and exploited by the compiler in order to include a transparent parallelization in the resulting binary.

The first extended instruction set, the Streaming SIMD Extensions (SSE), offered parallelism on up to 128 bits. This has been improved upon with the AVX instruction set, introduced 2012, which offered up to 256 bits for parallelization. Today, the current generation of Intel Chips codenamed Knights Landing and Skylake offer vectorized computation on up to 512 bits.

The enlargement of the vector registers were accompanied by an ever-expanding list of specialized instructions working on these registers.

Due to the aforementioned stagnating processor frequencies, the importance of an effective utilization of these new capabilities is growing.

7.4.1. Exploitation of Vector computing capabilities

In general, there are three ways for the utilization of vector capabilities:

1. Writing assembly code using the respective instructions
2. Manual vectorization in a higher language using intrinsics
3. Reliance on auto vectorization by the compiler

The first two options are not portable and require new code for every new architecture. The last option is completely portable. Because of this, further explanations will focus solely on the last option.

7.4.2. Writing auto vectorizable code

The first requirement for auto vectorization is the usage of the proper compiler flags. The compiler needs to be told about the vectorization capabilities of the target machine. These flags change for each instruction set, compiler vendor and language. Additionally, some compilers will only use auto vectorization for certain minimum optimization levels (`-On` flag, where commonly $n \geq 2$) and may need the permission to ignore certain numerical math restrictions in order to vectorize math functions (for example `-ffast-math` for GCC).

If configured correctly the compiler will look for vectorization opportunities. These are commonly mainly found in loops. A loop is eligible for vectorization if it

- has a countable number of iterations during run time. This implies that the loop counter and the loop range are not modified inside the loop and that there are no other entries and exits into or out of the loop.
- has a very small amount of branching. Since vectorization relies on the fact that the same operations are executed on different data elements, branching will hinder vectorization. Modern vector instruction sets, which contain masking capabilities, make some branching permissible.
- is the innermost loop. If a previous different optimization technique, like loop unrolling, collapsing or interchange, transforms a loop into the innermost one, it may become eligible for vectorization. Current compilers are also able to vectorize outer loops but need explicit commands and a high optimization level to do so.

- contains no function calls. Exceptions are inlined functions and intrinsic math functions for which the compiler has access to vectorized versions. OpenMP 4.0 introduced the `omp declare simd` clause which makes it possible to describe the vectorization properties of user functions, enabling the compiler to provide a vectorized version.
- contains only aligned and contiguous memory access. If the memory is not aligned to the vector register size, the compiler may add a peel loop at the beginning and a remainder loop at the end to account for unaligned data. In many cases vectorization will not be done, as the overhead for these loops makes it inefficient. The Fortran compiler flag `-align array64byte` will guarantee proper alignment for AVX512 for most memory locations, for example. Other languages need extra code to be added. Non-contiguous memory access may be vectorized using gather and scatter instructions, which are available on the newest architectures. Using the loop counter as the array index makes auto vectorization highly likely. Indirect addressing, for example using an array as an index, makes auto vectorization highly unlikely.
- contains no data dependencies between different iterations of the loop. Certain languages need help from the programmer in order to properly determine pointer aliasing properties. The newest architectures offer conflict detection instructions, which may help in vectorizing data dependent loops.

Following these rules is a deliberate and complex endeavor. It should therefore only be done for the most resource intensive parts of a code base. It can be beneficial to separate these parts from the less critical functions of the code. The subsequentially discussed ORB5 code does this by designating some code parts as kernels. These kernels are optimized and may not be as accessible to new developers as the rest of the code.

It is very important to note that many of these requirements have relaxed somewhat in the last years and continue to soften as the compiler development advances. Using new OpenMP SIMD directives, for example, the developer can instruct the compiler to generate vectorized function versions, introduce extra storage for large quantities of variables on the stack and unroll loops that are more complex.

7.5. *ORB5 optimizations*

ORB5 is a global delta-f Lagrangian GK PIC code, which solves the GK equations and is suited for tokamak simulations.

This code was chosen to be the first one to be investigated in the framework of the PICOPT project, because it was heavily modified and optimized in the last years. These optimizations were specifically targeting new vector processing capabilities of modern supercomputer architectures. The optimizations were performed by the Swiss Platform for Advanced Scientific Computing (PASC), a platform coordinated by the CSCS, the Swiss National Supercomputing Centre of the ETH Zurich in collaboration with the Università della Svizzera italiana and the other Swiss universities as well as the EPF Lausanne. It aims at enabling a tight collaboration between scientist and software developers in order to leverage the computing power of highly parallel computer systems.

7.5.1. **Restructuring gyro-points to be first class objects**

Most common GK PIC implementations create the necessary gyro-ring charge fractions on the fly during the charge deposition and field interpolation steps. They are not persistently stored as a whole and only ever present for short times and for the currently evaluated marker. You can see a visualization of this process in Fig. 75.

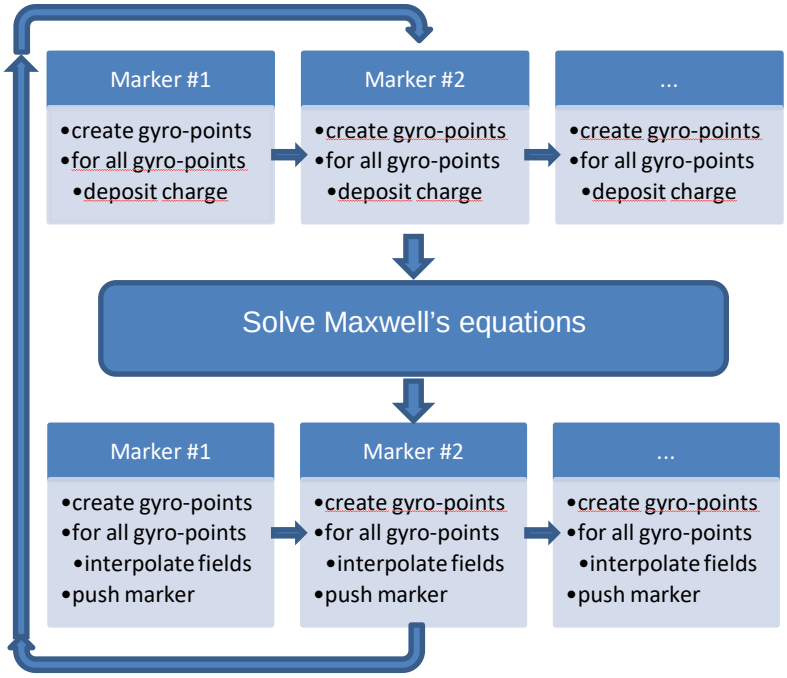

Fig. 78 Standard GK PIC marker and field update loop.

The most visible and prominent restructuring effort of ORB5 changes this algorithm. As a result, the code now contains new permanent arrays of gyro-point data, which contain all charge fractions from all markers. The differences in the update algorithm are outlined in Fig. 79.

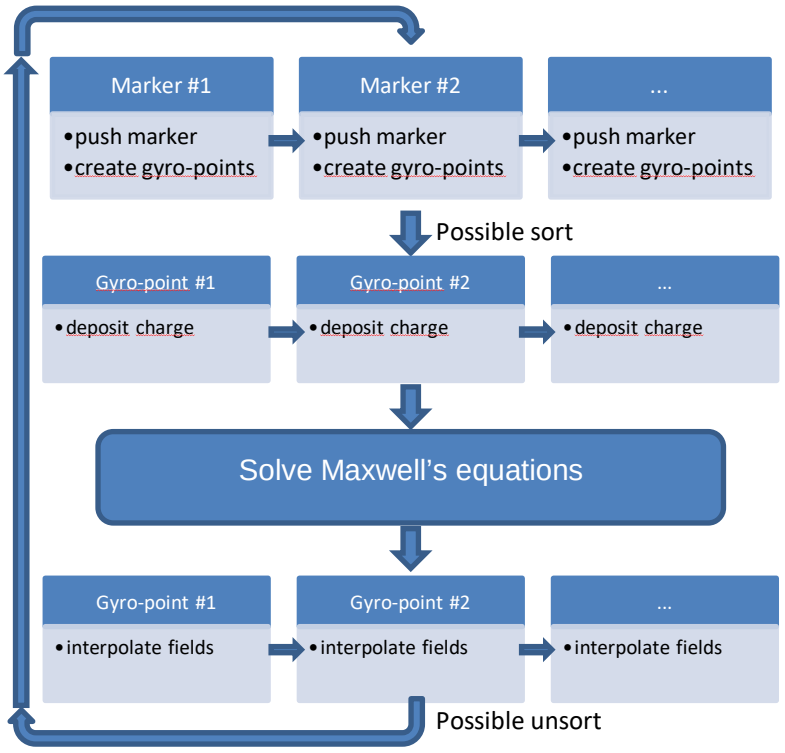

Fig. 79 Modified GK PIC marker and field update loop.

Advantages to the new structure:

- Less computation duplication
- When coupled with cell-wise sorting: less cache misses
- Better vectorization properties since the gyro-point loop is simpler
- When coupled with a hybrid implementation and a color charge-assignment scheme (both explained in 7.5.4) as well as cell-wise sorting: no grid data duplication

- The particle update scheme is close to a standard PIC particle update scheme, which may enable code reuse from standard PIC codes.

Disadvantages to the new structure:

- Larger memory footprint
- More complicated code structure
- When coupled with cell-wise sorting: sorting and unsorting overhead

7.5.2. Charge deposition using the four-color scheme

Standard PIC codes frequently make use of shared memory (for example through OpenMP) in order to parallelize the marker processing.

When depositing the charge of multiple markers, memory access to field values can result in race conditions if multiple markers are deposited by different threads at the same time. A solution to this problem is given by [1]. It proposes to partition the grid into multiple distinct regions following the pattern shown in Fig. 80. The size of the colored regions must be chosen large enough so that marker depositions in different regions of the same color do not overlap.

An extra loop cycling through the four colors is added to the charge deposition step. During each colors turn, each thread acts on the markers in one group of cells of that specific color. In a first iteration, for example, the first thread is assigned to the first red-colored region, the second thread gets assigned to the second red-colored region and so on. Now each thread freely deposits the charge of its markers without any risk of race conditions. In the second iteration, the process is repeated, using the color green. This is then done as well for the subsequent colors orange and purple.

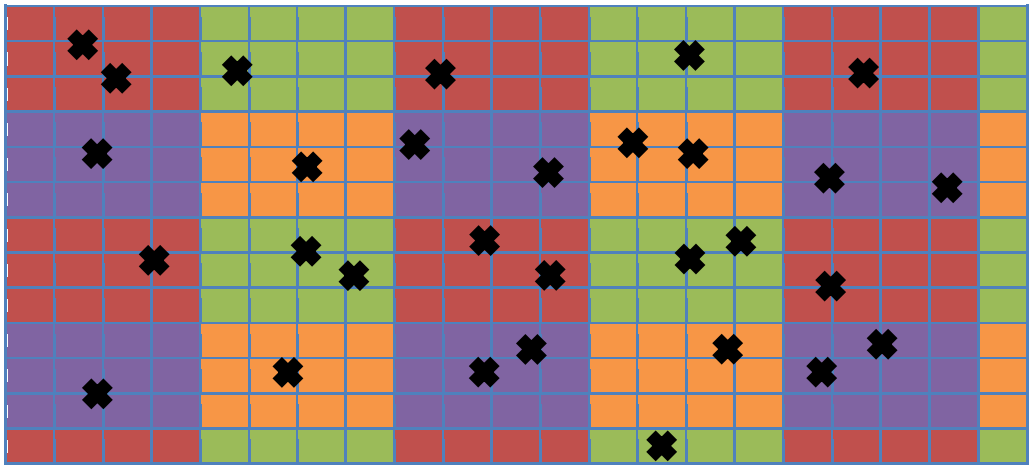

Fig. 80 Example of four color partitioning of a two dimensional grid. The black crosses represent marker locations. The four-color scheme enables the simultaneous update of multiple grid cells. For a detailed explanation of the process, please refer to section 7.5.2.

This optimization is only viable if the charge deposition of the markers is localized, so that the markers may be sorted to specific colored regions. This is not the case for an ordinary GK PIC code, as each marker may need to deposit charge anywhere on its gyro-ring. The restructuring, shown in section 7.5.1, enables the four-color scheme for GK PIC codes.

7.5.3. Further optimizations

In order to explain the effect of the new structure of processing the gyro-points on the performance it may be helpful to elaborate briefly on their meaning in the overall algorithm.

When investigating parallel algorithms it is beneficial to understand the amount of locality present in the underlying problem. PIC codes generally offer high locality in the fields due to the mean field approximation. The fields are the only non-local interaction a particle can have (field theory commonly assigns associated force

carriers to fields which makes them local as well). This is one of the bases of physics and is called the locality principle. Since the only non-local interaction is made local by the mean field approximation, PIC codes are generally fairly well suited to parallelization. Unfortunately, this changes when considering GK PIC. The introduction of gyro-points, which lie on gyro-rings, brings a non-locality to each marker in the system. Each marker acts on and reacts to fields, which are in a certain (dynamic) distance to itself. You can find an illustration of this effect in Fig. 81.

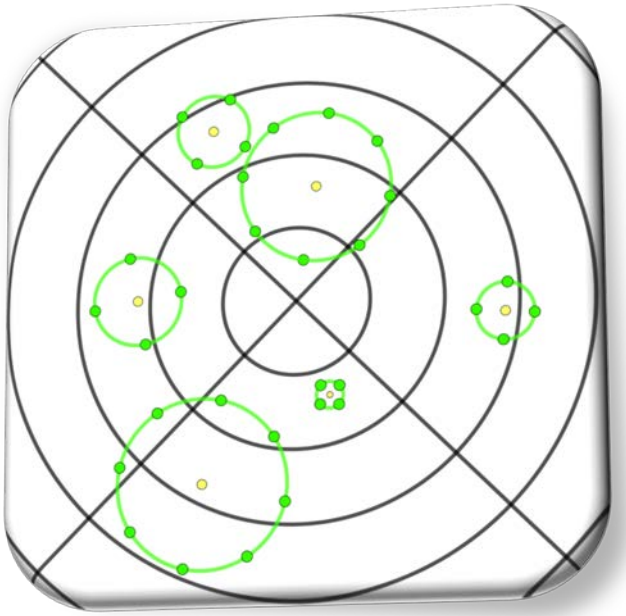

Fig. 81 Illustration of a toroidal slice with markers (yellow) and their associated gyro-points (green) in a polar coordinate system (origin at the center, straight black lines represent points of equal angle, while circular black lines represent points of equal radius). Non-locality can be seen for gyro-points, which lie in a different grid cell than their originating marker.

Due to these important differences, it is not feasible to perform a field parallelization of a GK PIC code in the poloidal space. Non-local memory access will meaningfully disrupt the calculations and severely hamper performance. Most GK PIC codes therefore resort to parallelizing the particles only, keeping copies of the fields in a poloidal slice for each participating core.

The new structure remedies this. The loop interchange can be seen as peeling of the non-locality of the gyro-points during a distinctively non-local sorting process. After this sorting, the resulting gyro-points are local and may be processed as such, enabling parallelization. This new structure cleanly divides the algorithm into local and non-local parts.

This interpretation suggests that further optimizations may either

- target the sorting process and therefore improve on the communication problem.
- target the particle processing and therefore improve on the computation problem.

The second bullet point requires investigation into the auto vectorization properties of the code. We will first suggest a strategy, which aims at contributing to the first bullet point.

The sorting process can be optimized by exploiting locality in the markers. Sorting the markers is a much faster process than sorting the gyro-points since

1. the number of markers is smaller by at least a factor of 4 in comparison to the number of gyro-points.
2. the markers do not switch grid cells as frequently as the gyro-points.

It is possible to optimize the distribution of the colored regions (cf. Fig. 80) in a way that makes the resulting gyro-points require less sorting if the originating markers are sorted to the colored regions. Fig. 82 gives an illustration of the color regions superimposed on the toroidal grid cells. The color scheme only requires sorting to these color buckets not to individual grid cells.

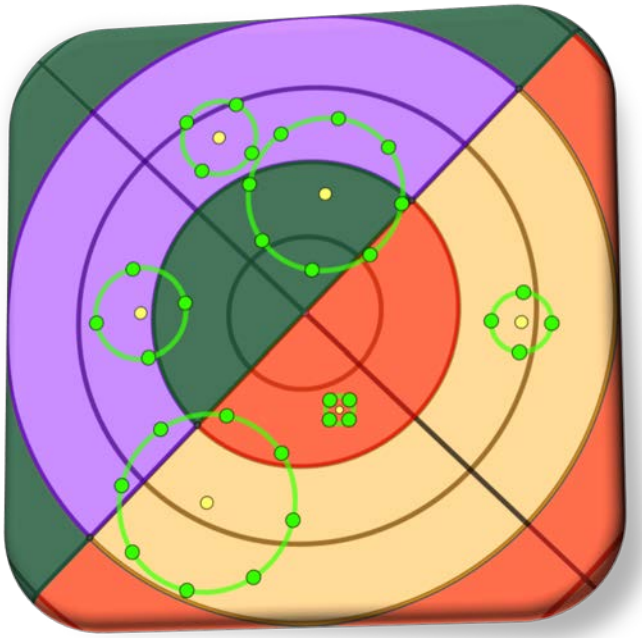

Fig. 82 Illustration of a toroidal slice with markers (yellow) and their associated gyro-points (green) in the polar coordinate system (origin at the center, straight black lines represent points of equal angle, while circular black lines represent points of equal radius). Color regions, in the same colors as in Fig. 80, are superimposed onto the grid cells. This example uses two grid cells per color region in the radial and the angular direction. Non-locality can be seen for gyro-points, which lie in a different color region than their originating marker.

In order to further investigate this we will first try to calculate the amount of gyro-points, which will be generated outside the color region in which their originating marker is located. This is a purely geometric problem. In a first approximation, we will use parallel lines as color regions. The calculation of the fraction of the gyro-radius, which lies outside the color region of its originating marker, is shown in Fig. 83.

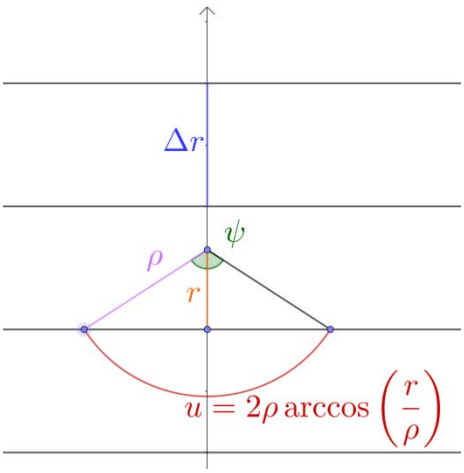

Fig. 83 Fraction u of the gyro-ring which lies outside the color region (approximated as parallel lines) of its originating marker which is located between the second and third parallel line. The color region size is given by Δr, the gyro-radius is given by ρ, and the distance to the grid cell boundary is given by r.

Using this formula, we can calculate the length of the gyro-ring U which will be generated outside the color region of their originating marker for randomly distributed markers in a region:

$$U = 2\int_0^{\Delta r} u(r)\,\mathrm{d}r$$

$$U = 4\rho\left[\Delta r\cos^{-1}\left({}^{r}/_{\rho}\right) - \sqrt{\rho^2 - \Delta r^2}\right]_0^{\Delta r}$$

Finally, we find the fraction f_m of missing gyro-ring length, which is equal to the average amount of missing gyro-points, in relation to the ratio of the gyro-ring and the color region size as

$$f_\mathrm{m}\left({}^{\rho}/_{\Delta r}\right) = \frac{U}{2\pi\Delta r\rho} = \frac{2}{\pi}\begin{cases} {}^{\rho}/_{\Delta r}, & {}^{\rho}/_{\Delta r} \le 1 \\ \cos^{-1}\left({}^{\Delta r}/_{\rho}\right) - \sqrt{{}^{\rho}/_{\Delta r}{}^{2} - 1} + {}^{\rho}/_{\Delta r}, & {}^{\rho}/_{\Delta r} > 1 \end{cases}$$

A plot of this function is given in Fig. 84. Using this plot it is easy to find good values for the color region size Δr, which is the size of the color regions in r-direction. Additionally, it needs to be a multiple of the real grid size.

The value of ρ for real simulations will not be a single value but a distribution as well. This needs to be taken into account when using this formula to find the amount of mismatched gyro-points. Integrating the formula over the ρ distribution will yield the final number.

For $\rho = \frac{1}{3}\Delta r$, for example, more than 80% of all gyro-points will automatically be generated in the correct color region and will not need any further sorting. This makes it possible to choose a different sorting algorithm, ideally an algorithm, which is optimized to sort highly presorted data. The best algorithm would probably be "insertion sort", but "bubble sort" might be a good candidate as well. Benchmarks of the final implementation are needed in order to find the optimal one.

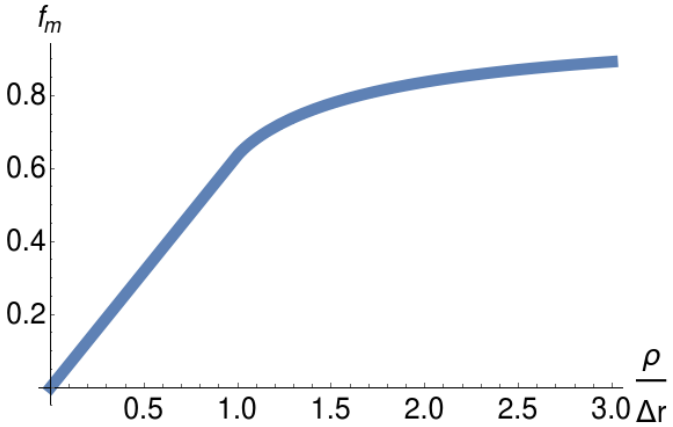

Fig. 84 Fraction f_m of gyro-points missing in the cell of the originating marker in relation to the ratio of the gyro-radius ρ and the grid size Δr.

The above approximation uses parallel lines for the color regions. This is a crude approximation. The approximation works better the farther the investigated region is situated from the origin. In order to improve upon this first approximation we can try to calculate the above value using rings or annuli as color regions. The new formula for u can be found in Fig. 85.

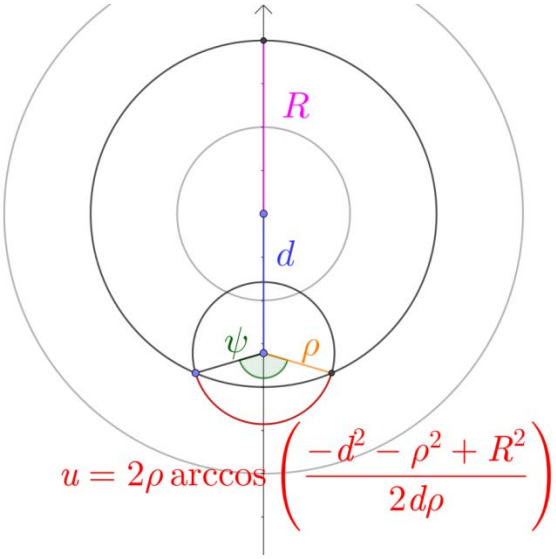

Fig. 85 Length u of the gyro-ring which lies outside on one side of the color region (approximated as annuli) of its originating marker which is located between the first and second circle (with increasing radius). The color region radius is given by R, the gyro-radius is given by ρ, and the distance of the marker to the origin is given by d.

Unfortunately, due to the occurrence of squared variables in the inverse cosine function, integrating this does not yield manageable formulas. It is also hard to find a g so that $f_{\mathrm{m}}(\rho, R) = f_{\mathrm{m}}\big(g(\rho, R)\big)$. This makes finding optimal values much more difficult as the domain of f_{m} therefore remains two-dimensional.

It would also be advantageous to use a single color region in the innermost ring (in other words, creating no angular divisions in the center, cf. Fig. 81), as gyro-rings in that region might otherwise be distributed over many color regions.

The next step would be to generalize to annulus sectors. Since the previous step did not bear any useful results, we did not investigate further in this direction. The result in Fig. 84 must therefore suffice for now. If an implementation of this result shows promising characteristics it may be worthwhile to reevaluate and invest more resources.

7.5.4. MPI-OpenMP hybrid implementation

The initial version of the code had a pure MPI parallelization. As a first step, it used toroidal domain decomposition with a second step of cloning toroidal slices to multiple MPI ranks. The markers are then distributed over the ranks and are unique to each clone. This method works well, but incorporates a lot of MPI communication when aggregating the clones. Furthermore, grid data is duplicated in memory.

The solution to this problem is to employ a hybrid scheme. Each MPI rank gets the ability to employ multithreading to process its markers (using OpenMP). The amount of threads used per MPI process can then be used as an additional leverage to get rid of some communication. Unfortunately, in order to avoid race conditions on the field data, data duplication and reduction is still necessary. This can be avoided by using a color scheme during the charge-assignment step coupled with gyro-points as first class objects. Each MPI rank can then be assigned to one toroidal slice.

A comparison between the different versions can be found in Fig. 86. An explanation of the color-charge-deposition scheme can be found in section 7.5.2.

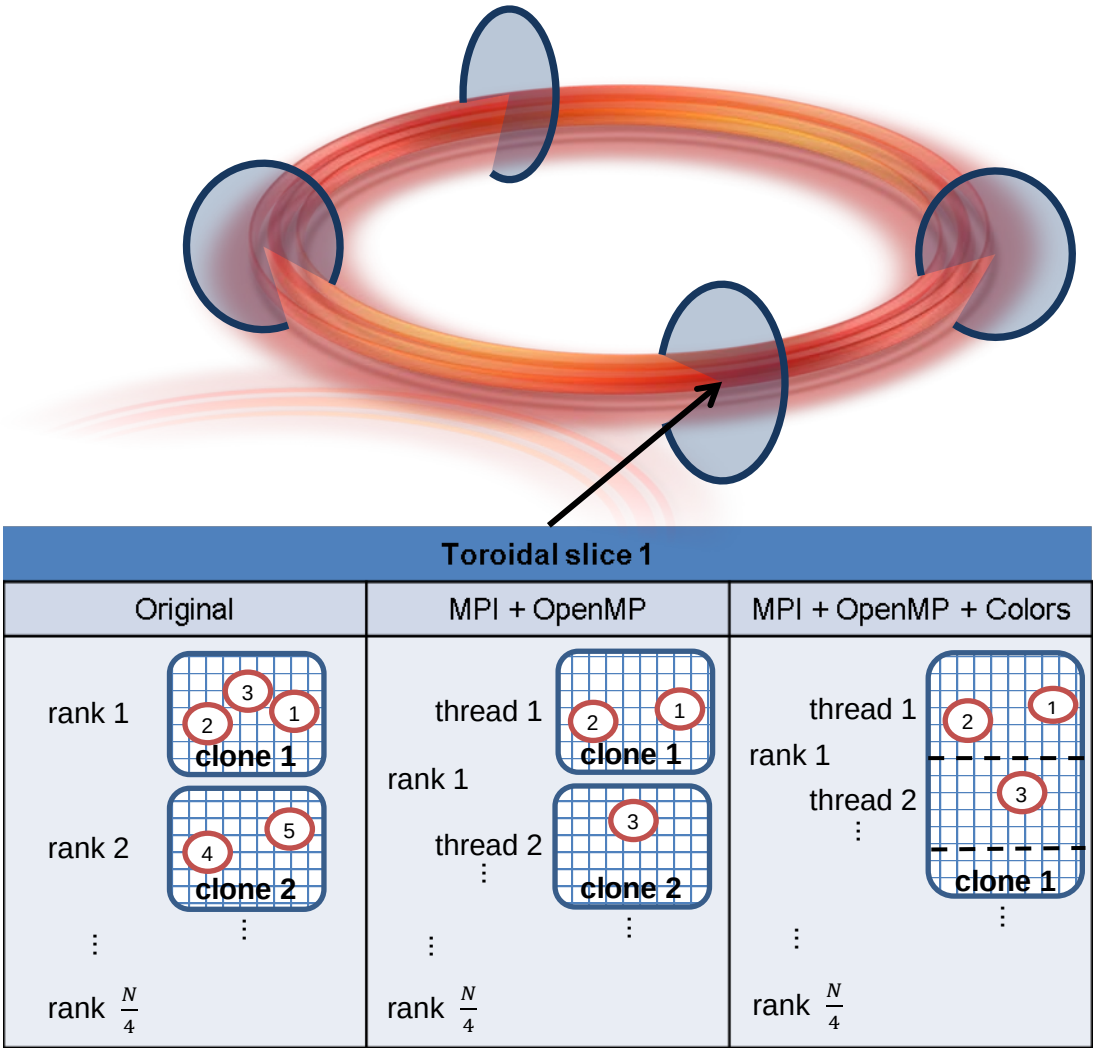

Fig. 86 Comparison of different parallelization configurations. The total number of processes is given by N. The plasma toroid is divided into 4 slices and each slice gets the same amount of processes $\frac{N}{4}$.

7.5.5. Switch to a structure of arrays where possible

It is well known that the usage of an array of structures (AoS) can lead to performance penalties in high performance computing. AoS schemes naturally occur when employing object oriented software design, as objects tend to be realized using structures. The performance penalties of this architecture arise from bad cache usage patterns. As an algorithm traverses a list of objects, it may not need the complete structure of the object but only certain fields of it. This leads to repeated memory jumps from the respective field of one object to the respective field of the next object.

A structure of arrays (SoA) inverts the data layout. All fields of each object are saved in distinct arrays. Each array contains all field values of all objects. If an algorithm now traverses the different objects, using only a specific subset of field values, it will still use contiguous arrays in memory. This leads to optimized cache behavior.

A visualization of the differences between SoA and AoS can be found in Fig. 87.

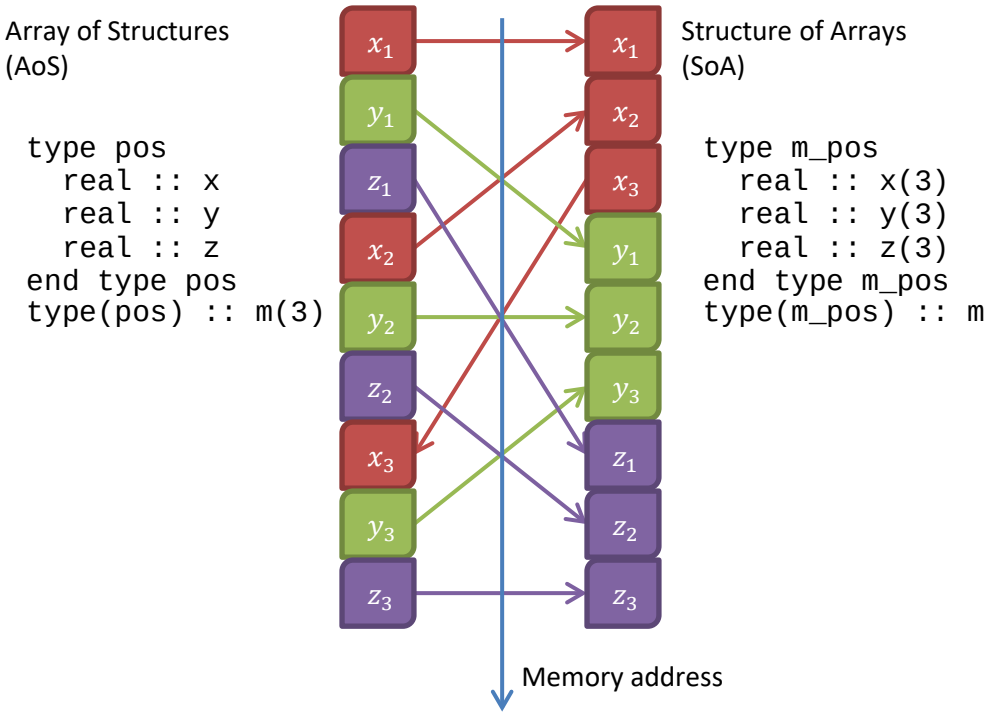

Fig. 87 Difference between the structure of arrays (SoA) and the array of structures (AoS) type of memory layout.

7.5.6. Additional Overhead introduced by the optimizations

Unfortunately, the new structure also brings additional costs with it. The following table shows the increase in memory consumption of the ORB5 code (2-weights scheme, single precision, a minimum of 4 gyro-points per marker).

Standard scheme	Gyro-points	Gyro-points sorted
144 byte per marker	≥ 4 x 60 byte for gyro-points + 144 for the marker → ≥ 384 byte per marker	≥ 4 x 120 byte for gyro-points + 144 for the marker → ≥ 624 byte per marker

The memory demand of the simulation for the markers will more than double when using the new gyro-point structure.

Additional computational overhead stems from the possible sorting and unsorting step, which needs to be compared to the possible speed-up, related to the sorting. Overall, the ORB5 team reports a 30% total speed-up for the optimization efforts, which should be contrasted to the large time investment required for this type of code refactoring. We also propose additional structural changes, which could increase the performance of the code even further.

7.6. *PIConGPU optimizations*

PIConGPU is a fully relativistic 3DV3 PIC code. It is developed at the Helmholtz Zentrum Dresden Rossendorf (HZDR). It offers a suite of different algorithms, post-processing options and configurations. Unfortunately, it does not contain any support for GK PIC.

It is still interesting to analyze the code as the stated goal of its development (and its name) is to provide good performance on modern computer architectures, which contain large vector computing capabilities.

7.6.1. Template Metaprogramming

The main difference of PIConGPU in comparison to other scientific codes is the extensive use of a specific modern programming language technique: template metaprogramming. This technique is available in a few programming languages, most notably in C++, which is used for PIConGPU.

Template metaprogramming is a technique, which provides automatic code generation. It is considerably more advanced than preprocessor code generation, which is one of the only extensively used automatic code generation techniques. Template metaprogramming, in contrast to preprocessor code generation, is Turing-complete and therefore enables the compile-time solution of all algorithms accessible to a Turing machine. It is also a complete and well-defined language feature and not only a very bare-bones search and replace functionality (with some extra functions).

Automatic code generation can be a powerful tool for the generation of auto vectorizable code. One of the most important prerequisites for auto vectorization is simplicity of the code, which is presented to the compiler. With automatic code generation, the number of branches in a code can be reduced significantly. The template metaprogramming part of the compiler can use knowledge from the user parameters in order to get rid of unnecessary branches in the code. Loop limits and counters may also rely on the parameters chosen by the user. Exploiting this extra information, the template metaprogramming part of the compiler can produce intermediate code, which is much simpler than the original code base. This may lead to improved auto vectorization.

A significant drawback of template metaprogramming is the fact that it is an advanced programming technique and requires more training in computer science. Since scientific codes are mainly written by people of other academic professions it may therefore be a fairly large hindrance to code adoption.

7.6.2. Computation – Communication Overlap

With the more structured approach to software engineering this code project exhibits, it can implement certain optimization strategies fairly efficiently and transparently.

By defining temporal or spatial locality for certain data structure it is possible to automatically make use of communication – computation overlap and to optimize cache locality without too much interference by the programmer.

7.6.3. Using PIConGPU for GK PIC

PIConGPU has a modular structure. A core component is PMacc, which mainly consists of MPI parallel data structures (containers) for particles and fields. The algorithms for particle pushing, field advancement, etc. are contained in a module, which builds on PMacc. This module has a GPL license, while PMacc has an LGPL license. A possible usage scenario could therefore be in the exploitation of the PMacc module. This would entail the need to rewrite most of the GK PIC algorithms from scratch.

A step-by-step approach is not feasible. It is also not possible to use parts of the PIConGPU code base as an external library with an appropriate interface connecting the two, as that would certainly be severely limiting for the performance. The structure of the code is built on the assumption that nearly all data is persistent with only small percentages of data interchange between time steps. Global operations are not in the purview of the project coordinators and should be avoided.

7.7. *Summary*

The PICOPT project started with an in-depth familiarization with the underlying theory. Previous experience with the standard particle-in-cell algorithm was leveraged and extended with the necessary details and differences to gyrokinetic particle-in-cell.

After being acquainted with the mathematical background, we continued with researching the current state of auto vectorization. This field is rapidly developing and staying on top of its details will prove invaluable for the envisioned optimizations of the GK PIC algorithm.

Subsequentially, multiple codes were investigated. From the widely used GK PIC code ORB5 we continued to the very modern PIC code PIConGPU.

Investigating the optimizations of ORB5 on the one hand, we conclude that it is a very novel and interesting way of structuring a GK PIC code. Implementing the gyro-points as first order objects enables some powerful optimization schemes, mostly related to the spatially localized nature of the gyro-points. The new data structure is very close to an ordinary PIC data structure and gets rid of the complicated and non-local gyro-ring averaging. It also entails significant drawbacks, which we have detailed in this report. Leveraging the full potential of these changes in terms of the exploitation of vectorization is still not finished and may require substantial additional investments. We made suggestions for additional improvements to the algorithm structure.

PIConGPU on the other hand is a completely different approach to scientific computing than ORB5. This code was written using elaborate software engineering and therefore exhibits many advantages which cannot be introduced into other codes without a complete rewrite of the code base. Unfortunately, it is also impossible to utilize it partially in order to optimize other codes, including ORB5. In summary, using the PIConGPU code base directly for GK PIC involves lots of obstacles and problems and will therefore not be pursued directly.

7.8. *Reference*

[1] Xianglong Kong, M. C. (2011). Particle-in-cell simulations with charge-conserving current deposition. *Journal of Computational Physics*, 1676–1685.

8. Report on HLST project MPI3-DG

8.1. *Introduction*

The high order 3D Discontinuous Galerkin code Fluxo [1,2] has been developed to solve the 3D full MHD equations, including nonlinear and resistive terms. It has an explicit time integration and uses unstructured hexahedral meshes. The code aims to improve the scalability of 3D non-linear MHD simulations of fusion plasmas. Fluxo is fully MPI parallelised and production runs of $\mathcal{O}(10{,}000)$ MPI ranks are possible, exhibiting very good weak and strong scaling properties on current NUMA architectures. This is related to the fact that the Discontinuous Galerkin scheme with explicit time integration has dense local operations, only direct neighbour communication and low memory consumption. However, the current MPI parallelisation has a single global MPI communicator, with each core representing one MPI rank, and hence does not distinguish between cores within a given compute-node and the remaining ones. Such a flat MPI communicator implementation is possibly not the ideal solution in terms of scheduling of the necessary communication and therefore might produce communication overhead that can play a role when large numbers of resources are used. This issue is expected to become even more pressing in future supercomputing hardware based on many-core architectures. The new Intel Xeon Phi (Knights Landing) demonstrates this trend by incorporating a CPU with 68–72 cores. On these architectures, a pure MPI implementation is questionable. One approach to avoid the overhead is to employ a hybrid OpenMP/MPI implementation. However, an attractive alternative exploits shared memory (SHM) capabilities of the new MPI-3 standard directly. Here, a two level communicator structure can be introduced to replace intra-node communication with SHM access without the need to invoke OpenMP threads. The aim of this project is to make a basic assessment of this approach in the framework of Fluxo, and to decide whether or not it can potentially improve the code's scalability. Feasibility, in terms of the required changes to the code design, needs to be considered. To this end, the experience gained with previous HLST projects on the subject of MPI-3 SHM windows (e.g. [3,4]) is of paramount importance, to the extent that it justifies the follow-up studies conducted in the course of the current project, even if not directly involving the Fluxo code.

8.2. *Forcheck analysis of Fluxo*

The first stages of the project were dedicated to getting acquainted with the basics of the Fluxo code. These included learning how to install and compile Fluxo, as well as to run predefined test cases. The following step, agreed upon with the project coordinator, was to perform a static analysis of the source code for FORTRAN standard conformance issues using Forcheck [5]. Since the code uses CMake for automatic code compilation and build generation, the corresponding infrastructure had to be extended to accommodate an option to perform the Forcheck analysis. To this end, two CMake module files were added, namely, `PreprocessorTarget.cmake` and `Forcheck.cmake`. These are invoked from Fluxo's CMake listfile `CMakeLists.txt` when the new CMake option `FORCHECK_ANALYSIS` is enabled. This allows a Forcheck report to be automatically generated for any given compilation configuration. Several main build configurations were analysed, resulting in the detection of a few minor bugs. The most common ones were related to type mismatches in function and subroutine calls. All detected issues were fixed.

8.3. *Profiling of Fluxo*

The subject of this section is the assessment of Fluxo's parallel scalability, regarding especially its communication components. The first steps involved performing strong and weak scaling studies on a Navier-Stokes case with representative problem sizes, spanning from 128 to 16384 finite-elements, on 1 to 16 Skylake (SKL) compute-

nodes of Marconi. Fluxo's built-in performance metric, the so-called *performance index* (PID), was used to check the code's scalability. It measures the time spent per time-step per degree of freedom (DOF).

Together with the project coordinator Florian Hindenlang (FH), the source code was modified to allow running the same Navier-Stokes cases with the MPI communication switched off. Comparing the simulation times and PIDs with and without communication showed, as expected, the negative impact of the communication on the code's performance. Indeed, communication causes an increasing degradation of the scaling efficiency with the number of processes over which the domain is distributed. Moreover, since the amount of computation (floating point operations, FLOP) is independent of whether communication is used or not, the faster run-times yielded without communication further indicated that the overlap of communication and computation implemented in Fluxo is not able to completely hide the communication. This was an expected outcome. Barring idealised cases, it is not typically possible to fully overlap communication with computation in a production code. However, such a simple analysis based on the total run-time of Fluxo cannot provide any further insight on the subject. The available results do not even allow ruling out the possibility of zero communication hiding. A more detailed assessment of the cost distribution inside the code became mandatory.

Initially, the profiling of Fluxo was done using Scalasca [6]. However, it was then decided to implement an explicit instrumentation into Fluxo's source code, which should allow for more flexibility. For our purposes, it was sufficient to focus on the subroutine `DGTimeDerivative`, which computes the residual d***U***/dt from the solution ***U*** at every time step employing the discontinuous Galerkin (DG) method. This is the most computationally intensive part of the code, and it is where the communication hiding is supposed to occur. By introducing differential time measurements (using `MPI_Wtime`) between the relevant subroutines called inside `DGTimeDerivative`, it was possible to obtain a much more detailed picture of where the simulation time is spent.

Fig. 88 shows an example that combines the appropriate time measurements to show the total time spent per iteration on communication and on the computations that are supposed to hide it. The solid yellow curve measures the total communication cost of exchanging the cell faces between neighbouring subdomains. This cost can be split into the communication of the solution ***U*** (solid red) and of the flux ***F*** (solid green). The dashed curves represent the cost of the floating-point operations used to potentially hide the communication with the same colour. We call these *computation buffers*.

Communication hiding can be achieved by using non-blocking communication calls. In Fluxo's case, these are `MPI_ISend` and `MPI_IRecv`. Once called, they should theoretically initiate the communication process and return immediately, releasing the resources to perform other operations on locally available data. The communication supposedly continues in the background, and if this indeed happens, at least to some extent, then part of the communication is being hidden. Then, just before the data being exchanged between resources is needed for further computations, the completion of the corresponding communication process must be ensured. In Fluxo's case, this means calling instances of `MPI_Waitall` to match both the non-blocking send and receive operations started before. The sum of all such communication calls (initiation and completion) yields the solid yellow curve in Fig. 88. As we mentioned before, the communication comprises two independent stages. First, the communication of the solution at the cell faces between neighbouring subdomains is performed (solid red curves). Subsequently, after the former stage is completed and the fluxes have been calculated, their values on the cell faces between neighbouring subdomains are also exchanged (solid green curves). In terms of cost, i.e. message sizes, both communication stages are equivalent. Therefore, if we were to use blocking communication instead, their cost should be very similar. However, in Fig.

88 they are not. This shows that some communication hiding must be taking place, as will become clear shortly.

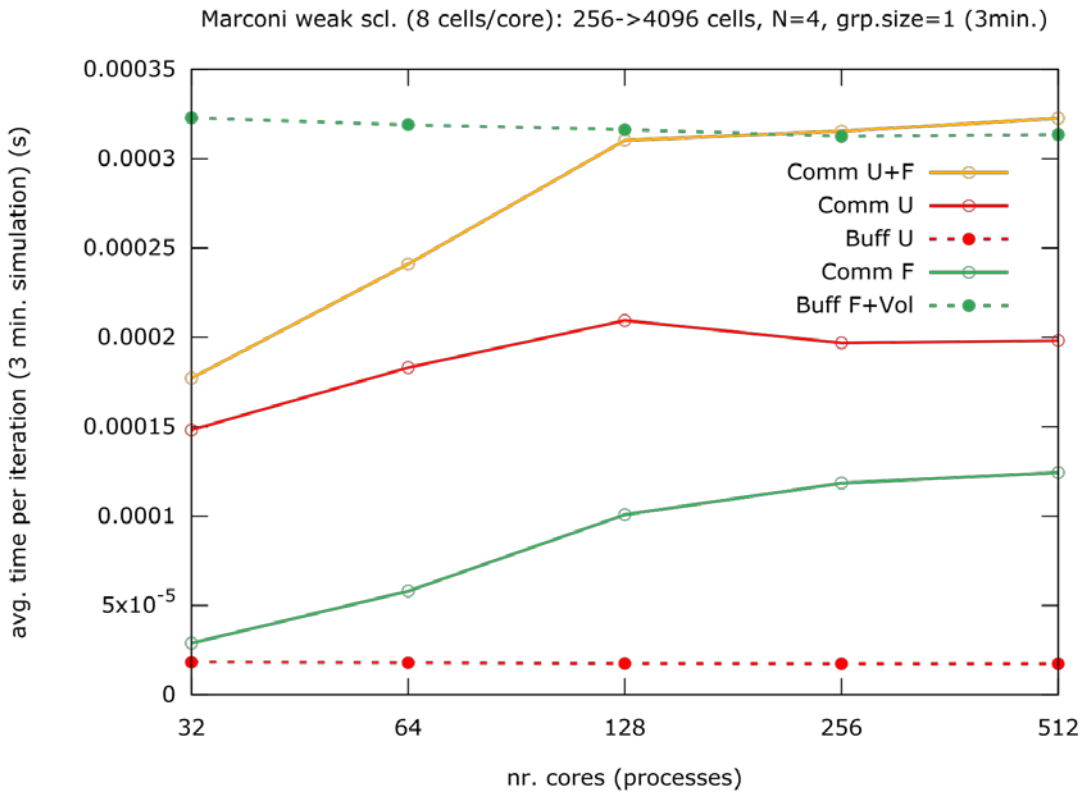

Fig. 88 Weak scaling of fluxo for a Navier-Stokes case with polynomial degree *N*=4 and 8 cells per core. With 32 (out of 48) cores per compute-node used, the smallest domain size ran has 256 cells (single compute-node) and the largest has 4096 cells over 16 compute-nodes (512 cores). Each simulation ran for three minutes of wall-clock time. The solid and dashed curves represent the communication and computation costs, respectively. Comparing the red and green curves shows that part of the communication is being hidden. The results were obtained on the A3 SKL partition of Marconi at CINECA, using the Intel Parallel Studio XE 2018 suite (compiler and MPI library).

As mentioned before, the computation buffers are represented by the dashed curves in Fig. 88. Naturally, they are also composed of two parts, each overlapping one of the two communication stages, respectively, as suggested by their colour coding. Inspecting these curves, one immediately notes an asymmetry in their weight. The dashed red curve values are much lower than the corresponding communication cost they are meant to hide. In other words, this computation buffer is in practice negligible, which yields this communication stage (solution on the cell faces) essentially blocking. Conversely, the dashed green curve has much higher values, so much so that, if the hiding was perfect, the communication cost of the second stage (fluxes on the cell faces) should tend to zero. Assuming that the actual (blocking) cost of both communication instances is similar, as proposed before, comparing the red and the green solid curves tells us that roughly half of the communication is hidden in the latter. Obviously, a way to confirm this unambiguously would be to replace the non-blocking communication directives in Fluxo with their blocking counterparts and repeat the same measurements. Due to time constraints and its lack of practical interest from the perspective of improving the performance of Fluxo, this was not attempted. Instead, we attempted to increase the degree of communication hiding by enabling the Intel MPI environment variable `I_MPI_ASYNC_PROGRESS`, which is disabled by default. Unfortunately, the obtained results were not the expected ones. Instead of improving the communication hiding, it made the whole communication much slower. Mixed results have been found previously [7,8], although these were in the framework of non-blocking collective

communication processes. Fluxo involves instead non-blocking point-to-point communication directives. Further investigations are needed to clarify the issue. For the moment, this is left as a suggestion for future work.

A final note concerns the communication-free parts of the algorithm. Before, we already hinted at the idea that Fluxo's scaling optimisation should focus on the communication part of the code. Now we can see additional data to support that claim. Namely, the weak scaling behaviour of the computation buffers, which are communication-free parts of the algorithm, is close to be perfect, as shown in Fig. 88 (dashed curves). Moreover, although not shown in this report, similar behaviour was observed for all remaining communication-free parts of the algorithm and for all the domain sizes tested.

8.4. *New two-level communication infra-structure*

The basic concept behind the MPI-3 shared memory hybridisation paradigm relies on the ability to differentiate between resources that can directly share data stored in memory (cores inside a compute-node), from those which cannot (cores across different compute-nodes). While the latter require explicit MPI data communication, the former can use instead the so-called shared memory (SHM) windows, which are available since the introduction of the MPI-3 standard. As the name suggests, the data stored therein can be accessed and modified by all resources within a compute-node. In our experience, this new programming concept can potentially bring performance benefits for certain problems [3,4]. However, the implied modifications to a full production code like Fluxo are beyond the scope of the current assessment project. Instead, we decided to investigate one of the components involved in the MPI-3 shared memory hybridisation. Namely, the impact of establishing a hierarchy in the communication, which groups the resources by compute-node. This allows distinguishing between communication inside and across the compute-nodes. In other words, rather than having a single global communicator (`MPI_COMM_WORLD`), two levels of communication are established. One level refers to the communication between the cores inside each compute-node. The other refers to the communication involving cores in different compute-nodes. Note that this is only a logical split of the global communicator, without any node-level data gathering/scattering involved. It simply allows calling separately these two levels of data communication.

Why do we hope to get something out of the two-level communication concept, especially if no actual SHM windows are to be implemented? Well, the main motivation behind it stems from the results obtained when comparing intra-node and inter-node communication bandwidth, even for a pure MPI implementation. This subject is covered in detail in Sec. 8.6. For understanding the material being covered here, it suffices to state that intra-node and inter-node communication performance differs. The former is expected to be faster for the setup at hand, whereby many cores per compute-node are involved in the MPI communication. This assumption alone is sufficient to argue that initiating and completing a communication process involving a mix of messages exchanged inside and across compute-nodes is not optimal. In such a case, all participating tasks will have to wait for the slowest path, which probably involves inter-node communication. Alternatively, one could first trigger the slowest communication, i.e., messages to be exchanged across different compute nodes, and concurrently do some calculations on locally available data. Afterwards, one would trigger the remaining faster intra-node communication, also followed by some remaining computation on data available locally. The communication completion should be made in the opposite order. The call to complete the intra-node level of communication is made first, followed by the call to complete the inter-node communication, possibly with some extra computational task in between. This concept increases the communication hiding possibilities. Obviously, for it to work optimally, the computational processes running in between

the different communication stages, which we have been calling computation buffers, need to be tailored accordingly. So far this has not been done. Only the two-level split in the communication was implemented, with the calls to initiate communication made first for the inter-node messages, and immediately after for the intra-node messages. The communication completion calls (`MPI_Waitall`) are made in the same fashion, but with an inverted order, i.e., first for the intra-node communication and then for the inter-node communication.

The impact on the communication costs is shown in Fig. 89. The two-level communication measurements corresponding to the solid lines. The dotted curves are the same as the solid curves in Fig. 88, that is, they correspond to the original code, with only single-level communication. It is clear that this simple change yields an improvement in the communication performance. This was a bit surprising at first, since nothing was done to increase the overlap between communication and computation. However, if one keeps in mind that each task communicates data with all its neighbours, this results in a non-trivial global data dependency equilibrium between all sub-domains in the simulation. Then, refining the granularity of the initiation and completion steps in the communication processes via the two-level group structure, should in principle change this global equilibrium. Furthermore, if this change is dictated by establishing an ordering for communication initiation and completion based on the hardware bandwidth characteristics of the system, rather than on the original “first come, first served” basis, then it is plausible to assume that it can improve the performance.

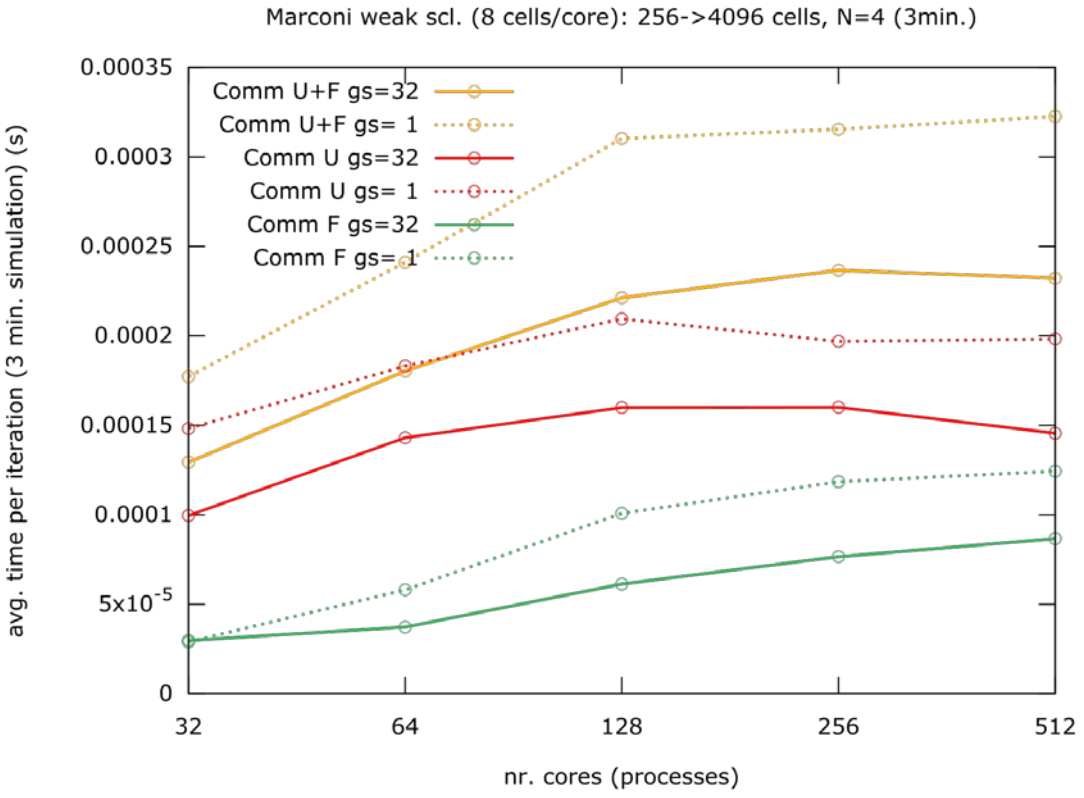

Fig. 89 Similar setup as in Fig. 88. The previous communication cost measurements are shown by the dotted curves. The corresponding values for the new two-level communication implementation, using 32 cores per intra-node communicator, are shown by the solid curves. Note the improvement for both communication stages (solution *U* and fluxes *F* at cell faces, respectively). The overlapping computation buffer costs are similar in both cases, and are therefore not shown.

In order to check the proposed interpretation of the improved two-level communication performance, and eventually understand the communication behaviour inside Fluxo in detail, more steps need to be taken. For starters, temporal

tracing is necessary, which will further allow a closer look into the communication hiding processes. This can be done by using Intel Trace Analyzer and Collector (ITAC) or Vampir. Since the former is available on Marconi, its use is planned. Besides this, several other ideas to modify the current two-level communication performance are available. One is to cycle through the MPI ranks on each communication level using calls to `MPI_Wait()` instead of a single `MPI_WaitAll()`, which is used currently. Another one is to assess if using instead the `MPI_WaitAny()` would be possible. Since we argued before that refining the granularity in the communication might help, one could consider the generalisation of the current model to arbitrary number of levels. Finally, the idea to experiment with so-called *persistent communication* can also be proposed. This avoids the creation and destruction of the `MPI_Request` object that occurs every time the `MPI_ISend/IRecv` and `MPI_WaitAll` functions are called. If many successive non-blocking communication steps take place, as it is the case for Fluxo, then this creation/destruction operation can start to yield a measurable overhead.

8.5. *Previously acquired knowledge on MPI-3 hybridisation (VIRIATO)*

We have already stated that the main goal of the current project is to devise a strategy to improve the scalability of the Fluxo code. The optimisation should specifically focus on the MPI communication components of the code. Furthermore, shared memory MPI-3 hybridisation techniques, available since the release of the MPI-3 standard, were put forward as strong candidates for the job. To this end, using the experience acquired during previous HLST projects on the subject is advantageous. This motivates the material presented in the remaining of this report, which builds on the results obtained during the HLST-VIRIATO2 project (2017) [4]. Even though these additional studies do not concern Fluxo directly, they provide insight that is very useful for devising its the optimisation strategy.

8.5.1. Previous results

The discussion provided in this section stems from the HLST-VIRIATO2 project (2017), during which a new parallel hybridised algorithm to calculate bi-dimensional fast Fourier transforms (2D FFT) has been developed. Details can be found in [4]. Here it suffices to know that the applied hybridisation idea relied upon using MPI-3 shared memory (SHM) windows as larger communication buffers accessible directly by several cores inside a compute-node. This makes (i) the algorithm less susceptible to network latency because the messages are larger, and (ii) reduces the complexity of the implied MPI all-to-all communication, which occurs only across the SHM windows, and not inside them. Although the project was successful in conceptual terms, there were questions regarding the communication bandwidth that remained open by the time the project ended. Since these questions are relevant to the current project, the corresponding points shall be revisited here.

Fig. 90 shows the strong scaling of the hybrid 2D FFT algorithm developed for the VIRIATO code presented in [4]. The solid blue curve corresponds to the original algorithm, which uses a pure MPI communication-programming model. It shows the breakup of the scaling behaviour that motivated the HLST-VIRIATO2 project. The remaining dot-dashed curves correspond to the results obtained using the new MPI-3 hybridised algorithm, developed during the HLST-VIRIATO2 project, with different sizes of SHM windows. In general, the size of each SHM window is established by the number of participating resources. Specifically, for this algorithm, the size grows with the square of the number of cores that share it, as shown in Table 5. Obviously, the more cores there are per SHM window, the fewer SHM windows fit inside each node, as also shown in Table 5. The cases shown in red, yellow, green and dark-green refer to 4, 8, 16 or 32 cores per window, respectively, which in turn correspond to eight, four, two and one SHM windows per compute-node, respectively. All the dot-dashed curves decrease monotonically, which, even though deviating somewhat

from the ideal theoretical linear scaling, is still considerably better than the original scaling. However, the best results of the group in absolute terms were yielded with the smallest SHM windows tested, namely, with four cores per window, for which there are eight SHM windows per compute-node (red dot-dashed curve). Apparently, as the number of cores per SHM window increases, an increasing offset on the curves arises. This came as puzzling result at the time. Looking in more detail into the measurements yielded by the performance counters implemented in the source code revealed that the increase in cost was solely due to the explicit MPI communication part of the algorithm [4]. The corresponding values are shown in the last row of Table 5. There, the average time spent on MPI communication during the execution of a single pair of forward and backward Fourier transforms distributed over 256 cores is shown.

nr. cores/SHM window	1 core	2 cores	4 cores	8 cores	16 cores	32 cores
nr. SHM windows/node	32	16	8	4	2	1
SHM window size (MiB)	0.125	0.5	2	8	32	128
elapsed time (on 256 cores)	20.1 s	17.8 s	20.4 s	52.4 s	101.4 s	174.7 s

Table 5 Average time spent on explicit MPI communication for different numbers of cores per SHM window on 100 pairs of forward and backward bi-dimensional Fourier transforms performed on the same domain used in Fig. 90 when distributed over 256 cores.

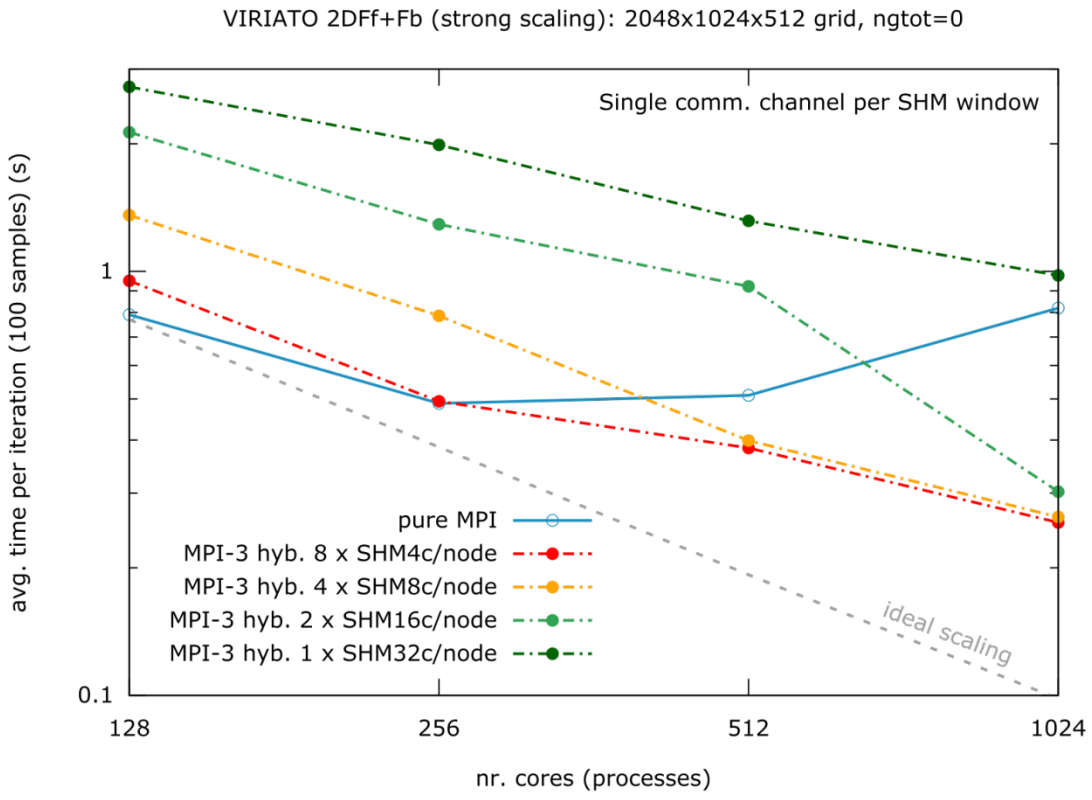

Fig. 90 Strong scaling of the parallel bi-dimensional FFT algorithm on a three-dimensional test domain, corresponding to, in VIRIATO terms, a configuration space grid-count of 1024x2048x512 in (x,y,z) with no velocity moments. A sequence of bi-dimensional FFTs both in forward and backward directions is executed 100 times and the average elapsed time is considered. VIRIATO's original algorithm is shown in solid blue. The new algorithm using shared memory windows is shown in dot-dashed colours, each corresponding to a number of cores $\in\{4,8,16,32\}$ per SHM window, respectively. The 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).

For up to four cores per window, the communication cost remains roughly constant. However, beyond that value it increases significantly. By algorithmic design, the

number of cores used per SHM window dictates how many SHM windows are allocated in total, and consequently how many cores are globally involved in the MPI communication. In our case, because the main idea was to minimise the number of messages involved in the 2D FFT algorithm (communication complexity), a single task per SHM window participates in the transpose all-to-all communication. So, the bigger the SHM window (more cores per window), the fewer communication channels are used. And, according to our results on Marconi's Skylake partition (Fig. 90 and Table 5), having a small number of MPI communication channels seems to negatively impact the communication performance. Therefore, we can infer that increasing this number should improve the performance. To test this hypothesis, the hybrid parallel 2D FFT algorithm has been extended accordingly. It now allows more than a single communication channel per SHM window, in other words, more communicating cores per SHM window. Note that, although this goes against the original motivation to decrease the communication complexity of the algorithm to a minimum, the corresponding increase in the number of exchanged messages is still kept relatively low. It increases linearly with the number of cores involved in the communication, and not quadratically, as is the case for a general all-to-all exchange pattern. Also noteworthy is that this extension has been already proposed towards the end of the HLST-VIRIATO2 project [4].

8.5.2. New results

Fig. 91 shows the same strong scaling test as Fig. 90, except that four communication channels per SHM window are used, instead of just one. This means that four cores per SHM window participate in the communication, each concurrently exchanging a specific part (one fourth) of the data stored therein. Therefore, there are four times more exchanged messages, whose size is four times smaller, compared to the previous situation, where a single core per SHM window handled the whole data communication. This constitutes the linear increase in communication complexity that we mentioned in the previous section.

The results clearly show an advantage in terms of the achieved communication bandwidth, in accordance with the previously proposed hypothesis. However, there is still not enough information to provide a complete answer to the question of how many cores should be used per SHM window, nor to how many of these should be involved in communication, in order to reach the maximum communication bandwidth for this algorithm. To do so requires dedicated measurements of intra- and inter-node communication bandwidth. A detailed discussion of the corresponding results is presented in the remaining sections. However, to be able to close the topics involving the previous HLST-VIRIATO2 project, we will simply draw the conclusions that have been reached regarding the algorithm at hand.

The time offset proportional to the number of resources per SHM window observed in Fig. 90 cannot be explained by the network bandwidth across compute-nodes, as initially suggested [4]. Under normal working conditions at least, the maximum communication bandwidth achievable between two compute-nodes is independent of the number of cores involved in the communication. The culprit is the dependency of the intra-node memory bandwidth on the number of cores used to access the memory inside the compute-node. On a typical NUMA architecture, like the Intel Skylake family of processors used on Marconi, roughly half of the cores available within a compute-node, distributed over the available sockets (two, in Marconi's case), are needed to reach the full memory bandwidth. Using a number of communicating cores inside a compute-node below this figure necessarily under exploits the available communication bandwidth. In our case, this affects the communication between SHM windows inside the compute-nodes and explains why having smaller numbers of bigger SHM windows inside each compute-node led to poorer results: a small number of cores per compute-node involved in the intra-node communication can only reach a fraction of the available memory bandwidth. However, this alone does not explain the special case of having a single SHM window per compute-node (dot-dashed dark-green curve in Fig. 90), for which by

definition there can be no data exchange between different windows inside any given compute-node. While this is true, it is important to remember that the current implementation of the algorithm treats the diagonal terms of the matrix transposition similarly to the off-diagonal ones. This means that they also involve explicit MPI intra-node communication, not between MPI ranks in different SHM windows, but rather in the form of a given MPI rank "exchanging" data with itself inside the same SHM window. Moreover, as the SHM windows grow (more cores per window), the role of the matrix diagonal as a bottleneck becomes more important simply because the ratio between diagonal and off-diagonal data also grows. To understand this in more detail, the reader is referred to the final HLST-VIRIATO2 project [4]. In particular, Fig. 1 therein should provide an intuitive graphical interpretation for this fact.

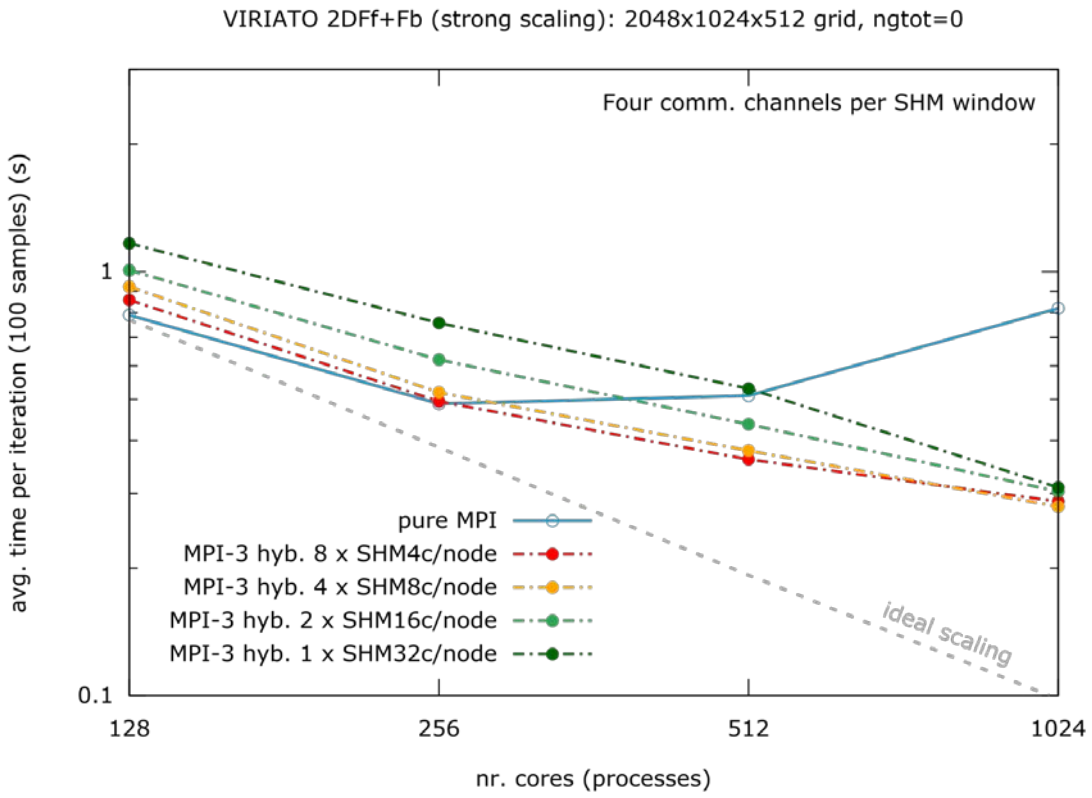

Fig. 91 Same as in Fig. 90 but, instead of using a single communication channel per SHM window, now four communication channels per SHM window are concurrently used to exchange the corresponding data.

The already implemented extension of the algorithm employs more cores to concurrently communicate the data stored in the SHM windows. Because of the NUMA architecture of the compute-nodes, this yields a higher memory bandwidth per SHM window. Since the data stored therein needs to be accessed in order to be exchanged via MPI communication, also a higher communication bandwidth is yielded. This is the reason why using more communication channels per SHM window yielded better results, as seen in Fig. 91. Still remaining is to change the way the diagonal transpose terms are handled, which is clearly an algorithmic weakness of the current implementation. Instead of using self MPI communication, a direct access to the SHM window data by all the participating cores should provide an optimal intra-node memory bandwidth access. This is left as a recommendation for future work, possibly within the framework of a small HLST project dedicated to the subject.

8.6. *Communication bandwidth*

The results reported in the previous section raise general questions on how to attain maximum communication bandwidth between SHM windows. To answer them in a

precise manner and at the same time support the conclusions already drawn regarding the hybrid 2D FFT algorithm, empirical direct measurements of the communication bandwidth had to be made. The simplest test that can be devised uses point-to-point MPI directives to exchange a given amount of data between a given set of resources. Distributing these resources either over a single compute-node or over two distinct compute-nodes allows a detailed characterisation of the intra- and inter-node communication bandwidth, respectively. This is the topic of the remaining sub-sections.

8.6.1. Intra-node

Let us start by measuring the intra-node communication bandwidth using a ping test, as schematised in the top left diagram of Fig. 92. This is the simplest possible configuration, corresponding to a single MPI communication channel within an MPI communicator using two tasks within the same compute-node. A message is then sent by one of the MPI tasks (rank zero) and received by the other (rank one), for a uni-directional data exchange. This situation can be generalised, in order to increase the number of MPI communication channels employed, by increasing the number of tasks in the communicator, still within the same compute-node, and splitting the message exchange between them. The exchange is still uni-directional, between the group of the sending tasks and the group of the receiving tasks. The example given in the bottom left diagram of Fig. 92 corresponds to decomposing the data exchange over four messages, sent concurrently from the ranks 0, 1, 2 and 3 to the ranks 4, 5, 6 and 7. This technique mimics the changes which were introduced with the 2D FFT algorithm extension presented in Subsec. 8.5.2 and allows inspecting the influence of the amount of resources on the communication bandwidth.

Fig. 92 Schematics of the intra-node communication bandwidth ping test. The top row shows the cases, which use a single MPI communication pair and the bottom, row the cases which use four concurrent MPI communication pairs. The left column corresponds to the pure MPI communication configurations, with standard local arrays as communication buffers. The right column corresponds to the MPI-3 SHM hybrid configurations, with SHM windows as communication buffers.

The diagrams on the right column of Fig. 92 represent the generalisation of the ping test to the MPI-3 SHM hybrid case. The communication buffers, which are standard Fortran allocatable arrays in the previously explained pure MPI communication case, are replaced with pointers to SHM windows in the hybrid case. Otherwise, the source code remains unchanged. The minimal SHM hybrid configuration that can be devised corresponds to the special case of having a single core per SHM window. Even though no practical shared access to the memory is made by any two different cores, the remaining hybrid infrastructure is still used, namely, the allocation of a SHM window per core and its access made via a pointer. However, the fact that the results obtained with this hybrid configuration show no difference whatsoever to the respective pure MPI results, seems to indicate that the MPI communication is handled indistinguishably on both cases. This is especially relevant for inter-node communication, where differences in the bandwidth measurements can occur, as we shall see in Subsec. 8.6.2. Therefore, in what follows, we shall use, instead, the minimal hybrid configuration, which shares memory over different resources, namely, the one which allocates SHM windows over two cores, which is the case shown in Fig. 92.

Even though there are two cores per SHM window, only one of them participates actively in the ping test. The remaining core does nothing other than making the code MPI-3 SHM hybrid by definition. Using the top right diagram to illustrate this, one sees that ranks zero and two execute the message exchange, while ranks one and three do not participate actively in the ping test, even though they have access to each of the corresponding SHM windows. Similarly, to the pure MPI counterpart, the bottom right diagram illustrates a four-fold increase in the MPI communication channels, this time by splitting the data exchange into four pairs of communicating SHM windows.

Fig. 93 shows the results of executing the ping tests described above using one, two and four communication channels (blue, red and yellow, respectively). For each message size, the exchange is repeated 100 times, separated by MPI barriers (after every single exchange), and the results are averaged in order to increase their statistical significance. The message sizes were varied between 64 B and 1024 MiB, which allows to identify different communication regimes responsible for the non-monotonicity of the curves. For small sizes, the bandwidth is limited by communication latency, defined as the finite cost of exchanging a zero-sized message. The middle region, in which the maximum bandwidths are yielded, corresponds to message sizes for which cache memory can play a role. Since 100 consecutive samples of the message exchange are performed, cache re-usage is possible, provided the data to be exchanged fits into the cache. Once the data has been loaded from main memory into the cache, which occurs during the first time MPI communication call, it can be repeatedly re-accessed from the cache during the subsequent communication calls. Indeed, it was verified that when a single exchange is performed, the cache effects are much less pronounced. Finally, the last communication regime evidenced in the plot occurs roughly beyond 33 MiB (L3 cache size), where the communication bandwidth saturation is reached.

The first important observation from Fig. 93 is that the pure MPI communication (solid curves) and the MPI-3 SHM hybrid (dotted curves) bandwidth measurements show no significant differences. This is relevant because it means that the MPI-3 SHM hybridisation infrastructure does not degrade the intra-node communication performance. Moreover, assuming that this is a general property, it should apply to any MPI-3 SHM hybridised code, and therefore to the 2D FFT algorithm of Sec. 8.5 also.

The second important observation is that for all message sizes, the yielded maximum communication bandwidth increases with the number of resources involved. This can be seen by comparing the curves with different colours in Fig. 93. Furthermore, this is precisely what we referred to in Subsec. 8.5.2 to conclude that it was the intra-node communication that caused the bandwidth degradation for larger SHM windows

(recall Table 5), since larger SHM windows used less communication channels inside the compute-nodes.

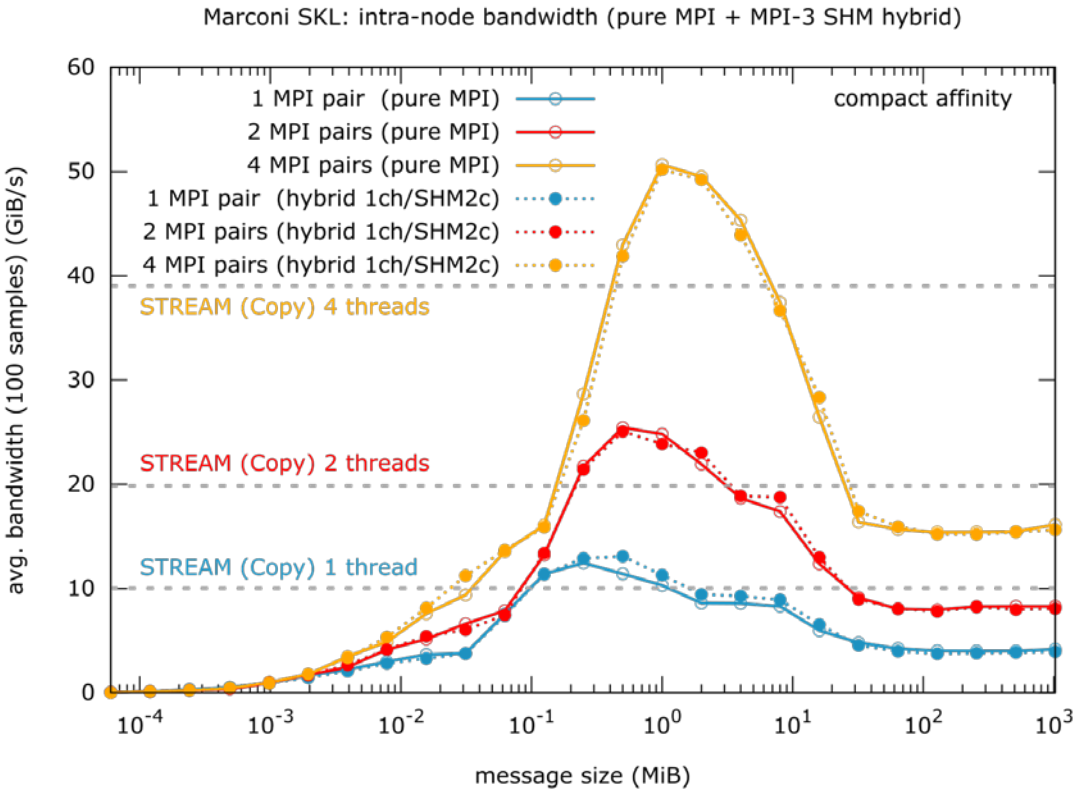

Fig. 93 Intra-node communication bandwidth on the Intel Skylake partition of Marconi, measured via a ping test using different numbers of MPI communication pairs inside a single compute-node, as schematised in Fig. 92. A compact affinity for the MPI ranks was used. The use of MPI-3 SHM windows instead of standard local arrays for the communication does not affect the results. The horizontal dashed lines show the STREAM [9] main memory bandwidth measurements using one, two and four OpenMP threads. Intel Parallel Studio XE 2018 compiler and MPI library were used.

The correlation between the communication bandwidth and the memory bandwidth inside a compute-node is clear. Since the data is stored in memory, it must be read/written from/to memory prior/after the communication. Therefore, the increase in communication bandwidth with the number of involved resources is the expected behaviour inside a NUMA compute-node, whose maximum memory bandwidth is a monotonic function of the amount of resources involved. The dashed horizontal lines in Fig. 93 illustrate this for the case of the main memory (RAM) bandwidth, which is used when the accessed data accessed is too big to fit in cache. They show the figures yielded with one, two and four OpenMP threads, measured via the STREAM benchmark [9]. These lines are below the communication bandwidth peaks yielded by the ping tests using the same number of communication channels, which is a strong hint that those peaks are due to cache effects, as previously suggested. On the other hand, as the message sizes increase, the communication bandwidth decreases to levels significantly below the corresponding main memory bandwidths. This interesting finding requires a more detailed discussion.

There are two different mechanisms available to perform intra-node MPI communication on Marconi's Skylake partition. They are denominated *Cross Memory Attach* (CMA) and *Shared Memory* (SM) [10]. The Intel MPI 2018 library allows to choose between them via the environmental variable `I_MPI_SHM_LMT`, which can be set to `direct` or `shm`, respectively. The latter, as the name suggests, is a double-copy mechanism, whereby the data is stored in an intermediate shared buffer that is used by all local processes to exchange messages [10]. Note that the naming

convention is unfortunate since this mechanism is unrelated to the MPI-3 SHM windows discussed in the remainder of this report. The CMA mechanism is more modern, being available since Linux kernel 3.2. It uses the so-called kernel assistance, which yields a single-copy mechanism via a kernel system call [11]. Kernel calls are costly though. They cause an overhead, which can be higher than the additional copy mechanism of the SM mechanism for very small messages. However, for large enough messages this approach should become advantageous, which justifies why CMA is the default intra-node communication mechanism used by the Intel MPI 2018 library available on Marconi. As such, it is the mechanism used in the measurements shown in Fig. 93.

Comparison tests confirmed that, for message sizes between a few kiB and few MiB, the CMA mechanism yields better communication bandwidths than the SM counterpart in our ping test. However, for even larger messages (beyond 33 MiB) the situation reversed, with a 30% higher in bandwidth yielded when switching to the SM protocol via `I_MPI_SHM_LMT=shm`. In this case, the saturated communication bandwidth was about half of the corresponding STREAM memory bandwidth, which is consistent with the double-copy mechanism of the SM protocol. Conversely, the CMA communication bandwidth for the same message sizes was even lower, which cannot be easily explained in light of the single-copy mechanism involved. After contacting Intel about this result, we learned that CMA does not currently have an ideal implementation and it might suffer from adverse side effects [11].

8.6.2. **Inter-node**

The same ping test code, used before to measure the intra-node communication bandwidth, can now be to measure the inter-node counterpart. However, the uni-directional message exchange between a group of sending tasks and a group of receiving tasks must be executed with an important difference. Namely, those two groups of tasks are no longer placed inside a single compute-node, but rather on two distinct compute-nodes. This way the communication is guaranteed to occur only between cores placed on different compute-nodes, as schematised in Fig. 94. As before, the diagrams in the left column illustrate the pure MPI communication cases and the ones in the right column correspond to the MPI-3 SHM hybrid cases. The remaining description of Fig. 92 provided in Subsec. 8.6.1 still applies here.

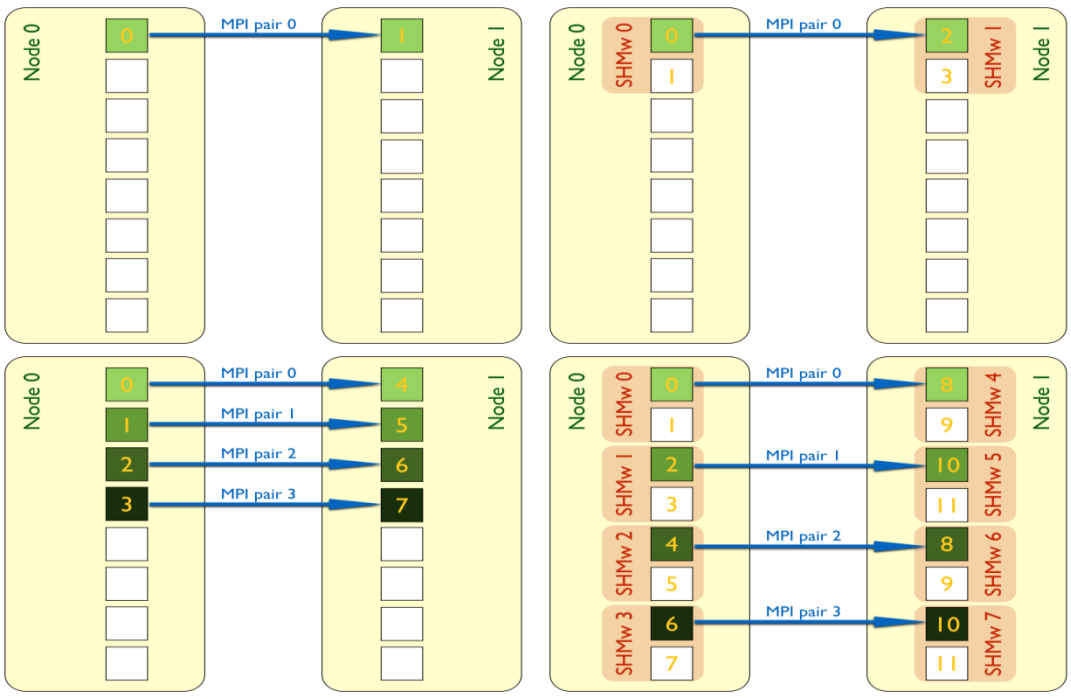

Fig. 94 Schematics of the inter-node communication bandwidth ping test. Similar to Fig. 92, except that the sending and receiving tasks are now placed on two different compute-nodes.

Fig. 95 shows the inter-node counterpart of the results displayed in Fig. 93. As before, one, two and four communication channels are shown in blue, red and yellow, respectively, and the message size is varied between 64 B to 1024 MiB. However, unlike before, there are now evident differences between the pure MPI communication and MPI-3 SHM hybrid measurements.

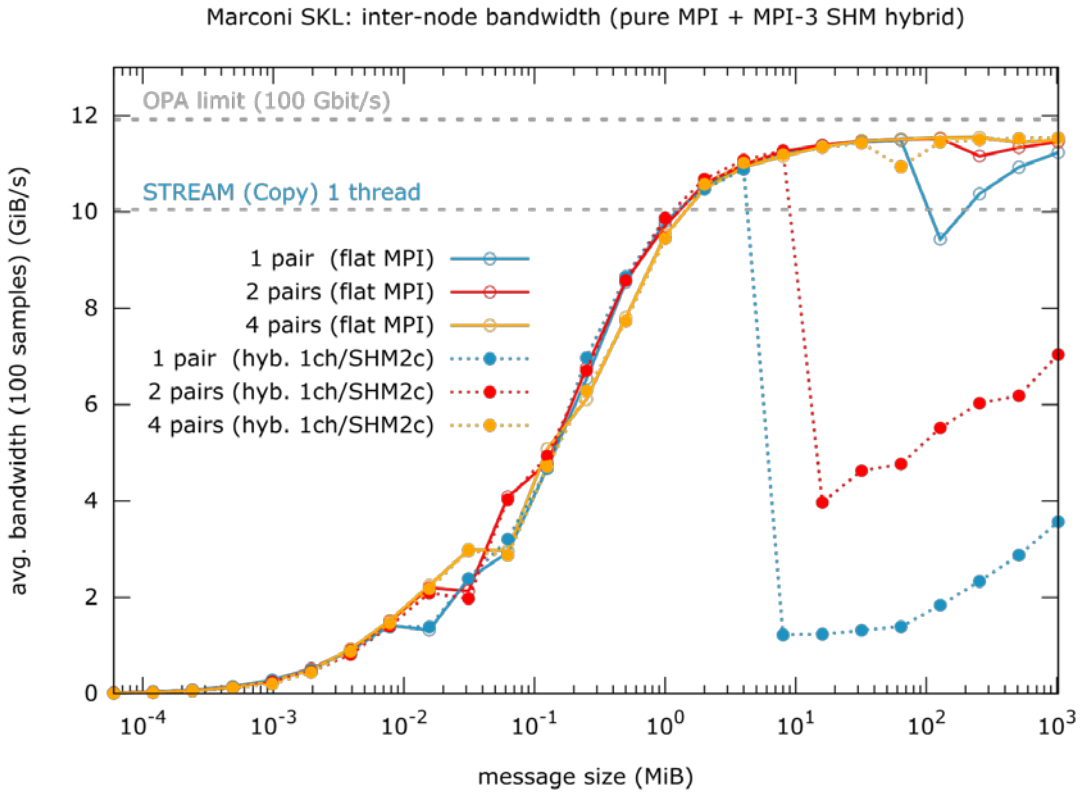

Fig. 95 Inter-node communication bandwidth on the Intel Skylake partition of Marconi, measured via a ping test using different numbers of MPI communication pairs between two compute-nodes, as schematised in Fig. 94. A compact affinity for the MPI ranks was used within each compute-node. The use of MPI-3 SHM windows instead of standard local arrays for the communication affects the results. The horizontal dashed lines show the STREAM [9] main memory bandwidth measurements using one OpenMP thread and the theoretical Omni-path switch simplex bandwidth. Intel Parallel Studio XE 2018 compiler and MPI library were used.

Let us start with the pure MPI communication configuration, which corresponds to the solid curves. Similarly, to the intra-node bandwidth, the inter-node communication bandwidth is latency dominated for very small message sizes. This is mostly caused by the Intel Omni-path switched network fabric that connects the compute-nodes. However, in contrast with the intra-node case, a monotonic increasing function is yielded as the message size becomes larger, independently of the amount of resources (communication channels) used. Note that the previous statement neglects the small dip observed in the blue curve around 100 MiB that, even though statistically relevant, is not very significant in absolute terms. For big enough messages (≥50 MiB) the saturation in the communication bandwidth is reached, with a figure quite close to the theoretical Intel Omni-Path bandwidth. This is shown by the upper dashed horizontal line, corresponding to the simplex bandwidth figure since the ping test uses unidirectional communication. Note that it is only be possible to reach the Omni-path duplex bandwidth (double in value) if bi-directional communication is used (for example, using a ping-pong test). Another noteworthy fact is that the measured maximum communication bandwidth is above the single thread main memory (RAM) bandwidth measured with the STREAM benchmark (lower dashed horizontal line), even when a single communication pair is employed (blue curve). This finding, which might seem puzzling in light of the discussion in

Subsec. 8.6.1 on the relation between NUMA memory bandwidth and intra-node communication bandwidth, is actually easy to understand. It comes about because the inter-node switched network fabric is able to perform remote direct memory access (RDMA), which involves no CPUs, caches or context switches. This is also the reason why the curves in Fig. 95 are independent of the number of cores/communication channels used.

Regarding the hybrid ping tests with SHM windows (dotted curves), one would in principle expect similar results to the pure MPI code, as was the case for the intra-node measurements reported in in Subsec. 8.6.1. However, this seems to be only true up to a given message size, when a small number of communication channels is used. The dotted blue curve in Fig. 95 corresponds to a situation in which a single communication channel between the compute-nodes is employed. It shows a significant bandwidth drop for message sizes around a few MiB. Using two communication channels between compute-nodes slightly ameliorates the issue, but it is only when at least four channels are used that the pure MPI communication result is recovered. It is worth mentioning here that these differences, between pure MPI and MPI-3 hybrid bandwidth measurements, are not seen for the special hybrid configuration using a single core per SHM window. This is the reason invoked in Subsec. 8.6.1 to justify using instead a hybrid configuration with at least two cores per SHM window.

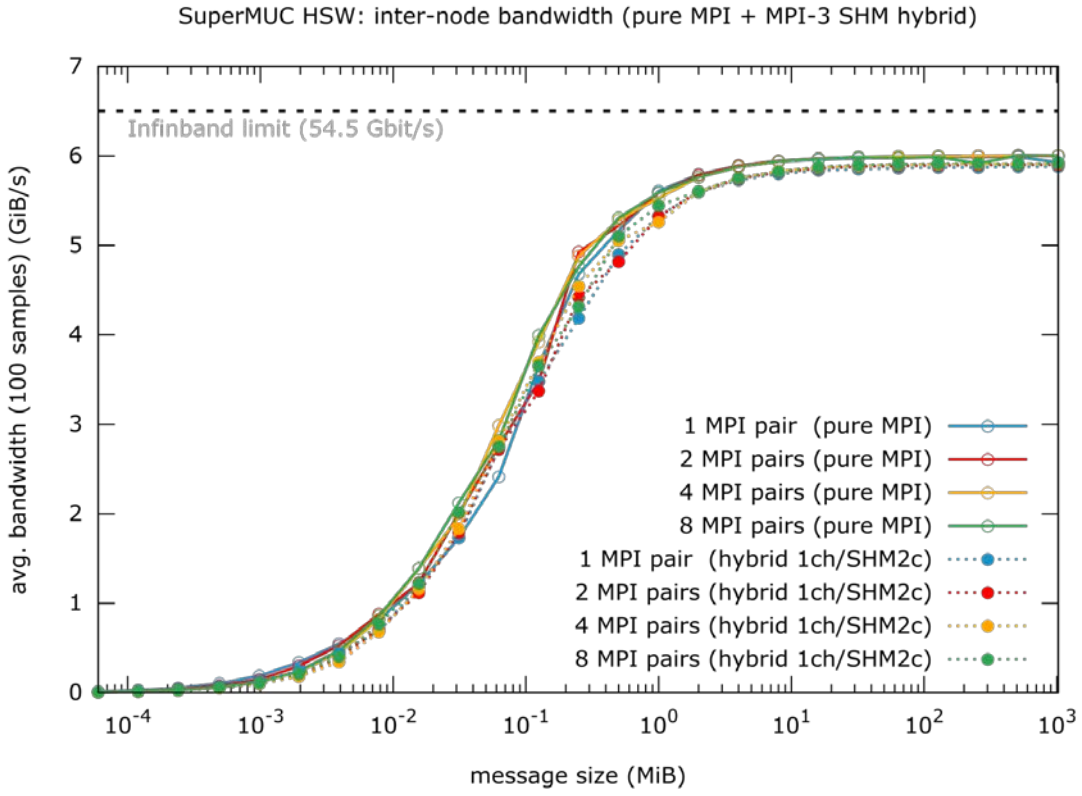

Fig. 96 Same as Fig. 95 but measured on the Intel Haswell partition of SuperMUC, at the LRZ computer centre in Garching. The use of MPI-3 SHM windows instead of standard local arrays for the communication does not affect the results. Intel compiler 16.0 and MPI library 5.1 were used.

The main difference between the pure MPI communication and the MPI-3 hybrid codes is what is passed as the communication buffer to the MPI send and receive calls. Therefore, an obvious refinement in our tests consists in replacing the pointer to the SHM window, used as the MPI-3 hybrid communication buffer, with a standard dummy array, used as the pure MPI communication buffer. Doing so allowed us to conclude that the bandwidth breakdown in the MPI-3 hybrid code occurs at the receiving side, when a SHM window is used. In other words, sending the data stored

in a SHM window and receiving it in a regular array recovers the pure MPI communication bandwidth results. However, if the data is received in a SHM window, then the bandwidth drop occurs. These results are especially puzzling in light of the RDMA capabilities of the switched network fabric, and as such seem to indicate a bug, most probably related to the network. To test this hypothesis the same ping tests were repeated on other machines. On Cobra, at the MPCDF computer center in Garching, which also uses Intel Skylake processors and an Intel Omni-path switched network, similar results were obtained. On the other hand, on the Intel Haswell partitions of both Draco, at the MPCDF, and SuperMUC, at the LRZ computer centre in Garching, no communication bandwidth drop was observed. The SuperMUC results are shown in Fig. 96, where no difference between pure MPI and MPI-3 hybrid is visible. Since both Draco and SuperMUC use Interconnect FDR14 switched networks, instead of the new Intel Omni-path used on both Marconi and Cobra, this seems to indicate that there is an issue with the latter when large SHM windows are used.

It is noteworthy that the tests were also made on the Knights Landing (KNL) partition A2 of Marconi, which also uses the Omni-path fabric. There, the communication bandwidth issues were found to be even more pronounced. First, when MPI-3 SHM windows are used, the drops are even deeper, corresponding to up to one order of magnitude slower communication, as the dotted curves in Fig. 97 show. Second, for large enough messages, contrary to what was seen on the Skylake partition A3, even the pure MPI ping test now displays significant communication bandwidth drops, as seen in the solid curves in Fig. 97. These results were obtained using KNL compute-nodes set in the multi-channel dynamic random access memory (MCDRAM) cache mode. However, no significant differences were seen when using MCDRAM flat mode KNL compute-nodes.

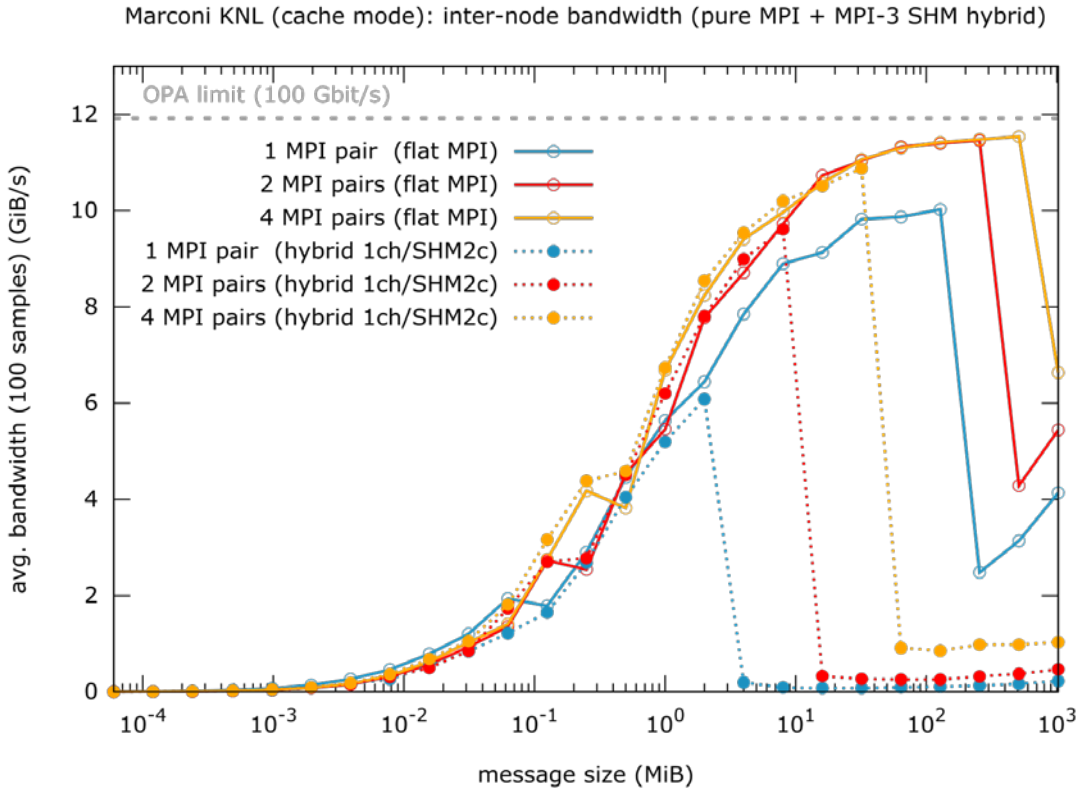

Fig. 97 Same as Fig. 95 but measured on the Intel Knights Landing partition of Marconi using MCDRAM cache mode. For large enough messages, both pure and MPI-3 hybrid ping tests display large communication bandwidth drops. They are more pronounced for the latter though. Intel Parallel Studio XE 2018 compiler and MPI library were used.

All this information has been passed to the Marconi Support Team, who at our request, opened a support ticket directly with Intel. The ticket is still open at the time of writing of this report, although meanwhile the issue seems to have been acknowledged by Intel, and a fix should be provided in the upcoming Intel Parallel Studio XE 2019 suite.

8.6.3. Inter-node, late 2018 update

To overcome the bandwidth drop observed in Intel Omni-path interconnected networks when using large messages stored in SHM windows, Intel has provided a fix. This involved introducing a new environment variable `I_MPI_SHM_FILE_PREFIX_2M` to allow using *huge pages* (HP). As the name suggests, using HP helps the virtual memory management of processes that require large amounts of memory. By increasing the size of the RAM pages from 4 KiB (default) to 2 MiB, the map between a process virtual address space and the corresponding memory address space has much less entries (pages), and is therefore simpler. This fix was first included in the candidate release *Intel(R) MPI Library for Linux* OS Version 2019 Update 1 (Engineering) Build 20180920* and then deployed to the main release *Intel(R) MPI Library for Linux* OS, Version 2019 Update 1 Build 20181016*. They are both currently available on Marconi. Fig. 98 compares this two MPI library releases to the previous 2018 release (*Intel(R) MPI Library for Linux* OS, Version 2018 Update 4 Build 20180823*). As we did before, the reference obtained from the pure MPI ping test (no SHM windows) is also included. Here these measurements correspond to the Intel 2018 release, but the ones made with the 2019 counterparts showed no differences.

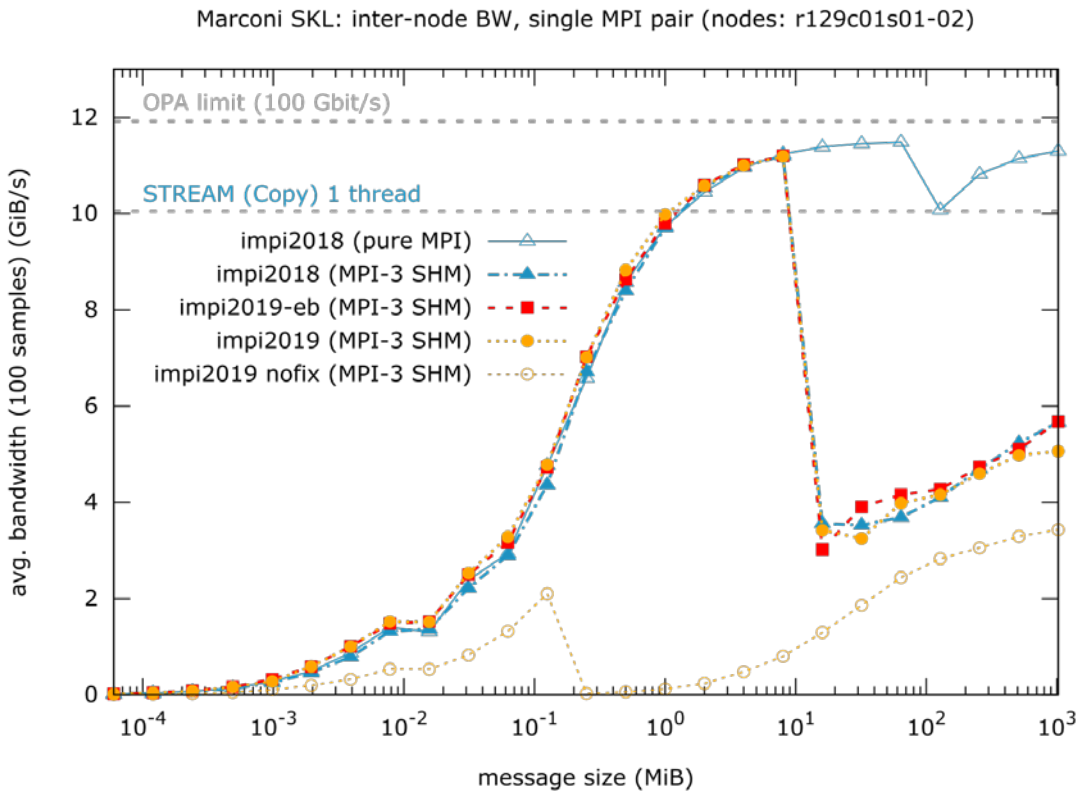

Fig. 98 Inter-node communication bandwidth on the Intel Skylake partition (A3) of Marconi, measured via a ping test using a single MPI communication pair between two compute-nodes, as schematised in top row of Fig. 94. A compact affinity for the MPI ranks was used within each compute-node. Both Intel Parallel Studio XE 2018, 2019 (Engineering Build) & 2019 compiler and MPI libraries were used. The horizontal dashed lines show the STREAM [9] main memory bandwidth measurements using one OpenMP thread and the theoretical Omni-path switch simplex bandwidth.

Since the bandwidth drop problem was most prominent in the case using a single communication pair between two SHM windows, each placed on a different compute-node (recall top right diagram in Fig. 94, this is the only case studied here. Looking at the MPI-3 SHM windows results (solid symbols), one can conclude that they basically reproduce the ones shown in Fig. 95. At most, there is slight increase in performance, in the sense that the drop is not as pronounced as before. This is most probably due to the different default choices used for `I_MPI_FABRICS`, which was set to `shm:tmi` in the measurements of Fig. 95 and to `shm:ofi` in Fig. 98. Furthermore, comparing the results between the three different Intel MPI releases, no improvement is seen when using the more recent releases. This is indeed the expected result because the HP is not enabled on the standard production compute-nodes of Marconi. For this reason, using the `I_MPI_SHM_FILE_PREFIX_2M` variable makes no difference. Before we dive deeper in this topic, it is noteworthy that, after the last Marconi maintenance (December 3rd–4th, 2018), an issue arose with the default *libfabric* bundled with the Intel 2019 releases. This caused a very low communication bandwidth, even with pure MPI communication. An example of this is shown by the yellow curve with the open circles in Fig. 98. The remaining Intel 2019 curves (red and yellow with solid symbols) show the expected behaviour. They were obtained by manually overriding the default *libfabric* setup via a set of environment variables provided by the Marconi Support Team. Note that this procedure is temporary, only needed until the issue is fixed system-wide.

In order to properly test the newer Intel MPI releases, regarding the inter-node bandwidth, the HP must be enabled at the system level. For this reason, the Marconi Support has configured four SKL compute-nodes where this was done for testing purposes, namely, the compute-nodes r171c15s01–04, to which access was given to us. To use the HP together with the Intel MPI 2019 releases on these compute-nodes, one has to set the Intel MPI environment variable mentioned above to the mount point of 2 MiB HP in the file system, namely, `/dev/hugepages`. In practice, this can be done by simply prefixing the usual `mpirun`/`srun` command according to

```
I_MPI_SHM_FILE_PREFIX_2M=/dev/hugepages mpirun <options> ./<executable>
```

Using the HP as explained before yields the curves shown in Fig. 99. Starting with the Intel 2018 results, the pure MPI case (solid blue curve with open triangles) together with the HP shows the same behaviour as before without HP (Fig. 98). Even though not shown here, using the Intel 2019 release instead lead to the exact same results. Therefore, it appears that the HP does not affect the pure MPI inter-node communication.

Focusing now on the inter-node communication between MPI-3 SHM windows, the story is different. For the earlier Intel 2018 MPI release (dot-dashed blue curve with closed triangles), although there is still a significant drop in bandwidth for larger messages, it is noticeably less pronounced than what is seen on the corresponding curve of Fig. 98. This can only be an effect of the HP, which is the only difference between them. Even more importantly, for both the newer Intel MPI releases (*Version 2019 Update 1: Engineering Build 20180920* in red and *Build 20181016* in yellow), using the HP completely removes the bandwidth drop. It is clear from the plot that both these Intel MPI releases produce the correct results, yielding a monotonic increasing function for the inter-node bandwidth, saturating at values very close to the theoretical maximum (100 GBit/s). In this sense, it even slightly outperforms the pure MPI case (solid blue curve with open triangles), which displays a dip around 100 MiB. This confirms the Intel claim that the issue had been fixed when they closed the corresponding ticket. It is also important to mention that, at the time of writing of this report, there was still no documentation available on the new environment variable `I_MPI_SHM_FILE_PREFIX_2M`. After contacting Intel about the need to make this information available in their online documentation, we were informed that they were preparing it.

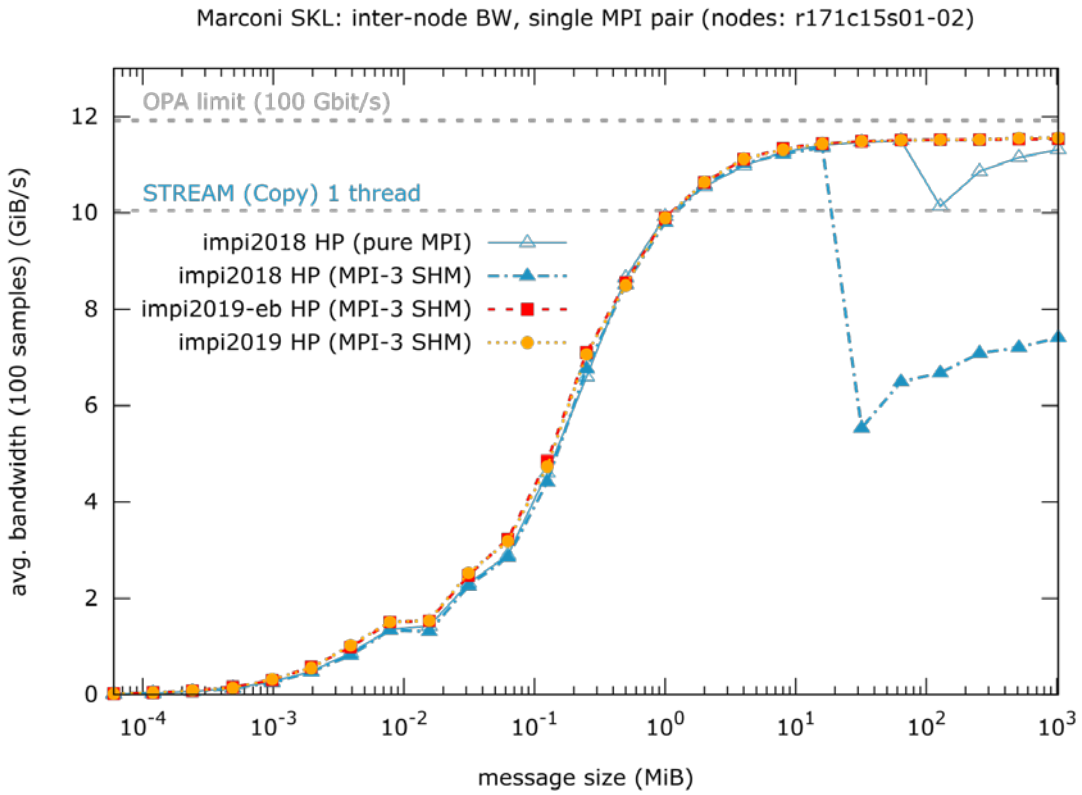

Fig. 99 Same as Fig. 98 applied to the dedicated test nodes (r171c15s01–02), where the HP have by enabled by the Marconi system administration. It is now clear that, using the Intel MPI 2019 releases together with the HP via the `I_MPI_SHM_FILE_PREFIX_2M` environment variable, there is no longer any bandwidth issue.

What remains to be done is to check for any possible negative impact of enabling HP on other codes in general. Only then can one be confident to recommend enabling HP over the whole Marconi system. To this end, some basic tests were already performed by Serhiy Mochalskyy using a small set of representative fusion codes. So far, no performance degradation was measured. This study shall be continued during 2019, in the scope of the HLST-CINCOMP4 project.

8.7. *Intra-node vs. inter-node communication bandwidth (Marconi support ticket)*

While performing the communication bandwidth tests discussed in the previous subsections, a previously reported issue regarding the differences between intra-node and inter-node pure MPI communication [12] was re-encountered. Even if not directly related to the HLST-MPI3-DG project, the subject is nevertheless of general interest for the Marconi-Fusion user community. Therefore, an effort (representing one dedicated additional ppm) in collaboration with S. Mochalskyy, was made to clarify the issue.

The top plot in Fig. 100 shows the communication bandwidth measurements for a pure MPI configuration (without the MPI-3 SHM hybridisation) with a single communication channel (solid blue curves), already presented in Fig. 93 and Fig. 95. The intra-node measurement is now displayed in blue and the inter-node counterpart in green. As before, both the Omni-path maximum theoretical bandwidth and the single thread STREAM main memory access bandwidth are shown for reference. The focus is on the saturated communication bandwidths obtained (values for large message sizes), which are higher when two cores are placed in two different compute-nodes than when they are placed inside the same compute-node. The interpretation for this finding in light of the material already provided in Subsecs. 8.6.1 and 8.6.2 is relatively straightforward.

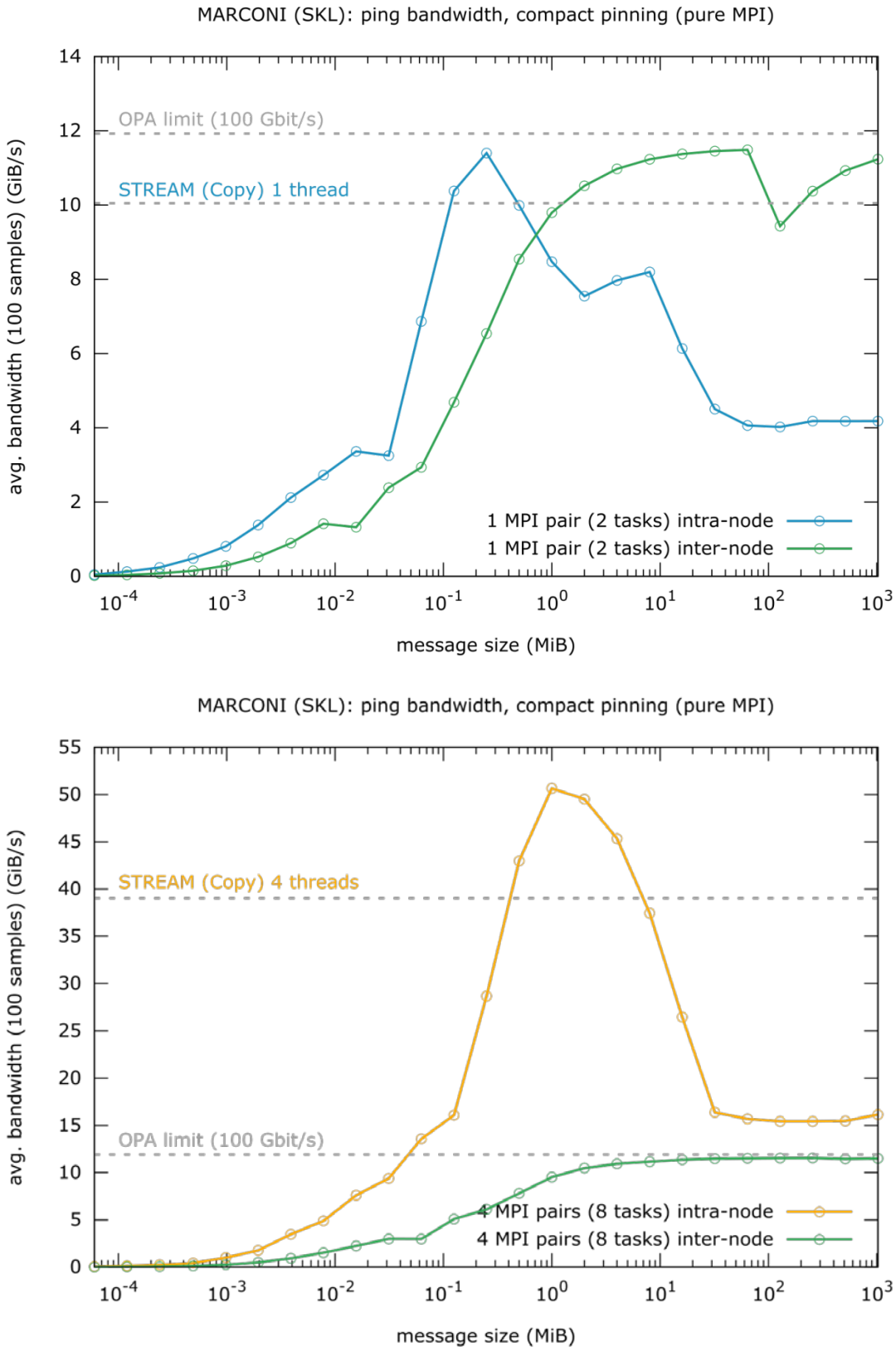

Fig. 100 Intra-node vs. inter-node communication bandwidth using a single communication pair between two MPI tasks (top) and using four communication pairs between eight MPI tasks (bottom). All measurements use the pure MPI communication configuration of the ping test.

First, the Intel Omni-path switched fabric is able to perform remote direct memory access (RDMA), without involving the CPUs, cache hierarchy or context switches of the compute-node. This means that it is independent of the intra-node memory

access bandwidth. Second, there is no loop-back mechanism in place, meaning that inter-node links cannot be used for intra-node data exchange and therefore intra-node communication must be done differently. Indeed, the latter uses either the CMA (the case plotted) or the SM mechanism. As discussed in Subsec. 8.6.1, none of these provides an ideal solution. The former due to a non-optimal implementation of the kernel assist neither the single copy protocol, nor the latter because it involves an additional copy operation. Both imply that only a fraction of the compute-node main memory access bandwidth can be reached for the intra-node communication. However, this limitation can be overcome by using more cores inside the node to concurrently access the data stored in memory, which in our case corresponds to using more communication channels. This amounts to significantly increasing the intra-node memory bandwidth, eventually surpassing the Omni-path network bandwidth. Our measurements show that four communication pairs are enough for the intra-node communication bandwidth to be higher than its inter-node counterpart (lower plot in Fig. 100), which, by construction, is independent of the number of communication pairs involved. This information is of interest for anyone using hybrid (MPI+MPI-3 SHM or MPI+OpenMP) codes for production simulations. It sets a lower bound on the number of intra-node communication channels needed not to degrade communication across compute-nodes. As this is directly related to the number of cores that are set to share memory among themselves, knowing this information *a priori* facilitates the choice on the optimal distribution of resources between communicating cores (MPI tasks) and cores sharing memory (MPI-3 SHM tasks or OpenMP threads).

To conclude this section we would like to acknowledge that the clarification of the intra- vs. inter-node communication bandwidth issue was provided directly by Intel [11], with further details also found elsewhere [10]. Moreover, as a result of the effort hereby reported, this information is to be added in the near future to the Marconi documentation web-pages. Details on the Marconi's communication bandwidth properties can be especially relevant for hybrid codes running on production configurations that might use a sub-set of the intra-node resources to perform large amounts of MPI communication.

8.8. *Summary and outlook*

The main goal of the the HLST-MPI3-DG project is to devise a strategy to improve the scalability of the Fluxo code [1,2]. MPI-3 shared memory hybridisation techniques have been proposed as a candidate solution for the problem. However, this concept requires relatively extensive changes to Fluxo's source code design. Therefore, as a prerequisite, a detailed assessment of the communication behaviour of the Fluxo code is necessary. Only then can one decide whether this optimisation strategy is sound.

So far, the assessment has included measuring the degree of overlap of the communication with computation. This implied profiling the code, which was initially attempted with Scalasca, but then changed to an explicit source code instrumentation, due to the added flexibility. From this it was possible to confirm that some communication hiding is taking place within Fluxo when running it on Marconi's Skylake partition (A3). However, due to the asymmetric weight of the local computation buffers used to hide the communication, only the communication of the fluxes on the cell faces between sub-domains overlaps effectively with computation. This local computation buffer is large enough that about half of the corresponding communication cost is hidden. Conversely, the exchange of the solution values on cell faces between sub-domains was measured to behave mostly as blocking communication, simply because the available computation buffer is too small. This result suggests that some effort should be put into analysing the way the local buffer computations are currently split between the two steps of communication involved, in order to try to have a more balanced distribution of their weights.

Additionally, an optimised communication strategy has been devised and implemented with the help of the project coordinator Florian Hindenlang. It better reflects the NUMA compute-node architecture utilised in modern HPC machines, like Marconi, by introducing a two level communicator hierarchy that distinguishes between communication inside and across the compute-nodes. This allowed splitting the initiation and completion of the communication steps inside and across the compute nodes. The latter, being potentially slower, is started first and completed last. The former is started and completed in between. No other changes to the code, particularly involving the calls which execute the local computation buffers (meant to hide communication), have been made. However, better communication performance was still yielded. This suggests that the refinement made to the granularity of the initiation and completion steps of the communication processes, which results from the two-level group structure implementation, is positive. Our current interpretation is that, by imposing an ordering which better reflects the hardware communication bandwidth characteristics (faster intra-node communication), we favourably change, even if only slightly, the non-trivial global data dependency equilibrium established by the communication between all sub-domains in the simulation.

Ideas to further improve the communication performance are already available, e.g., by generalising the two-level to multi-level group communication, or using *persistent communication* directives. However, the next step should be to perform a detailed tracing analysis of the Fluxo code on Marconi using Intel Trace Analyzer and Collector (ITAC). This has already been agreed upon with Florian Hindenlang as it should bring a much more detailed characterisation of the communication-hiding taking place in Fluxo.

Whether or not the replacement of the intra-node communication with shared memory access in the framework of MPI-3 hybridisation techniques is something that can be recommended in the framework of Fluxo, is still an open question at this stage of the project. Deeper insight is needed in order to provide a definite answer. Moreover, because MPI-3 hybridisation is a topic of interest, we decided to include the extension of the 2D MPI-3 SHM hybrid FFT algorithm in the current project, which was originally developed in the framework of the previous HLST-VIRIATO2 (2017) project [4]. The modifications required were very easy to implement. We simply increased the number of MPI communication channels employed per shared memory window. The results obtained showed better scaling and led to the detailed investigation of the communication bandwidth inside, as well as, across compute-nodes. This allowed us to conclude that the behaviour in these two situations is quite different. In particular, it was found that intra-node communication needs to use several communication channels to be able to match the available inter-node network bandwidth. Additionally, it was found that using shared memory windows as communication buffers to exchange large sized messages across compute-nodes on machines with the Intel Omni-path network fabric, like Marconi, leads to large bandwidth drops, if few communication channels are used. This finding was escalated to Intel in the form of a support ticket during the first quarter of 2018.

The issue was acknowledged by Intel via a private communication by the end of July 2018. Subsequently, a bug fix was implemented and communicated to us by the end of September 2018. The fix was deployed first to a candidate release (*Engineering Build 20180920*) of *Intel(R) MPI Library for Linux* OS, Version 2019 Update 1* and then to the official release (*Build 20181016*). Both these versions were then installed on Marconi. However, the fix further depends on using *huge pages* (HP) for the virtual memory space, which needs to be enabled at the system level. This is not the case for the standard compute-nodes of Marconi. Therefore, for testing purposes, the Marconi Support Team configured four compute-nodes with HP enabled. On these, we could verify that the inter-node communication using MPI-3 SHM windows works as expected. No more bandwidth drops are observed, provided that the HP are used, which further confirmed that enabling the HP at the system-level is a necessary condition. This naturally raises the question of whether or not the HP should be

enabled for the whole Marconi system. The answer at this stage is probably yes. However, a final recommendation can only be given once further tests are made to ensure that there is no negative impact on the generality of production codes used routinely on Marconi. This shall be addressed within the scope of the future HLST-CINCOMP4 project (2019).

It is important to emphasise that the solution for the inter-node communication bug when using MPI-3 SHM windows provided by Intel is not merely something of academic interest. The bug went against the basic hybridisation ideas of having a smaller overall number of larger sized messages. Namely, one large message per group of resources sharing a memory region, rather than a local (smaller) message for every core involved in the computation, as is the case in a pure MPI implementation. This was indeed the motivation behind the MPI-3 SHM hybridisation effort carried during the HLST-VIRIATO2 (2017) project [4]. There, the reduction of communication complexity was proposed to increase its parallel scalability by simultaneously reducing its sensitiveness to communication latency and increasing the communication bandwidths reachable. Even though we could find a way around potential network bandwidth drops in our case, by increasing the number of communication channels between compute-nodes through an increase in the number of communicating tasks per SHM memory window, this could only be regarded a temporary algorithmic fix, not the ideal solution. Furthermore, the fact that on all the Infiniband switched networks tested so far, no change to the original algorithm was required to obtain the ideal result, stands to support the importance of bug fix provided my Intel motivated by our work. Additionally, all the information collected so far on Marconi's communication bandwidth characteristics provided a valuable insight for the project on Fluxo.

8.9. *References*

[1] F. Hindenlang, G. Gassner, C. Altmann, A. Beck, M. Staudenmaier and C.-D. Munz, *Explicit Discontinuous Galerkin methods for unsteady problems*, Computers & Fluids **61** (2012) 86

[2] F. Hindenlang, *Mesh Curving Techniques for High Order Parallel Simulations on Unstructured Meshes*, PhD thesis (2014), Universität Stuttgart

[3] T. Ribeiro, *Final report on HLST project VIRIATO* (2015)

[4] T. Ribeiro, *Final report on HLST project VIRIATO2* (2017)

[5] *Forcheck, A Fortran source code analyzer and programming aid* (http://www.forcheck.nl)

[6] M. Geimer, F. Wolf, B. Wylie, E. Ábrahám, D. Becker and B. Mohr, *The Scalasca performance toolset architecture*, Concurrency & Computation: Practice & Experience **22** (2010) 702

[7] S. Mochalskyy, *Final report on HLST project BEUIFERC* (2015)

[8] T. Fehér, *Final report on HLST project SOLPSOPT* (2016)

[9] J. D. McCalpin *STREAM: Sustainable Memory Bandwidth in High Performance Computers* (1991-2007) (http://www.cs.virginia.edu/stream/)

[10] J. Vienne, *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment* (2014) 33

[11] M. Steyer, *Private communication, Intel* (2018)

[12] S. Mochalskyy, *Final report on HLST project CINCOMP2* (2017)

9. Final report on HLST project REFMULIO

9.1. *Introduction*

The finite-difference time-domain (FDTD) method is one of the most popular numerical techniques used to simulate reflectometry. It offers a comprehensive description of the plasma phenomena. However, this method requires a fine spatial grid discretization to keep the error to a minimum, which in turns implies a high-resolution time discretization to comply with the CFL numerical stability condition. Simulations in three-dimensions (3D) are therefore very demanding computationally, both in terms of floating-point operations and memory resources. They become only possible if the problem can be efficiently distributed over large numbers of resources. The `REFMUL3` code was developed in 2016 within a collaboration between the Instituto de Plasmas e Fusão Nuclear (IPFN-IST) in Lisbon and the HLST, precisely to meet this goal [1]. It is a 3D full-wave code using the Yee scheme [2] with full polarisation that simultaneously copes with o- and x-modes, supports a general external magnetic field and a dynamic plasma. `REFMUL3` employs a hybrid MPI/OpenMP parallelisation using an explicit 3D domain decomposition, which yields very good scaling properties up to a few thousands of cores. This has opened the door to the simulation of much larger domains (grid-counts) which, as expected, exposed new challenges regarding data input/output (I/O) operations. This project aims at addressing those challenges by improving the I/O capabilities of `REFMUL3`.

9.2. *Parallelisation of REFMUL3*

The domain decomposition implemented in `REFMUL3` relies on `MPI_Dims_create` to automatically find a suitable distribution of the resources (cores) over its three spatial dimensions. A Cartesian virtual topology is then created by invoking `MPI_Cart_create` and the process ranks (MPI tasks) are mapped to a 3D coordinate system using `MPI_Cart_coords`. Additionally, a thread-based parallelisation (`#pragma omp parallel for`) can be applied to the slowest varying index (outer loop), which in `REFMUL3` corresponds to the *z*-direction. This choice was made to optimise memory access speed (cache reusage) and takes into account that the numerical kernel is largely symmetric, in terms of the numerical stencil, in all spatial directions.

The MPI parallelisation of `REFMUL3` does not allocate the global computational domain. Instead, it decomposes it into subdomains, each belonging to a different MPI task that locally allocates only the corresponding memory. Each subdomain contains one extra cell per face in all three dimensions, called a ghost-cell, where the values of the corresponding faces of the neighbouring subdomains are stored. This is a necessary and sufficient condition to enforce continuity of the distributed solution across the subdomains for the algorithm under consideration. This rule is, however, relaxed for the subdomains, which include faces that are part of the global domain boundaries. There, the neighbours are replaced by boundary conditions, and therefore no ghost-cells are needed.

The subdomains are defined using the concept of first and last grid-node global indexes. As the name suggests, these specify the global index values of the first and last grid-nodes of each subdomain in each dimension (x,y,z). They are calculated using the global domain size and the number of MPI ranks used, together with the MPI rank of the task to which the subdomain belongs. The size of each subdomain in each dimension is calculated from the difference between the corresponding last and first grid-node indexes. This technique is quite flexible as it allows the subdomain sizes to differ between MPI tasks, lifting the common constraint that the grid-count must be a multiple of the number of tasks used, which is very handy when dealing with staggered grids. It is also possible for any task to easily calculate the set of

numbers characterising any other subdomain (size, first and last grid-node indexes) in the simulation.

The local multi-dimensional subdomain memory allocation is done in a linear fashion using a C-pointer, with the length given by multiplying the grid-counts in each dimension (including the ghost-cells). A mask is provided to access the locally allocated memory using the global index values in each dimension via a C macro that makes the mapping on-the-fly. For a more detailed description of the `REFMUL3` hybrid parallelisation, please refer to the final HLST-REFMUL3 project report [1].

9.3. *Parallel I/O*

9.3.1. Strong scaling of `REFMUL3`

In order to understand why I/O operations became a bottleneck in `REFMUL3`, we must first demonstrate the success of `REFMUL3`'s parallel implementation described before. This can be done by looking at the strong scaling from a typical production case, as shown in Fig. 101. It is very important to note that no I/O operations were performed in these simulations. Otherwise, the scaling would have been significantly impaired because the I/O operations were only available sequentially at the time. For instance, it was measured then that typical (sequential) disk I/O operations represented 30% of the total execution cost of an OpenMP simulation on a single node (36 cores on Marconi's Broadwell partition). This fraction obviously increases with the number of resources used since, unlike the rest of the code, the sequential I/O cost, by definition, does not scale.

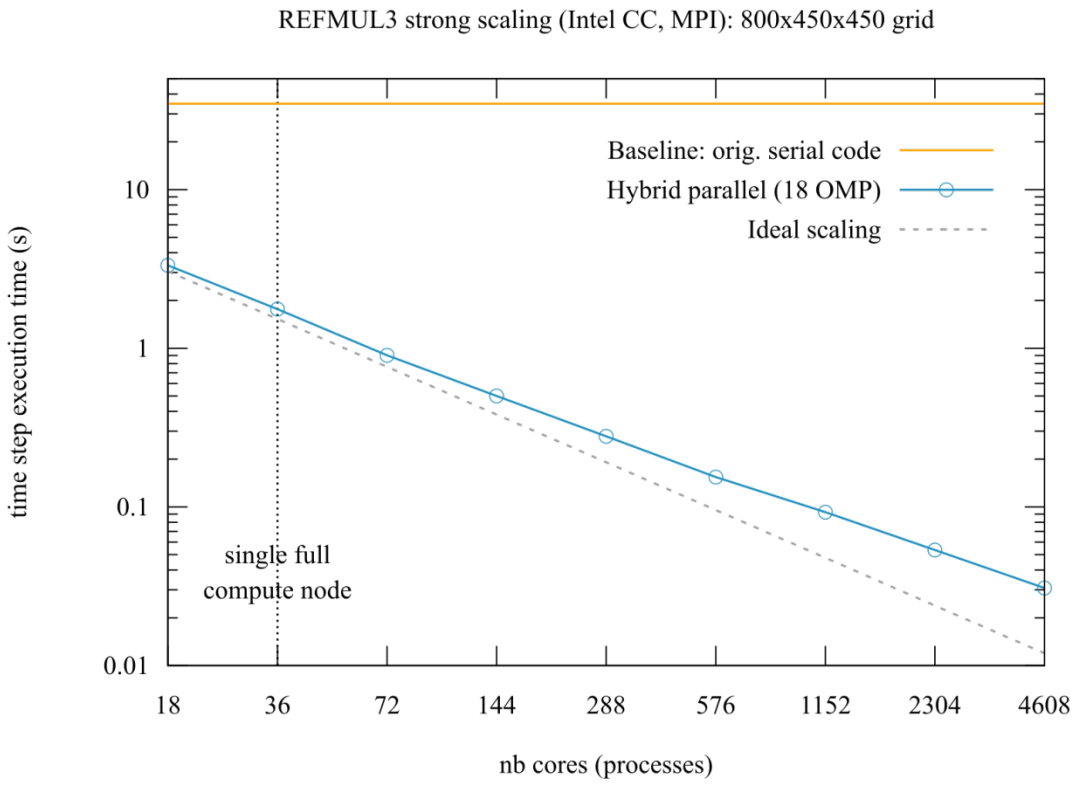

Fig. 101 `REFMUL3` strong scaling on a 800x450x450 (*x*,*y*,*z*)-grid. A hybrid OpenMP/MPI configuration with 3D domain decomposition and 18 OpenMP threads per MPI domain is shown (blue). The baseline execution time needed by the original serial version of the code is also shown (yellow) to highlight the performance gains. The average execution time per iteration is plotted. The measurements were made on the former Broadwell nodes of Marconi.

9.3.2. Previous solution: multi-file serial HDF5

As already mentioned in the introductory Sec. 9.1, addressing the I/O bottleneck in REFMUL3 is the motivation for the HLST-REFMULIO project. The solution must rely in the usage of parallel I/O techniques, which was already recognised during the HLST-REFMUL3 (2016) project [1]. To that end, one simple solution was devised and implemented at the time. Building on top of the pre-existing sequential I/O infrastructure of REFMUL3, each MPI task independently writes a (serial) HDF5 [3] file to disk. Each file contains the requested data (physical quantity) on every grid-node of the subdomain belonging to the task that writes it, identified by including the MPI Cartesian virtual topology coordinates (MPI ranks in each spatial direction) in the filename. Additionally, a single XDMF (eXtensible Data Model and Format [4]) descriptor file is saved to disk by the master MPI task to allow the HDF5 files to be opened directly using ParaView [5] for visualisation purposes. This file contains a collection of all descriptors, one for each subdomain, which are simply the global index values of their first grid-node and the size of their data set. As explained in the previous section, this can be easily calculated by the master rank for all MPI ranks.

The solution described, using a different HDF5 file per subdomain (MPI task) in the simulation, solves already two of the most pressing I/O issues. Firstly, it allows performing I/O operations on domains, which do not fit memory-wise in a single compute node, which was not possible with the original sequential I/O infrastructure. Secondly, it should be able to potentially deliver very good I/O scaling performance, provided that the number of MPI tasks is not too large (below a couple of thousands). The constraint on the number of MPI tasks is not, in principle, a very strong limitation if one keeps in mind that REFMUL3 employs a hybrid MPI/OpenMP parallelisation and runs very efficiently with many threads per MPI task (18 in the case shown in Fig. 101). However, it must be mentioned that, if much bigger domain sizes were foreseen, which require tens of thousands of cores, this approach would most probably break down as the pressure put on the file system builds up due to the handling of very large numbers of HDF5 files being accessed simultaneously. Perhaps more importantly, there is another clear disadvantage of this solution. Namely, that exactly the same domain decomposition used when writing the data to disk, i.e. the same number of MPI tasks with the same distribution over the three spatial dimensions, needs to be used for subsequent I/O operations. This is a rather limiting constraint, especially if one thinks about a check-pointing and restart infrastructure, which is something desired for REFMUL3 and explicitly mentioned in the HLST-REFMULIO project proposal.

9.3.3. New solution: single-file parallel HDF5

To overcome the parallel I/O dependency on the domain decomposition, the usage of the parallel version of the HDF5 library (PHDF5) is proposed here. It provides a framework that allows to have a single HDF5 file containing the requested data-set on the global domain grid, written collectively by all MPI tasks. This means that, in the file, each MPI task must access the spatial region of the global domain corresponding to its subdomain. The HDF5 library provides an elegant way to achieve this using the *hyperslab* concept. It allows selecting dataspace regions inside the HDF5 file by specifying access pattern parameters: *start*, *stride*, *count* and *block*. This fits very well our needs since the sizes of the subdomains can vary, as we have seen in Sec. 9.2. Due to the memory distribution concept employed, which defines each subdomain in terms of their first and last grid-node using global indexing, it is trivial to map this to the hyperslab parameters above to define the HDF5 file access pattern to be followed by each MPI task. However, there is an additional complication related to the local memory access by each task.

By definition, the simulation data is contiguous on the global domain grid. The same must obviously apply to the data set stored in the (parallel) HDF5 file. This means that all ghost-cells, which are part of the subdomains, must be excluded when

accessing the memory for I/O purposes. In other words, the access to local subdomain data in memory is discontinuous, since the ghost-cells in each dimension must be skipped. So, it is not enough to pass a pointer to the place in memory where the data is located to the HDF5 functions. Either one uses an intermediate temporary buffer to store the data (subdomain without ghost-cells) contiguously, or one specifies the appropriate discontiguous memory access pattern. The former is much simpler to implement, but essentially doubles the memory requirements due to the needed data replication, which most probably also implies extra overhead costs. The latter does not require any extra memory allocation, since the subdomain data can be accessed directly. For this reason, we chose to implement it. In particular, the devised solution uses again the HDF5 concept of *hyperslabs* to choose only the "real" grid-cells of the subdomains stored in memory, therefore excluding the ghost-cells. Finally, as was the case for the multi-file serial HDF5 solution developed during the previous project [1], a single `XDMF` descriptor file is also saved to disk by the master MPI task to complete the parallel data output (O) infrastructure. The difference is that it now simply contains the descriptor for the global domain.

The corresponding parallel input (I) of data was implemented following the same principles, except for the descriptor files, whose purpose is solely to make the output data ready for visualisation in ParaView. The data stored in the (single) HDF5 file is read in parallel by all MPI tasks. The hyperslab construct is used by each MPI task to both read in the data portion from the file corresponding to its subdomain and store it locally in memory including the ghost-cells. Together with the parallel output functions, this comprises what is needed to implement the check-pointing and restart file infrastructure. Furthermore, it allows reading in antenna information stored in external (HDF5) files, which was another specific request for the project work. Using the already built-in input flags, it is now possible to construct the antenna in three different ways:

- default option: use the basic wave-guide and antenna horn, which are hard-coded in parallel in the source code; write it to a single HDF5 file as part of the standard code output (`zFrm_P_t_0.h5`);
- `--zfl` input flag: read (in parallel) a partial antenna description from a single external HDF5 file and append to it the basic hard-coded wave-guide and antenna description mentioned before; write everything to a single HDF5 file as part of the standard code output (`zFrm_P_t_0.h5`);
- `--zfl & --noant` input flags: read the whole wave-guide and antenna structure exclusively from a external HDF5 file;

Another specific request of the project coordinator comprised the calibration of the impulse response. During the original HLST-REFMUL3 project, the implementation developed allowed only to run a simulation using previously written calibration files. The domain-decomposed versions of the code open these files by letting the master MPI rank read them and then broadcast the data to the remaining ranks. However, they could only be produced using the serial/pure OpenMP versions of REFMUL3, which necessarily excluded larger domains, namely, the ones that do not fit on a single compute-node. This limitation was lifted. It is now possible to create impulse response calibration files using the domain decomposed REFMUL3 (pure MPI or hybrid OpenMP/MPI). During the main cycle iterations, the calibration data is stored on local temporary arrays by the tasks that include the wave-guides in their subdomains. The remaining tasks simply keep them allocated in memory and initialised to zero. Note that these arrays are small compared to the sub-domain grid-count, so their replication does not impose any relevant disadvantage (please refer to [1] for more details). After the main cycle is exited, the data is reduced (in-place) from all tasks to the master task rank (0), which performs a global sum operation. Since all the tasks that are not involved in the calibration have zeros therein, the global sum produces the correct result. The master task then writes the reduced data to the calibration files.

At last, it is important to mention that, during the last stages of development, a bug has been found in REFMUL3. Unfortunately, it only manifests itself rather late in terms of number of iterations of the simulation, which makes it very time consuming to solve. About two weeks of the project budget have been dedicated to help the project coordinator to identify its causes. During this process, several possibilities have been excluded. Most prominently, it could be proven that the bug is not caused by any parallelisation error, which is very important in the context of the HLST project. The results are independent of the choice used for the domain decomposition, including when running REFMUL3 in pure OpenMP mode, without any domain decomposition. The bug manifests itself in all these situations, provided enough iterations are simulated. Solving this issue became the top priority of the project coordinator during the last weeks of the HLST-REFMUL3, which meant that some compromises had to be made. In particular, the restart file infrastructure could not be completed, even though the needed functions to write out and subsequently read in the data have been implemented, as explained above. The last step required deciding which quantities need to be included in the restart file, which is not yet available information. However, being trivial to implemented, the project coordinator Filipe da Silva decided he would do this afterwards himself, once the bug has been fixed.

9.3.4. **Results**

The above described method uses the parallel HDF5 library to handle REFMUL3's I/O operations, independently of the domain decomposition. The part corresponding to the code's output was first implemented and validated. This is illustrated in Fig. 102, where the *z*-component of the electric field (E_z) data for a given simulation is shown. The data was saved using both available parallel I/O methods, and is plotted together for direct visual inspection. The top part of the plot corresponds to the newly implemented solution using a single parallel HDF5 file. The lower part corresponds to the alternative method using a different serial HDF5 file per subdomain, as can be inferred from the different colours in the figure. As expected, a perfect match between the surfaces in the upper and lower parts of the plot is obtained.

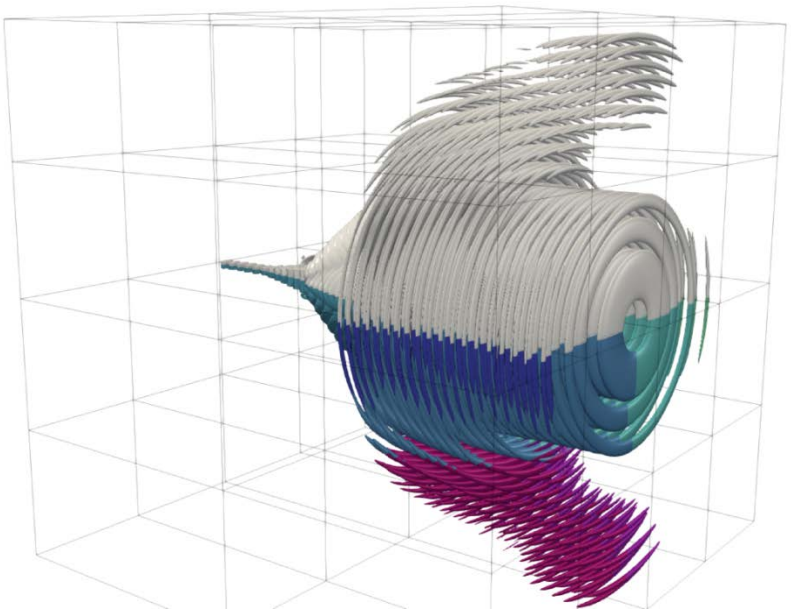

Fig. 102 3D REFMUL3 simulation of a 720x700x700 domain decomposed over 32 MPI tasks. Iso-surfaces of the *z*-component of the electric field (E_z) after 2000 iterations are shown. The upper half of the plot (grey) shows to the global domain written to a single file by all MPI tasks using the parallel HDF5 library. The lower half shows the same data but decomposed into the

corresponding subdomains, each represented by a different colour. Each subdomain is stored in an independent HDF5 file written by the MPI task it belongs to (serial HDF5).

The performance measurements made during the simulations shown in Fig. 102 provide some preliminary insights on the costs of I/O operations. They indicate that a competitive performance can be obtained with the parallel HDF5 implementation, at least in comparison with its multi-file serial HDF5 counterpart. Using 32 MPI tasks distributed over 8 compute nodes (and 12 OpenMP threads per MPI task), the total I/O cost was of about 30 s for the former, and about 60 s for the latter. In reality, the advantage in terms of bandwidth is even slightly larger than a factor of two because the total amount of data written was larger in the former case (63 GB) than in the latter (45 GB). The reason for this lies in the data compression, which was deactivated in the parallel HDF5 case as it crashed the code. It is not yet clear why, but it is planned to investigate it in the future.

Although a systematic scaling performance assessment of both parallel I/O methods is still lacking, several additional measurements have been already made. In particular, the granularity of performance counters implemented on REFMUL3 had to be increased to better suit them for new measurements. From these it was possible to confirm the preliminary finding that the parallel HDF5 performance was superior to the one yielded by the multiple serial HDF5 files solution, when a relatively small number of MPI tasks is used. However, for typically more than 64 MPI tasks, the trend inverts, and the latter become more performing. Most probably, this is a result of lack of time to invest in the topic of performance tuning within PHDF5, which could potentially bring significant gains. Working on this topic is an important task, but at the same time, also a rather involved one, which at this stage can only be left as recommendation to the project coordinator. Perhaps as something to be done in the framework of a future HLST project.

9.4. *Visit to the Project Coordinator in Lisbon*

Tiago Ribeiro visited the project coordinator Filipe da Silva at the IST-IPFN Lisbon for three weeks (09–27 July 2018) at the very beginning of the HLST-REFMULIO project. This visit was instrumental to define in detail the strategy for the activities conducted in the subsequent months, which are being reported in this document. Furthermore, since the parallelisation of REFMUL3 was made by Tiago Ribeiro in the framework of the previous HLST-REFMUL3 project [1], this visit served as well to address many questions on this topic. These were mostly concerning implementation details, which had arose after a thorough inspection of the complete REFMUL3 source code by Filipe da Silva.

9.5. *Summary and outlook*

The REFMUL3 code, which is being developed at the IPFN-IST association in strong collaboration with the HLST, yields very good scalability over a few thousands of cores. However, I/O has become a major bottleneck, as the parallelised code allows for much larger grids in comparison to its two-dimensional predecessors (see HLST-REFMUL2P (2014) report [6]). The current project, HLST-REFMULIO (2018), requested support for the development of parallel input/output (I/O) capabilities for REFMUL3, including, if time allows, the implementation of a check-pointing and restarting infrastructure.

The simplified parallel I/O solution, implemented towards the end of the previous project HLST-REFMUL3 (2016) [1], writes one (serial) HDF5 file per subdomain (MPI task). Even though it addresses the basic I/O bottleneck issue to some extent, it has, however, an important drawback in that it is domain decomposition dependent. In other words, the same distribution of resources over the simulation's spatial domain is needed if data stored in I/O files are to be reused by REFMUL3. The solution implemented now lifts this limitation by having all subdomain data stored within a single parallel HDF5 (PHDF5) file, accessed collectively by all MPI ranks.

The new PDHF5 output functions give faster writing speeds compared to the former simplified multi-file serial HDF5 solution for moderate numbers of MPI tasks. For more than 64 MPI tasks, however, the situation inverts. This is most probably due to the little amount of time invested in tuning the parallel HDF5 implementation. This highlights the need to do so in the future. The PHDF5 input functions additionally allow the specification of localised functions inside the computational domain, like antennas and wave-guides. They can now be read from pre-produced files outside REFMUL3 during the initialisation phase of a simulation, be appended to the basic hard-coded default antenna and wave-guide configuration, or simply replace it. Additionally, the PHDF5 I/O functions together provide the ingredients needed to have a basic check-pointing and restart file infrastructure for REFMUL3.

The possibility to perform the impulse response calibration for domain decomposed simulations was also implemented. This was specifically requested by the project coordinator, since being restricted to non-distributed domains, as was the case before, barred the simulation of larger cases, which do not fit in a single compute-node.

Unfortunately, towards the end of the project, a bug has been detected in REFMUL3 by the project coordinator. While it could be checked that its cause is not related to the parallel implementation if the model, some time has still been dedicated to help Filipe da Silva finding the issue. At the time of writing of this report, however, the bug had not yet been solved. This affected somewhat the project activities during this time. In particular, the later stages of implementation of the check-pointing infrastructure, namely, deciding which quantities should be part of the restart file, could not be finish. However, this should be something relatively simple to be done by the project coordinator, as soon as the bug has been fixed.

In summary, the work fulfilled the proposed milestones by providing an effective domain decomposition independent parallel HDF5 I/O capability to REFMUL3 that can be directly analysed in ParaView. What remained to be done due to lack of time was the PHDF5 I/O performance tuning and optimisation. This should in principle bring significant gains in performance, and is hence left as a recommendation for future work. Finally, even if beyond the scope of the current project, there is one further topic that is worth mentioning, as it can potentially improve the overall code scalability, especially when higher number of resources are used. Namely, the overlapping of communication with computation. Even though it should be relatively straightforward to implement in REFMUL3, as the existing code already includes all the necessary ingredients, it still requires time to be addressed. As such, it is also left for future work.

9.6. *References*

[1] T. Ribeiro, *Final report on HLST project REFMUL3* (2017)

[2] K.S. Yee, *Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media*, IEEE Trans. Antennas Propag. **14** (1966) 302

[3] The HDF Group, *Hierarchical data format version 5* (2000-2017) http://www.hdfgroup.org/HDF5

[4] *eXtensible Data Model and Format* (2000-2017) http://www.xdmf.org/

[5] Sandia National Labs, Kitware Inc and Los Alamos National Labs, *Paraview: Parallel visualization application* (2000-2017) http://paraview.org

[6] T. Ribeiro, *Final report on HLST project REFMUL2P* (2014)