EUROfusion

EUROFUSION WPISA-REP(16) 18574

R Hatzky et al.

HLST Core Team Report 2016

REPORT

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

HLST Core Team Report 2016

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

Contents

1. Executive Summary

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

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

Project acronym	Core team member	Status
CINCOMP	Serhiy Mochalskyy	finished
FWTOR-16	Michele Martone	finished
GOKE	Tamás Fehér	running
GRIMG	Kab Seok Kang	finished
JORSTAR	Serhiy Mochalskyy	finished
MGBOUT3D	Kab Seok Kang	finished
REFMUL3	Tiago Ribeiro	running
SOLPSOPT	Tamás Fehér	finished
VIRIATIO	Michele Martone	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 IFERC-CSC computer, HELIOS and 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 projects GOKE and SOLPSOPT project.

The Gysela code is a nonlinear global full-f gyrokinetic code that can be used to model turbulence and heat transport in tokamaks close to reactor conditions. The aim of the GOKE project is to evaluate and optimize kernels of the Gysela code on different accelerators, with primary focus on the Intel Knights Landing (KNL) architecture.
The roofline model was used to draw a general comparison between the new Knights Landing architecture and the existing Broadwell nodes of MARCONI. The stream benchmark was used to measure the memory bandwidth of a KNL test node at the Occigen supercomputer. One could conclude that efficient usage of the high bandwidth memory integrated into the KNL processor will probably be an important step in achieving good performance on the KNL processor.
A set of OpenMP benchmarks was performed to test the overhead of different OpenMP constructs. The overhead time increased by a factor of 2–3 on the KNL because of the reduced clock frequency and the larger number of threads.
The execution time of the Gysela code was measured on a KNL test node and the results were compared to a Broadwell node of Marconi. For the studied test case the wall-clock time for the numerical calculation is 32% longer on KNL. The VTune profiling tool was used to identify which subroutines take most of the execution time, and work has started to improve those subroutines that give relatively long execution times on the KNL architecture.

The SOLPSOPT project aims to optimize the SOLPS package, which is a collection of several codes. The two main components are the Eirene and the B2 codes. The B2 code is a plasma fluid code to simulate edge plasmas and the Eirene code is a kinetic Monte-Carlo code for describing neutral particles. This project was carried out in collaboration with Lorenz Hüdepohl from the Max Planck Computing and Data Facility (MPCDF).

The OpenMP parallelization of the B2 code was improved in 2014 and 2015 yielding a factor of six speedup for the B2 code. This work was originally done in SOLPS 5.0. Lorenz Hüdepohl is currently incorporating these changes into SOLPS ITER.
The work in 2016 focused on improving the MPI parallelization of Eirene. The parallel code gave slightly incorrect results; this had to be corrected first. A testing framework was constructed to help find the error in the parallelization of Eirene. The framework can automatically generate unit tests for legacy Fortran codes. Using these tests, the error was found in the coupling subroutines between B2 and Eirene. The problem was solved, and the code now calculates correct results in parallel mode. The bug fix has been implemented in the SOLPS-ITER version of Eirene.
The original parallelization strategy was improved by calculating the communicators in advance before the Monte-Carlo loops. This can give a significant performance improvement for test cases with high post processing overhead.
The parallel performance of Eirene was further improved by adding two parallelization strategies (APCAS, balanced). Both of them provide good scaling for the case when the number of MPI tasks are less than, or comparable with the number of strata. The balanced strategy is also efficient for a large number of MPI tasks. These parallelization strategies are now integrated into SOLPS-ITER, and the balanced strategy is the new default parallelization method.
The technical details of the coupling between the OpenMP B2 code and the MPI Eirene code were investigated. The current implementation is adequate for the hybrid code, and we do not expect any performance degradation in the coupled code.

Kab Seok Kang worked on the GRIMG and MGBOUT3D projects.

The contribution of the project coordinator of the GRIMG project was to implement the matrix generation routine for the Neumann boundary condition with zero-order approximation. As a result, GRILLIX can handle the Dirichlet and Neumann boundary conditions on a general shaped domain. This may be improved in a further step by adding first- and second-order approximations for the Neumann boundary condition.
An unreasonable implementation of the restriction operator was found which used only the value of one point at the finer grid to get the value at the coarser grid. This caused a five time pre- and post-smoothing with the Jacobi smoother in order to achieve convergence. Therefore, we had to modify the restriction and prolongation operators to achieve faster convergence. In addition, we implemented Jacobi and Gauss-Seidel smoothers. The multigrid solver itself was rewritten. Also, a multigrid solver as a preconditioner for GMRES was implemented. In the future we will have to investigate which of the two algorithms is the more efficient one.
As a next step we need to verify the modified code and to tune some numerical parameters, e.g. an optimal number of smoothing iterations for the Jacobi and Gauss-Seidel smoothers.

For the MGBOUT3D project we developed and tested multigrid algorithm classes for 2D problems. Especially, we compared three different types of memory allocation and selected the memory pre-allocation as it had the best performance. The numerical results show that the multigrid method has good strong and weak scaling properties up to more than 16K MPI tasks.
The BOUT++ team finished the 2D parallelization with OpenMP and on our side we finished the OpenMP/MPI hybrid multigrid solver. One has to consider the benefit from multithreading in comparison to the penalty overhead time for using the threads in the OpenMP/MPI hybrid code. So, one should investigate further what setting is the optimal in the hybrid algorithm.
For the Poisson problem with constant coefficients, the basic solver (FFTW) in the BOUT++ has better performance than the multigrid solver. However, we expect this to change for complicated problems. To confirm this, the project coordinator would need to construct a reasonable but demanding example to compare both solvers.
In addition, we prepared the implementation of the multigrid classes for the 3D problem. The BOUT++ team is further investigating how the multigrid solver for 3D

problems can be efficiently used. The multigrid classes for 3D problems might be finished in a follow up project.

Michele Martone worked on the FWTOR-16 and VIRIATIO projects.

FWTOR is a full-wave code solving Maxwell's equations for the propagation and absorption of electromagnetic wave beams in tokamak plasmas using the FDTD method. Most of its execution time is in the FDTD iterations, updating a 2D grid via several loops.
Due to unforeseen workforce shortage the project coordinator could not provide us in time neither with the necessary references for serial checkpoint-restart functionality nor with a 3D physics model. Nevertheless, we provided partial support for 3D communications in the HLST communication module for FWTOR (HMF). These communication routines have been added to serve as parallelization infrastructure for a 3D physics model to be developed by the project coordinator in the future. As the parallel I/O based checkpoint-restart functionality within our HAC module had to be postponed, we nevertheless advise it to be addressed with priority before planning a campaign of large jobs: I/O is the current performance bottleneck.
The project resulted in an optimization of the existing MPI parallelization by reducing the latency impact. This was achieved by an aggregation of successive array exchanges and by trading off exchanges for computation via subdomain overlap. The combined performance improvement of these optimizations can reach a factor of eight.
The HMF version we have obtained is capable of serving the post-2015 FWTOR code while supporting any combination of the two proposed MPI-related optimizations on the original 2D physics. To benefit from both optimizations, however, certain code adaptions are still necessary.

The original checkpoint / restart technique in VIRIATO was inefficient. Now, after having interfaced VIRIATO to our HAC module based on the ADIOS I/O library, checkpointing a case with 256 points in each direction with over 200 Hermite moments should take a few seconds and write at over 20 GiB/s to the LUSTRE file system of the HELIOS machine. This is a speedup approaching the order of a thousand over the original. Further experiments on the same machine align with our experience and findings with the ITM-ADIOS project ([A14]). The performance problem of checkpoint/restart in VIRIATO on HELIOS (and similar machines) seems to be resolved.
Since our solution uses the ADIOS file format for storing checkpoints, we recommend the VIRIATO users to convert to their own formats of choice for post-processing and archiving. Apart from the ADIOS-provided converter, they can use our converter example program.
In order to keep the HAC-based I/O of VIRIATO in good operation it is necessary for the VIRIATO users to keep an eye on the I/O statistics. Diagnostic information is being provided at each run. In case of e.g. incompatibilities requiring maintenance they are encouraged to contact the HLST.

Serhiy Mochalskyy worked on the CINCOMP and JORSTAR projects.

In the CINCOMP project we analyzed the new supercomputer MARCONI before the official production phase. Different benchmarks and tests were made to determine the performance of the new system. Issues were found that significantly limited its use. Some of them were resolved by the MARCONI support team, however others are still under investigation.
The Stream and Intel MPI benchmarks show a high memory bandwidth inside a single *Broadwell* node as well as for the inter node communication. The test code that computes the global reduction shows also good performance scalability inside one node. However, the Intel MPI PingPong test for the inter node benchmarks had

shown a drastic bandwidth drop for message sizes larger than 8–32 kB, which was finally resolved by the maintenance on 2016.09.30.
Scalability tests of codes like STARWALL and GYGLES were also performed. It was found that in spite of remaining unresolved issues the performance on MARCONI is higher than on HELIOS for any node numbers.

Within the JORSTAR project the STARWALL code has been analyzed for potential improvements and optimization by means of MPI parallel computation. It was found that for a large production run the whole code must be parallelized due to the lack of memory for saving the input/output matrices and due to the computational time.
All sequential LAPACK subroutines were analyzed and selected for replacement by their parallel analogues from the ScaLAPACK library. All these subroutines were replaced in the final code version because of the required large size of input matrices.
The LAPACK subroutine for the eigenvector solver was replaced by the parallel subroutine counterpart from the ScaLAPACK library. A very good agreement was found in terms of the eigenvalues. In addition, the correctness of the results was proven by their consistency with the underlying physical model. The ScaLAPACK subroutine has shown better performance not only by using several processes in parallel but also in sequential mode due to the advantage of using IEEE arithmetics. Good parallelization efficiency was obtained for this subroutine for large problem sizes.
Finally, the complete code was parallelized including all LAPACK and user written subroutines. The new parallel version of the code provides identical results in comparison with the original one. This includes the part of the code handling the magnetic coils. The parallelized version allows production runs with much larger numbers of finite elements, i.e. to resolve the realistic wall structure. The simulation time in such a case is less then 12 hours using 128 computing nodes on HELIOS.

Tiago Ribeiro worked on the REFMUL3 project and he did supplementary work on the VIRIATO project of 2015. In addition, he worked on process pinning on MARCONI-Fusion.

Simulation of reflectometry using a Finite-Difference Time-Domain (FDTD) code is one the most popular numerical techniques used, as it offers a comprehensive description of the plasma phenomena. This method requires however a fine spatial grid discretization, which implies a high-resolution time discretization to comply with CFL stability condition. As a consequence, simulations in three-dimensional become prohibitive in computational time. The only way to circumvent this is to develop a parallel three-dimensional code. The REFMUL3 project aims precisely at this goal, namely, obtaining a parallel scalable three-dimensional FDTD simulation code with the same name.
Starting from a serial version of the code, the single-core profiling and optimization have been completed, resulting in an overall 1.6x speedup factor. OpenMP threads have also been implemented successfully to exploit the parallelism at the node-level, with a 11.2x speedup obtained on 20 threads in IPP's TOK cluster.
The task of three-dimensional MPI domain decomposition is also well underway. First an implementation of a stand-alone Jacobi iteration Poisson solver was made to test and prove the parallelisation concept. With similar numerical properties to `REFMUL3`, the very good scaling properties that the Jacobi iteration solver has shown, gave confidence on the parallelisation strategy chosen for `REFMUL3`. The deployment of the same parallelisation techniques to `REFMUL3` has then started, with the infrastructure for the domain decomposition already in place. This includes the MPI initialisation, creation of the MPI communicators and the necessary changes regarding the partition of the global domain into sub-domains.
The remaining step is currently ongoing, namely, the implementation of the necessary communication steps, which shall be completed during the remaining time of the project.

VIRIATO uses a unique framework to solve a reduced (4D, instead of the usual 5D) version of gyrokinetics applicable to strongly magnetized plasmas. It is pseudo-spectral perpendicular to the magnetic field and uses a spectral representation (Hermite polynomials) to handle the velocity space dependency. In terms of numerical accuracy, this is rather advantageous as spectral representations are more powerful than grid-based ones. However, being parallelized using MPI domain decomposition over two directions in the configuration space domain, the scalability of the resulting algorithm, which uses extensively the 2D Fourier transforms, becomes a non-trivial problem.
The main goal of this proposal was to improve the parallel scalability of VIRIATO, and the work has been carried out successfully in 2015. The possibility of further improving the parallel scalability of VIRIATO was considered by devising a strategy to take advantage of the node-level NUMA topology of current HPC machine to reduce the transpose communication complexity. The idea is to split the communication inside and outside a compute node. Two different strategies were devised; one based on an MPI/OpenMP hybridization and another on MPI communicators splitting together with shared memory segments, which are now part of the MPI-3 standard. Some tests were made with the latter showing very promising results. Due to time constraints, their implementation in VIRIATO is left as a recommendation for future work.

The investigation on process pinning was made to provide practical guidelines on how to use the process pinning options within the Intel MPI library together with the PBS batch queueing system available on the MARCONI supercomputer. Examples of scripts for pure OpenMP, pure MPI and hybrid OpenMP/MPI cases were given for process affinities which are typically relevant for the codes used within the community. Their impact on code performance was then illustrated with strong scaling results from an FDTD production code on MARCONI. For more detailed information on the Intel MPI library and OpenMP environment variables, the reader is referred to the corresponding technical reference guides.

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

1.2.1. **Dissemination**

Mochalskyy, S. and Hatzky, R.: Experience with new architectures: moving from HELIOS to MARCONI, *Accelerated Computing for Fusion, 28th–29th November 2016*, Maison de la Simulation, Gif-sur-Yvette, France.

Mochalskyy, S. and Hatzky, R.: Experience with new architectures: moving from HELIOS to MARCONI, *NMPP Seminar*, 15th December 2016, IPP, Garching, Germany.

Ribeiro, T., and Hatzky, R.: The experience of the High Level Support Team (HLST), *4th IFERC-CSC Review Meeting*, 15th March 2016, Tohuku, Japan.

1.2.2. **Training**

Kab Seok Kang has visited:
The project coordinator Fulvio Militello to work for the MGBOUT3D project, 4th–8th July, CCFE, Culham, UK.

1.2.3. **Internal training**

The HLST core team has attended:

- HLST meeting at IPP, 19th April 2016, Garching, Germany.
- HLST meeting at IPP, 6th October 2016, Garching, Germany.

Roman Hatzky, Kab Seok Kang, Michele Martone, Serhiy Mochalskyy and Tiago Ribeiro have attended:
Introduction to hybrid programming in HPC, 14th January 2016, LRZ, Garching, Germany.

Roman Hatzky, Kab Seok Kang, Michele Martone, Serhiy Mochalskyy and Tiago Ribeiro have attended:
Fifth European Training session for users of IFERC-CSC, 16th–17th February 2016, IPP, Garching, Germany.

Michele Martone has attended:

- C–C++ multicore application programming, 14th–16th March, 2016, Maison de la simulation, Gif-sur-Yvette, France.
- Intel MIC Programming Workshop, 27th–29th June 2016, Leibniz-Rechenzentrum (LRZ), Garching, Germany.

Serhiy Mochalskyy has attended:

- Intel KNL training session, 18th May, Max Planck Computing and Data Facility (MPCDF), Garching, Germany.
- C++ Language for Beginners, 25th–27th October 2016, Leibniz-Rechenzentrum (LRZ), Garching, Germany.
- Advanced C++ with Focus on Software Engineering, 22th–24th November 2016, Leibniz-Rechenzentrum (LRZ), Garching, Germany.

T. Ribeiro has attended:
Runtime systems for heterogeneous platform programming@MDS, 6th–7th June 2016, Maison de la Simulation, Gif-sur-Yvette, France.

1.2.4. **Workshops & conferences**

Roman Hatzky has attended:
IPP Theory Meeting, 7th–11th November 2016, Schloss Ringberg, Germany.

Fehér, T.: Joint EoCoE-PoP Workshop on Performance Evaluation, 30th May–2nd June 2016, Maison de la Simulion, Gif-sur-Yvette, France.

Fehér, T.: Eirene Parallel Optimization, *WPCD-SOLPS Optimization Final Working Session*, 28th–30th November 2016, IPP, Garching, Germany.

Hölzl, M., Atanasiu, C., Mochalskyy, S., Huijsmans, G., Hatzky, R., Lackner, K., Zakharov, L., Strumberger, E., Orain, F, Artola Such, J.: JOREK-STARWALL Developments, *JOREK General Meeting,* 13th April 2016, Sophia-Antipolis, France.

Kang, K.S.: Multigrid solver in BOUT++, *Computational Methods in Applied Mathematics (CMAM-7)*, 31st July–6th August 2016, University of Jyväskylä, Finland.

Kang, K.S.: Multigrid method and solvers, *BOUT++ workshop*, 19th–21st September 2016, York Plasma Institute, University of York, UK.

Martone, M.: Interfacing Epetra to the RSB sparse matrix format for shared memory performance, *5th European Trilinos User Group Meeting*, Technische Universität München (TUM), 18th–20th April 2016, Garching, Germany.

Mochalskyy, S. and Hatzky, R.: Simulation using MIC co-processor on HELIOS, *Intel MIC Programming Workshop*, Leibniz-Rechenezentrum (LRZ), 27th–29th June 2016, Garching, Germany.

Mochalskyy, S. and Hatzky, R.: MPI Parallelization and Optimization of the Coupled JOREK-STARWALL Codes, *IPP Theory Meeting, 7th–11th November 2016, Schloss Ringberg, Germany.*

Ribeiro, T.: Parallelization of the full-wave code REFMULX, *HPC Days in Lyon*, 6th–8th April 2016, Lyon, France.

Tskhakaya, D. and Kang, K.S.: 2D kinetic modelling of the Scrape-off layer, *International Conference on Plasma Surface Interactions in Controlled Fusion Devices*, Pontificia Universita Urbaniana, 30th May–3rd June 2016, Rome, Italy.

Voitsekhovitch, I., Hatzky. R., Coster, D., Imbeaux, F., McDonald, D.C., Fehér, T.B., Kang, K.S., Leggate, H., Martone, M., Mochalskyy, S., Sáez, X., Ribeiro, T., Tran, T.-M., Gutierrez-Milla, A., Heuraux, S., Hölzl, M., Pinches, S.D., da Silva, F., Tskhakaya, D., Aniel, T., Figat, D., Fleury, L., Hoenen, O., Hollocombe, J., Kaljun, D., Manduchi, G., Owsiak, M., Pais, V., Palak, B., Plociennik, M., Signoret, J., Vouland, C., and , Yadykin, D.: Recent EUROfusion achievements in support to computationally demanding multi-scale fusion physics simulations and integrated modelling, *26th IAEA Fusion Energy Conference, 17th–22nd October 2016, Kyoto, Japan.*

1.2.5. **Meetings**

Roman Hatzky attended on a regular basis:

- CSC management meeting
- CSC European ticket meeting
- HPC Operation Committee meeting
- EUROfusion MARCONI Ticket Meeting

2. **Report on HLST project GOKE**

2.1. *Introduction*

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

The aim of the GOKE project is to evaluate and optimize kernels of the Gysela code on different accelerators. Recently a new generation of the Intel MIC architecture, the Knights Landing (KNL) has appeared on the market, and it is being deployed in HPC clusters. The MARCONI supercomputer will also add a large KNL partition by the end of the year; therefore, the primary optimization target is the KNL architecture.

In this report, the KNL architecture is introduced and the performance of a KNL node is compared to a regular node on MARCONI. Afterwards, the first measurements of the actual performance of Gysela code on KNL are also presented.

2.2. *The Knights landing architecture*

The new Knights Landing processors are available as self bootable processors. Every node of the A2 partition of MARCONI has just one KNL processor. As the opening of the A2 partition got delayed due to technical difficulties, the project coordinator arranged an account at the Occigen supercomputer at CINES, France, where a few KNL test nodes were available. These test nodes were used to perform the benchmarks in this section.

Table 2 shows the basic parameters of the test node at Occigen, the A2 node of MARCONI and the current Broadwell node of MARCONI. Note that all three processors have a separate AVX base frequency (used for high AVX workload), which is lower than the default base frequency. Every core has two Vector Processing Units (VPU) on the KNL architecture. A single thread can issue a vector instruction on both VPUs at every cycle. Similarly to the KNL architecture, one thread on the Broadwell architecture can use two execution ports and issue two Fused Multiply Add (FMA) instructions per cycle. In principle we can reach peak performance on both architectures by using one thread/core. Multiplying rows 1, 3, 4, and 5 in Table 2 gives the number of arithmetic instructions that can be executed in a second. The peak performance in row 6 is calculated that way, considering that only FMA instructions are executed (2 Flops/instruction).

		Occigen test node Xeon Phi 7210 (KNL)	MARCONI A2 Node Xeon Phi 7250 (KNL)	MARCONI A1 node Xeon E5-2697v4 (Broadwell)
1	Cores	64	68	2 x 18
2	Frequency	1.3 GHz	1.4 GHz	2.3 GHz
3	AVX Frequency	1.1 GHz	1.2 GHz	2.0 GHz
4	Vector register	8 doubles		4 doubles
5	FMA execution units	2 Vector Processing Units / core		2 ports / core
6	Peak TFlop/s	2.25	2.61	1.15
7	Power	230 W		2x145 W
8	Memory	96 GB + 16 GB on chip MCDRAM		128 GB

Table 2 Comparison of KNL nodes at Occigen and MARCONI and a Broadwell node.

The KNL processor has a larger number of cores and two of these cores are grouped into a tile that shares an L2 cache (see Fig. 1). The L2 caches are kept coherent across all the tiles. There are different operation modes of the KNL processor, in

which the mapping between the L2 caches and the memory is different. These operation modes will be discussed in the next report.

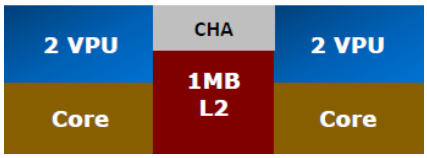


Fig. 1 A KNL tile contains two cores and 1MB L2 cache. Each core has two vector processing units (image source: Intel).

2.3. *Memory bandwidth*

The KNL node has 96 GB of DDR4 system memory and 16 GB of high bandwidth memory integrated on the KNL processor which is called Multi-Channel DRAM (MCDRAM). The stream benchmark (McCalpin, 1995) was used to measure the memory bandwidth of the KNL node at Occigen, and the results are shown in Fig. 2. The system memory on the KNL node can deliver 81 GB/s bandwidth which is slower than what we can achieve on the A1 node in MARCONI. Using the MCDRAM can greatly increase the bandwidth. The MCDRAM in the KNL node at Occigen delivered 332 GB/s memory bandwidth but the KNL hardware is advertised to be able to reach 445 GB/s bandwidth. The tests will be repeated on MARCONI, once the A2 partition is opened.

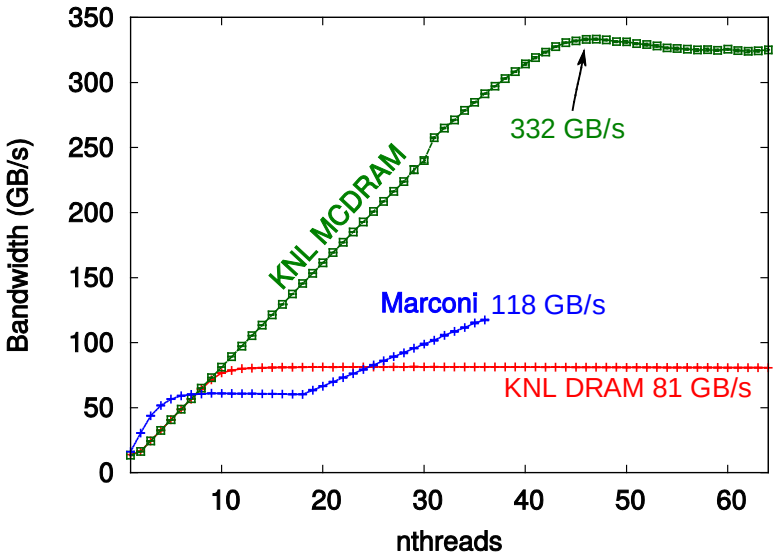


Fig. 2 Memory bandwidth of the KNL node at Occigen compared to the Broadwell node of MARCONI.

The latency of the MCDRAM is comparable to the system memory. The bandwidth per core is also similar to the system memory, as can be seen in Fig. 2 for the range between 1 and 10 cores. But the MCDRAM is able to serve a larger number of cores before it becomes saturated, this is why the aggregated bandwidth is significantly larger than for the system memory.

There are different operating modes for the MCDRAM. The most important modes are: flat mode, where the MCDRAM is available to the user as a separate NUMA node; and cache mode, where the MCDRAM acts as a third level cache.

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

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

2.3.1. **Roofline model**

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

Using the memory bandwidth measured by the stream benchmark and the maximum theoretical floating point performance from Table 2, the roofline model for the KNL node (at Occigen) and the Broadwell node (at MARCONI) is compared in Fig. 3.

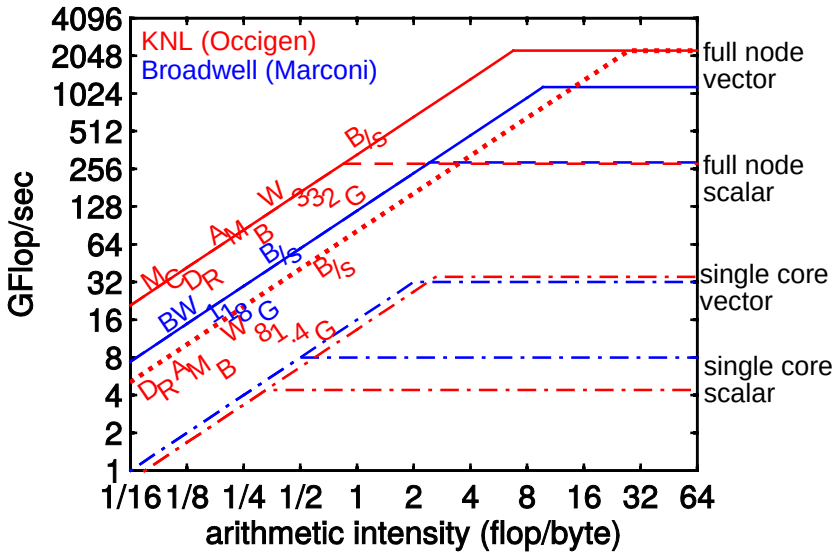


Fig. 3 Roofline model: comparison of a KNL node (red) and a Broadwell node (blue).

The dash-dotted lines compare the single core performance of KNL and Broadwell. The single core scalar performance of KNL is slower because of the lower core frequency, but the wider vector registers make the KNL cores competitive with the Broadwell cores when vectorization is fully exploited. The solid lines give the maximum performance as a function of the arithmetic intensity. There is a factor of 2.8 potential gain in memory speed, and a factor of 2 for compute bound applications. For memory bound applications the challenge is to make efficient use of the MCDRAM which has a limited size of 16 GB. Otherwise the dotted line will be the memory roofline. The hope is that the cache mode can help utilizing this memory without any modifications on the user side.

The dashed lines denote the peak performance of the node without vectorization. This is similar for both architectures. The dashed red roofline intersects the memory roofline at arithmetic intensity of 0.85 for the KNL node. The vectorization does not matter for computational kernels below this arithmetic intensity; such kernels are memory bound even in scalar mode. On the KNL, vectorization becomes important for kernels that have O(1) or higher arithmetic intensity.

To summarize, the KNL node has a higher peak performance, and (using MCDRAM) higher memory bandwidth than the A1 (Broadwell) node on MARCONI. Lack of vectorization, or inefficient use of the MCDRAM might prevent reaching the peak performance, but the roofline model suggest that the KNL and the Broadwell nodes should still have comparable performance in such case.

2.4. *OpenMP benchmark*

The EPCC OpenMP Microbenchmark Suite (Bull, 1999) was used to measure the overhead time of OpenMP constructs on the KNL architecture. Fig. 4 shows how the overhead depends on the number of threads. The overhead on the KNL node at Occigen is compared to the overhead on different Xeon processor architectures (MARCONI A1: Broadwell, HELIOS: Sandy Bridge, HELIOS SOL: Haswell).

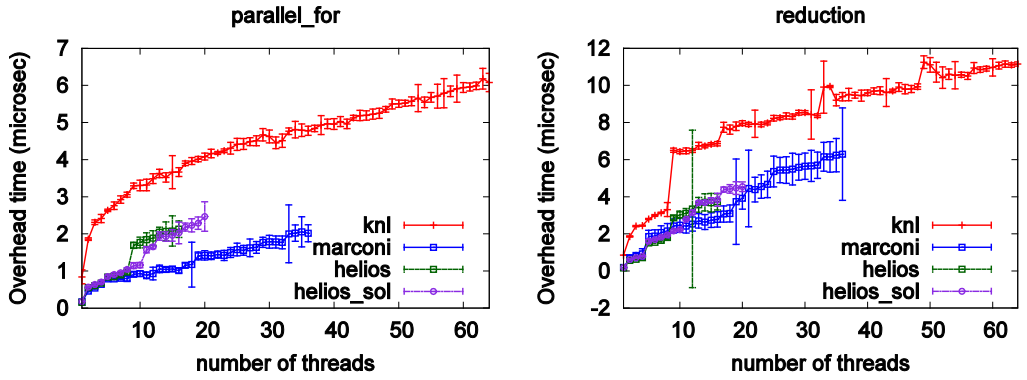


Fig. 4 Overhead time of OpenMP constructs: parallel for (left) and reduction of a single scalar (right). The overhead is measured on the KNL test node at Occigen, on the MARCONI A1 (Broadwell) node, on the regular Sandy Bridge node on HELIOS, and on a Haswell node at the HELIOS Sol partition.

Because of the lower clock frequency, the overhead on KNL is generally larger than on a regular Xeon processor. In case of the `PARALLEL FOR` OpenMP construct, the overhead is usually 2–3 μs larger than on MARCONI A1, if the same number of threads are used. Using all cores of the node, the overhead becomes 3x larger. For the reduction (of a single scalar variable), the difference between the overhead times on different architectures is less pronounced.

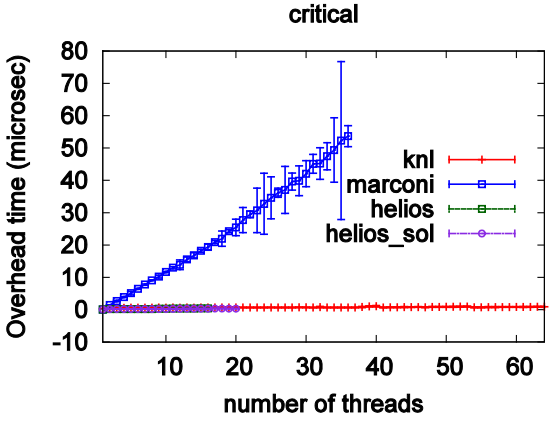


Fig. 5 The overhead time of the OpenMP `CRITICAL` directive. The overhead on the MARCONI Broadwell node is 50x larger than on other architectures.

The two graphs in Fig. 4 is representative for other OpenMP scheduling constructs: the overhead on KNL has the same order of magnitude as on a regular processor, and it can be a factor of three larger for certain constructs. In general we can say that the KNL architecture performs reasonably well in the OpenMP overhead measurements.

An anomaly was detected for the `CRITICAL` synchronization construct. The overhead time on KNL and on both HELIOS architectures follow the same trend as seen in the previous picture, but on the Broadwell node of MARCONI, the overhead becomes significantly longer (see Fig. 5). The same problem was detected with the

overhead of the `LOCK` and the `ATOMIC` directives. This problem will be further investigated on MARCONI.

2.5. *Gysela KNL performance*

The first test of the Gysela code on KNL was performed by G. Latu. He provided a test case that can be used for single node performance measurements. This test case fits into the MCDRAM, and the flat memory mode with `numactl` binding to the MCDRAM provided the fastest execution time on KNL. The same test case was used to measure the performance on a Broadwell node on MARCONI. In both cases 4 MPI tasks were used, and the number of OpenMP threads was selected to utilize all cores in the node (9 threads/core in MARCONI, 16 threads/core in Occigen KNL). The time for the computation on KNL was 47.7 sec, which is 32% slower than on the Broadwell node of MARCONI (37.9 sec).

A detailed profiling was performed using the VTune tool. Fig. 6 shows the relative execution time of the most time consuming functions. These functions take around 40% of the total CPU time. The CPU time is normalized to the total CPU time (summed over all threads) individually for both architectures.

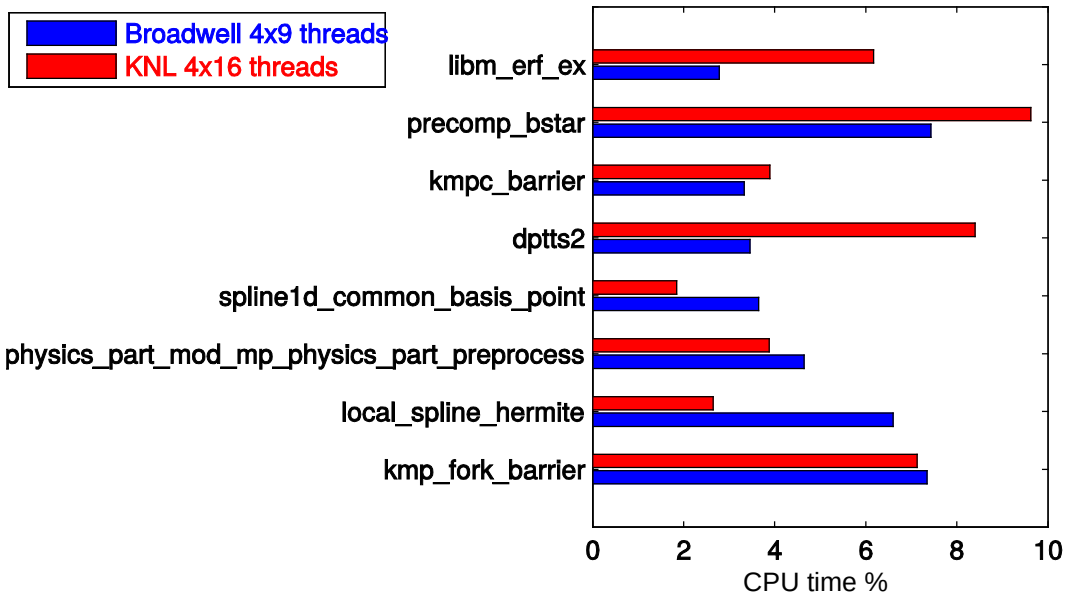


Fig. 6 Execution time of different subroutines in Gysela, normalized to the total CPU time.

On both architectures it is the same set of subroutines that take most of the execution time, but the relative share of individual subroutines are different. On the KNL architecture, the DPTTS2 Lapack subroutine takes a significantly larger share of the execution time than on the Broadwell architecture. This subroutine solves a tridiagonal system. Gysela does not use any libraries to access this function, it includes the source code of it. An obvious question is whether the MKL version of the subroutine would deliver any performance improvement. The code was recompiled in a way to use the MKL version of DPTTS2 and its driver subroutine, but it did not lead to any improvements.

Another important difference is in the execution time of the error function (`libm_erf_ex`). This function is linked when the DERF intrinsic Fortran function is used, and it seems to be slower on the KNL architecture. The MKL library offers a vectorized version of this function, which should be more efficient for vectors over 40 elements. Work has started to modify the code in order to use the vectorized subroutine.

2.6. *Summary*

The roofline model was used to draw a general comparison between the new Knights Landing architecture and the existing Broadwell nodes of MARCONI. The stream benchmark was used to measure the memory bandwidth of a KNL test node at the Occigen supercomputer. Efficient usage of the high bandwidth memory integrated into the KNL processor will probably be an important step in achieving good performance on the KNL processor.

A set of OpenMP benchmarks was performed to test the overhead of different OpenMP constructs. The overhead time increases by a factor of 2–3 on the KNL because of the reduced clock frequency and the larger number of threads.

The execution time of the Gysela code was measured on a KNL node (at Occigen) and the results were compared to a Broadwell node of MARCONI. The wall-clock time for the numerical calculation is 32% longer on KNL for the studied test case. The VTune profiling tool was used to identify which subroutines takes most of the execution time, and work has started to improve those subroutines that give relatively long execution times on the KNL architecture.

2.7. *Bibliography*

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

McCalpin, J. D. (1995). Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE TCCA Newsletter, December.

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 52(4), 65-72.

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

3.1. *Introduction*

The Scrape-Off Layer (SOL) is directly related to the heat exhaust in tokamaks, therefore understanding the physics in this layer is crucial for improving the performance of present and future fusion devices. The SOLPS code package is widely used to simulate SOL plasmas. Numerical modelling plays an important role in understanding the basic physical concepts, interpreting results from experiments and making predictions for future experiments. The SOLPSOPT project aims to optimize the SOLPS package. This project was carried out in collaboration with Lorenz Hüdepohl from the Max Planck Computing and Data Facility (MPCDF).

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

Last year the MPI version of Eirene was tested, and it was found that the parallel code gives slightly incorrect results (Fehér, 2015). The first part of this report discusses the tests that were implemented to find the MPI error in Eirene, and describes how the error was corrected. Afterwards, the performance improvements in Eirene are discussed. Two parallelization strategies (APCAS, Balanced) have been added to Eirene, these provide improved parallel performance for several test cases. The second part of the report describes and compares the now available three strategies[1]. Finally, the coupling between the B2 and Eirene codes is discussed.

Several test cases were used to test the correctness and performance of the SOLPS code. An overview of these test cases is given in Section 3.5.

3.2. *MPI Error in Eirene*

3.2.1. **Reproducibility**

To be able to compare the results from two Eirene simulations, one has to ensure that exactly the same random numbers were used. Otherwise the statistical noise makes it very difficult to compare the results. For debugging purposes, the random number generator was reinitialized for each particle with a seed equal to the global particle index. This guarantees that the parallel and the sequential simulations use identical random numbers.

A test case is set up with only one stratum, and a reduced number of particles (2000 particles in stratum one, zero in other strata). When we compare the results from the serial and parallel simulations, we see large errors in the electron and ion heat flux as shown in Table 3. We can conclude that the MPI implementation of Eirene is not correct. Other users of Eirene have also reported that they see wrong results with the parallel version of the code.

To ensure that indeed an identical random number sequence was used during debugging, every time a random number is generated it is printed to the standard output together with the global index of the particle that is currently processed. The output from different MPI processes are merged and sorted by the particle index using stable sorting (that preserves the order of elements if the particle index is equal). This way a single file can be created to store the stream of random numbers

[1] The comparison in this report is only valid for test cases where we want to calculate a fixed number of particles in each stratum. We do not discuss here the other operation mode of Eirene, when the available CPU time is defined for each MPI tasks, and we calculate as many particles as possible within this time.

that is used for the calculation. We can compare these files to check if the random streams agree between two calculations.

1 MPI task	2 MPI task
NEUTRAL PLASMA INTERACTION:	NEUTRAL PLASMA INTERACTION:
SSEI,SSEE 4.5541E+04 -1.3184E+05	SSEI,SSEE 4.0076E+04 -1.3409E+05
SSNI(IF = 1) 4762.17908751428	SNI(IF = 1) 4757.30883715227

Table 3 Output of Eirene in serial and parallel mode, the differences are highlighted in red. The ion and electron heat flux (SSEI and SSEE respectively) are calculated by Eirene and returned to B2.

To track down the problem in parallel mode, a unit test framework was created. In the following sections we describe how this framework was used to test Eirene.

3.2.2. **Unit test framework**

A subroutine might depend on module data or internal variables that have to be initialized before we call the subroutine. In other words, a certain runtime environment has to be set up for such subroutines, before we can call them. When we want to write a unit test for a legacy code, then a key issue is how to restore the runtime environment for the different program units before we test them.

A new unit test framework has been developed, to address this problem and automatically generate unit tests for Eirene. To be able to save and restore the runtime environment, the framework generates procedures that save module data, variables with SAVE attribute, and subroutine arguments. It can be used to save data for individual subroutines, as well as to create global check points (that save all module data). Call statements are automatically inserted into the legacy code to call these data saving subroutines. When we execute the modified code, then the unit test data is saved to file. This file is used to restore the program status when we run the separate unit tests. An illustration of this procedure is shown in Fig. 7.

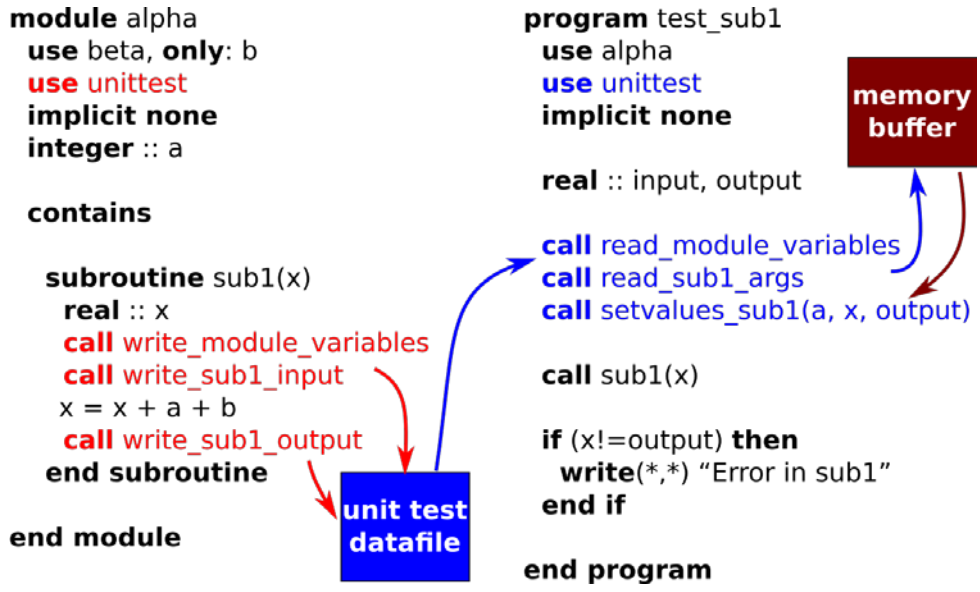


Fig. 7 Unit test example. The left hand side shows a module with a subroutine *sub1*. The right hand side is the unit test program for *sub1*. The unit test generator inserts the write statements (red) into *sub1*, to create a data file that is used in the unit test program to restore the program state before calling *sub1*.

In the test programs, the unit test data is loaded first into a temporary buffer in memory, to avoid additional file IO when we compare the results to the saved data. This also allows scaling studies without file IO overhead, but the price for this is a

factor of three higher memory consumption (apart from the original variables, we have two additional buffers to save input and output data).

To build the unit test framework, the ANTLR parser generator tool (Parr & Harwell, 2016) was used to create a Fortran parser, and using this parser an application was written that can collect information about the scope hierarchy. This information is used to generate the subroutines that save all the variables that are relevant to execute a certain procedure. A detailed description of this framework will be presented in a technical report. The new unit test framework allows us to handle derived types, allocatable and pointer variables, and it is also possible to save private variables. All these features are necessary to save and restore the program status in Eirene. The test generator should in principle work for any Fortran code, and it is expected to be used in future HLST projects.

3.2.3. **Check points for debugging**

The unit test framework was used to instrument the Eirene code. At the current stage, we do not generate separate test programs for all the subroutines; we generate only global check points for the `MCARLO` subroutine, which implements the main loops of Eirene. A simplified outline of this subroutine is presented Fig. 8.

Eirene uses the stratified sampling technique, where the particles are grouped into different strata. When the code is executed in parallel, then the loop over the strata and the loop over the particles can be executed concurrently. After following the particles, the results are summed up over the MPI processes (lines 6 and 8). For debugging, the program status is dumped into file at three check points. The blue arrows on Fig. 8 mark the position where the data was saved for debugging.

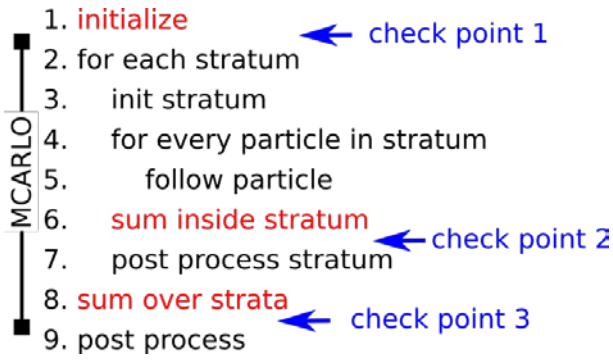


Fig. 8 Outline of Eirene algorithm, steps 2–8 are implemented in the `MCARLO` subroutine. Loops in the second and fourth lines can be parallel. The steps in the first, sixth and eight lines involve MPI communication. Debugging check points are indicated by the blue arrows.

`MCARLO` and the subroutines called from `MCARLO` use global data stored in more than forty modules. Every module was equipped with extra procedures that can save module data to file, load it, or compare to values that were saved earlier. Fig. 9 gives an overview of the amount of data that was saved from each module. The next sections will discuss how we can use this information to find the error in the parallel version of Eirene.

3.2.4. **Input data**

To find the bug in the parallel code we compare the check point data between serial and parallel execution. In parallel mode rank 0 reads the input data from file and broadcasts it to other MPI tasks. After initialization, at check point 1, all MPI tasks should have the same input data. However it does not mean that all program variables should have the same value.

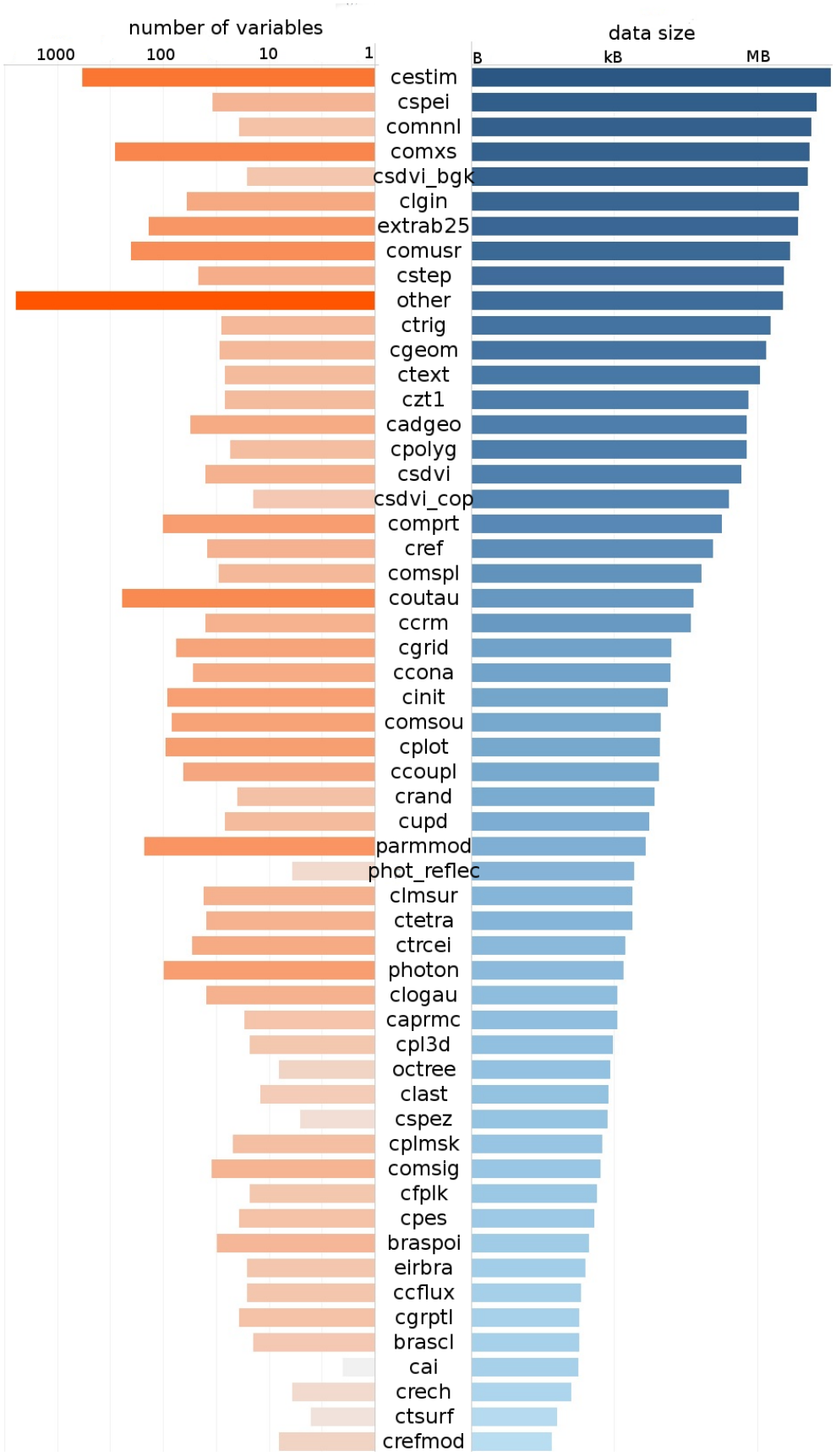


Fig. 9 Saved module data for a global check point in Eirene. For each module the bars show the number of variables and the data size that was saved. Apart from module variables, we save every other variables that have the SAVE attribute. For the AUG1 test case at each check point 116 MiB data is saved to store the values of around 2800 variables. This is comparable to the total memory footprint to the program. Each MPI task saves its data into a separate file.

Comparing the check point files of task 0 and task 1 showed that around 400 variables in 35 modules (or subroutines) had different values. Some of the differences were due to temporary variables used only during reading the input data file. A large number of differences were caused by variables that are defined later during the program execution. To find out whether any of the remaining differences are significant, we used the strategy depicted in Fig. 10.

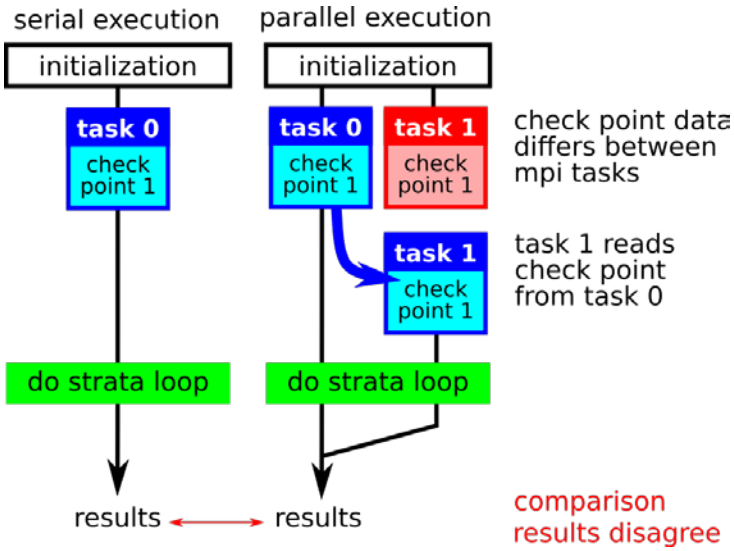


Fig. 10 To test the input data distribution, we compare a serial and a parallel (2 task) simulation. The check points are used to compare the program status and to reset the differing variables of task 1.

At check point 1 the program state of the serial program agrees with the state of task 0 of the parallel program, but task 1 has differences, as discussed above. Using the testing framework, the differing variables at task 1 are set to the same value as task 0. Then we continued the program execution from check point 1. The final result did not change; therefore we can conclude that the differences were insignificant. This means that the input data is properly distributed among the MPI tasks.

3.2.5. **Summing inside a stratum**

After investigating the distribution of the input data, we checked the MPI reduce operations that are used while summing up the data inside a stratum (step six in Fig. 8). Using only one stratum, the results between serial and parallel code versions should be the same at check point 2.

All arrays that are part of any MPI reduce operations were checked before the check point, and they had correct values in parallel mode. The remaining question is: are there any variables that should be summed up over the MPI processes, but do not have the corresponding MPI reduce operation? Answering this question is more complicated, since there are many variables in scope, a lot of them have different values between serial and parallel execution, but not all of them are significant for the results.

To focus the search, we consider SSEI, one of the variables that had a wrong final value (see Table 3). We map how the value of SSEI depends on other variables (Fig. 11), and we compare these variables at check point 2. Surprisingly, all of these variables agree between the parallel and serial runs at the check point. This means, that the parallel code is correct at check point 2, but later it becomes incorrect, even though there are no more MPI communication required to calculate the results.

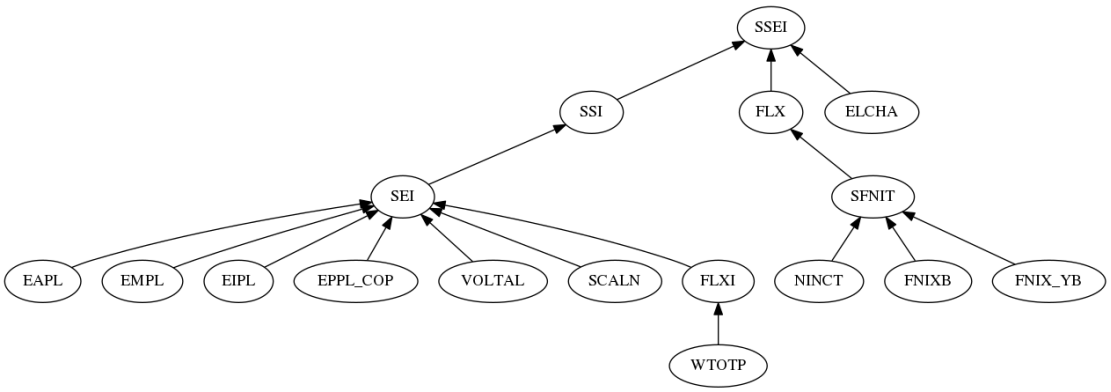


Fig. 11 The value of the SSEI variable depends on many other variables. The nodes of the graph are variables in the Eirene code, an arrow between A → B means that the value of A was used to calculate the value of B.

3.2.6. **Error in the interface subroutine `IF3C0P`**

The SSEI scalar variable stores basically a sum of the SEI array, multiplied with some scaling factors. SEI is calculated in the `IF3C0P` subroutine. This subroutine is part of the interface for coupling B2 and Eirene. Note that while we are using a standalone version of Eirene for the tests, the coupling routines are called similarly as they would be called in the coupled B2-Eirene code package.

A check point was added to `IF3C0P`, and it revealed a logical error in the program execution. Fig. 12 shows how the `IF3C0P` subroutine is called from Eirene, and how the stratum data is handled inside `IF3C0P`.

Let us consider the serial execution of Eirene (first column in Fig. 12). The `IF3C0P` subroutine is called in the main loop over the strata, and inside `IF3C0P` the atomic heat source contributions from the current stratum are added to the SEI array. Several other quantities (that will be passed to B2) are also calculated. To calculate these contributions, helper arrays need to be initialized, and an input data file is read.

In parallel mode, only rank 0 has the helper arrays initialized, therefore all the contributions to SEI are calculated by rank 0. But in parallel mode rank 0 is not necessarily calculating every stratum, therefore the stratum data needs to be communicated. Additionally, to save memory, only the data for the current stratum is kept in memory, and it is overwritten at every iteration of the strata loop. Therefore, in parallel mode, we have two different methods for calling `IF3C0P`, depending on whether the stratum data is available in memory or not.

If the number of strata is smaller than the number of tasks (right column in Fig. 12), each stratum will have at least one MPI task associated. One of them will be the stratum leader, who collects all the results inside the stratum. An MPI tasks calculates exactly one stratum, therefore after the main strata loop, the data for every stratum is in the memory of one of the tasks. `IF3C0P` is called for all strata in the final post processing step, and the inside `IF3C0P` the stratum data is sent to task 0 using MPI send and receive calls.

A problem arises if the number of tasks is smaller than the number of strata (middle column in Fig. 12). In that case, at least one of the MPI tasks is calculating more than one stratum which overwrites the values of the previous stratum. As a quick and dirty way to avoid this, the stratum data is dumped to file, and essentially this file is used to communicate the data between the stratum leaders and task 0. Several problems arise with this method:

- Using a file to communicate between MPI tasks is inefficient.
- The file IO that is used is not MPI safe.
- Synchronization would be needed before task 0 reads the data from the file, which was not implemented properly.

- Incorrect variables were used to decide whether to use the file data transfer or MPI send operations (column two or three).

Because of this last point, the debugging test case (1 stratum and 2 MPI tasks) used file based communication instead of the MPI communication. This error actually helped to discover the bug.

Serial simulation	Parallel simulation	
	$N_{task} <= N_{strata}$	$N_{task} > N_{strata}$
2. for each stratum . . . 7. post process stratum **call IF3C0P(stratum)** 8. sum over strata 9. post process	2. for each stratum . . . 7. post process stratum **write stratum data file** 8. sum over strata 9. post process **call IF3C0P(all strata)**	2. for each stratum . . . 7. post process stratum 8. sum over strata 9. post process **call IF3C0P(all strata)**
subroutine IF3C0P(stratum) ! process only 1 stratum ! **use data from memory** calc SEI for stratum	subroutine IF3C0P(strata) **for every strata** if (rank == 0) **read stratum data file** calc SEI for stratum	subroutine IF3C0P(strata) **for every strata** **MPI collect data rank 0** if (rank==0) calc SEI for stratum

Fig. 12 The IF3C0P interface routine between B2 and Eirene. The three columns show three operation modes of IF3C0P. The middle row shows how IF3C0P is called within or after the main loop of Eirene, the last row shows the IF3C0P subroutine. Depending on the number of MPI tasks, different methods are used to access the stratum data that is needed to calculate the fluxes for B2. The differences are highlighted in bold.

3.2.7. **Correct interface to B2**

To correct the problem is conceptually trivial: one should let the stratum leader (instead of task 0) calculate the heat flux and other variables for B2. This can be done within the main strata loop. The result of IF3C0P will be distributed among the stratum leaders, and it can be summed up in the post processing step after the strata loop. Fig. 13 shows the conceptual steps. The problem was discussed with one of the developers of Eirene, P. Börner, and she encouraged this solution.

The only difficulty is that the initialization of the data structures used in IF3C0P is cumbersome, and MPI reduce or send/receive operations had to be implemented for around 50 arrays.

It would be possible to implement this solution only for the $N_{task} < N_{strata}$ case, and let the other two branches in Fig. 12 handle the other cases. But since this solution works in general for any number of tasks, it was decided to keep only this solution, so that the resulting code is easier to maintain.

The corrected code was tested for variations of the AUG test case using 1, 2, and 6 strata and 1 to 16 MPI tasks. In all cases, all fluxes printed to the standard output agree up to machine precision, and the fort.44 file, which stores all the data that is calculated for B2 is identical to the serial result.

Serial and parallel simulations
2. for each stratum . . . 7. post process stratum **call IF3COP(stratum)** 8. sum over strata 9. post process **Sum IF3COP over MPI tasks**
subroutine IF3COP(stratum) ! process only 1 stratum ! **use data from memory** **Init data structures** calc SEI for stratum

Fig. 13 Corrected interface routine, every stratum leader calls IF3COP, the results are summed up after the strata loop.

3.3. *MPI performance improvements*

In this section, we will use the abbreviation PE (processing element) to refer to MPI tasks. In all our tests, the MPI tasks are pinned to separate CPU cores.

3.3.1. **Original parallelization**

Let us recall the basic algorithm of Eirene represented in Fig. 8. In parallel mode, we need to distribute the work of the strata and the particle loops (2nd and 4th lines) among the PEs. Every PE processes one ore more strata and the number of particles within a stratum are divided among the PEs. For each stratum, the post processing step (7th line) is executed by only one PE, the stratum leader. Different strata can have different leaders. The steps highlighted in red include MPI communication.

A parallelization strategy can be represented in a work distribution table: *N(i, k),* which describes how many particles should be processed by the *i*-th PE from stratum *k*. The table also highlights who is the leader of the strata. An example of the work distribution table is shown in Table 4.

Original

PE \ Strat	1	2	3	4	5	6
0	125000#					
1	125000					
2		250000#				
3			20000#			
4				20000#		
5					20000#	
6						23334#
7						23333
8						23333
sum	250000	250000	20000	20000	20000	70000

Table 4 Work distribution tables for the original parallelization strategy (6 strata and 9 MPI tasks). For each MPI task (PE, rows of the table) we list the number of particles that will be processed for each stratum. Zero values are not shown; the stratum leader is marked with #.

The original parallelization strategy of Eirene assigns PEs to each stratum. The strata with larger workload get more PEs. The time needed to calculate each stratum is

measured, and the number of PEs per strata is optimized accordingly. The PEs that process a strata divide the work evenly among each other. Blocking reduction operations are used to sum up the results inside the stratum.

The original strategy tries to assign an optimal number of PEs to each stratum, and this ensures that the communication within the stratum is kept minimal. It depends on the number of strata and the number of PEs, how evenly we can distribute the work. The strategy is efficient when a large number of PEs are used, but it usually leads to load imbalances with a low number of PEs (comparable to the number of strata).

As discussed in the previous report, a bug was discovered and fixed in the `IF3COP` subroutine. Small parts of the code were also refactored during the bug fix. This led to a performance increase as we can see in Fig. 14. In the original version, the communicators that are used in steps 6 and 8 (in Fig. 8) are calculated on the fly in the strata loop of Eirene. The communicators are created using `MPI_COMM_SPLIT` which is a collective call. This way a global synchronization point is introduced in the strata loop. This has been changed in the `bugfix/IF3COP` branch of Eirene, where the communicators are created in advance before the strata loop. This way the parallel execution time of the `MCARLO` subroutine is 29% to 68% shorter for certain test cases (Fig. 14 right).

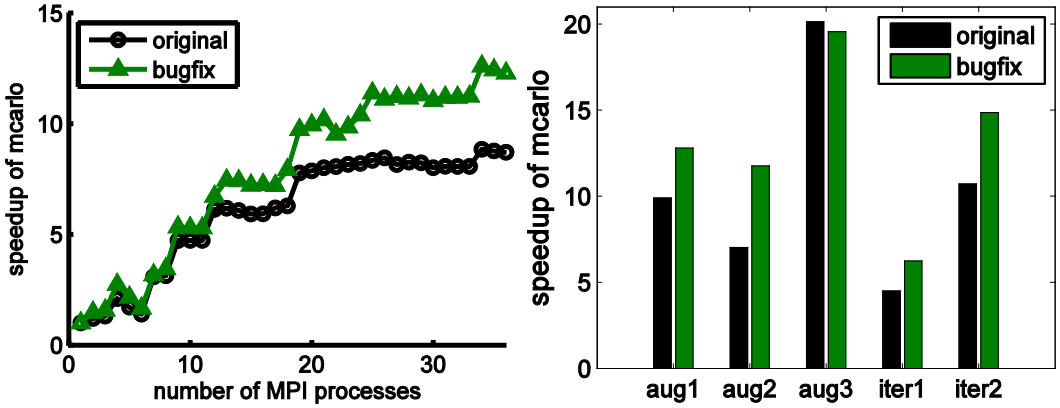


Fig. 14 Speedup of the `MCARLO` subroutine using the original parallelization strategy. Left: speedup for the `ITER2` test case, right: speedup of different test cases using 36 MPI tasks. Black color shows the performance before the bug fix, green shows the performance afterwards. A description of the test cases is given in Section 3.5.

3.3.2. **Additional parallelization strategies**

To fix the problem of load imbalance, two parallelization strategies were implemented in Eirene. In previous reports (Fehér, 2015) a simple parallelization strategy (APCAS) was already discussed. In addition, a new strategy was implemented which is called balanced strategy. A short description of the new strategies:

1. **APCAS**: All PE Calculates All Strata. The work is distributed evenly between the PEs, blocking reduction operations are used.
2. **Balanced**: We use non-blocking reduction operation while summing up the results inside a stratum. This way we do not have to divide the work evenly within a stratum between the PEs. We can divide the work freely between the PEs. We use this freedom to optimize the work distribution based on measurements of the processing time and overhead times.

Examples of these strategies are show in Table 5. The APCAS strategy aims to achieve load balance by assigning the same amount of work for each PE. We should note that there is a post processing step for each stratum, executed only by the stratum leaders. If the post processing time is negligible, then APCAS guaranties load balance. The disadvantage is that since all PEs calculate all strata we have much more MPI communication than with the other strategies, which leads to inferior performance if a large number of MPI tasks is used.

When the balanced strategy is used, we initialize the work distribution table with the APCAS method and measure the performance of each PE on each stratum. In the subsequent iteration we optimize the work distribution table based on the measurements. As we can see in Table 5, a PE can process more than one stratum, and the work within a stratum is not distributed evenly. In balanced mode, Eirene also prints a work distribution table which shows the estimated working time (see Table 6). We can see that the estimated total working time (last column) shows only a small variation among the PEs.

APCAS

PE \ Strat	1	2	3	4	5	6
0	27777#	27777#	2222#	2222#	2222#	7777#
1	27777	27777	2222	2222	2222	7777
2	27777	27777	2222	2222	2222	7777
3	27777	27777	2222	2222	2222	7777
4	27777	27777	2222	2222	2222	7777
5	27777	27777	2222	2222	2222	7777
6	27777	27777	2222	2222	2222	7777
7	27777	27777	2222	2222	2222	7777
8	27784	27784	2224	2224	2224	7784
sum	250000	250000	20000	20000	20000	70000

Balanced

PE \ Strat	1	2	3	4	5	6
0	75402#					
1		137825#				
2			20000#			15794
3				20000#		14647
4		112175			20000#	
5						20687#
6	84353					
7	74572					
8	15673					18872
sum	250000	250000	20000	20000	20000	70000

Table 5 Work distribution tables for different parallelization strategies (6 strata and 9 MPI tasks). For each MPI task (PE, rows of the table) we list the number of particles that will be processed for each stratum. Zero values are not shown; the stratum leader is marked with #.

PE \ Strat	1	2	3	4	5	6	overhead	sum
0	4.293#						0.067	4.359
1		4.293#					0.081	4.374
2			1.005#			3.206	0.179	4.391
3				1.129#		3.033	0.141	4.303
4		3.589			0.581#		0.156	4.327
5						4.293#	0.076	4.369
6	4.293						0.012	4.305
7	4.293						0.016	4.308
8	0.845					3.650	0.022	4.517

Estimated working time / PE 4.36 +/ 0.06 sec

Table 6 Work distribution table with estimated working times (in seconds) for the balanced strategy (corresponding to the values in Table 5). The overhead is the time to sum results inside a stratum plus the post processing overhead.

3.3.3. **Balanced strategy implementation**

The key ingredient in the balanced strategy is the modification of the `CALSTR` subroutine that is responsible for summing the results inside a stratum. Originally `CALSTR` uses `MPI_REDUCE` calls, which block until the reduction operation is completed. This means that all PEs that process a strata have to wait for each other

while summing the results (step 6 in Fig. 8). Therefore in the original strategy, it would be counterproductive to assign different amounts of work to different PEs.

In the balanced strategy the reduction operations are replaced with the non-blocking `MPI_IREDUCE` call (Fig. 15 right column). We collect all data to a buffer that is used for reduction. This way, we can reuse the original variables, and continue processing the next stratum (Fig. 15 left column). The stratum leader is waiting for the reduction operation to finish, because it has to post process the received data.

We should note that the reduction operation is not progressing automatically in the background. In the particle loop, we are calling the `MPI_TEST` function to check the status of the reduction, and this ensures message progression. Performing 100 tests per stratum seemed to be sufficient and caused only a small overhead. Alternatively, one could set the `MPICH_ASYNC_PROGRESS=1` environment variable, in this case the MPI library uses threads in the background to progress the message automatically (without the need to call `MPI_TEST` manually). This method was usually 40% slower.

The balanced strategy needs an MPI library that implements the 3.0 standard, and some extra memory for the reduction buffers. In case these are not available, the code automatically falls back to the APCAS strategy.

Eirene algorithm	Sum inside stratum (CALSTR)
1. Initialize 2. For each stratum 3. Init stratum 4. For every particle in stratum 5. Follow particle 6. If (time_to_progress) 7. MPI_TEST(request) 8. Sum inside stratum 9. Post process stratum (leader only) 10. MPI_WAIT(request) 11. Sum over strata	1. MPI_WAIT(request) ! ensure that ! buffer is ready 2. Pack data into buffer 3. MPI_IREDUCE(buffer, request) 4. If (I_am_leader) 5. MPI_WAIT(request) 6. Unpack buff

Fig. 15 Modified Eirene algorithm for the balanced parallelization strategy, new steps are highlighted with blue.

A second essential ingredient for the balanced strategy is optimizing the work distribution table. Let $N(i,k)$ denote the number of particles that PE i follows from stratum k. We measure the execution time $t_x(i,k)$ of the particle loop (4th and 5th step in Fig. 15 left column) for each stratum and PE, and define T as the throughput

$$T(i,k) = \frac{N(i,k)}{t_x(i,k)},$$

which tells us how many particles can be processed per second. We define the ideal working time by dividing the total work with the aggregated throughput

$$t_{\text{ideal}} = \sum_{k=1}^{N_{\text{strata}}} \frac{N_{\text{particles}}(k)}{\sum_{i=1}^{N_{\text{PE}}} T(i,k)}.$$

Here the number PEs and strata are denoted by N_{PE} and N_{strata} respectively. Without any communication and post processing overhead, each PE would calculate for t_{ideal} seconds. Using the algorithm in Fig. 16, we aim to fill the work distribution table so that the total execution time is close to t_{ideal}.

0. Estimate the ideal working time
1. Select leaders round robin
2. Assign the work to stratum leaders
3. do k=1,N_{strata}
4. while work is left in stratum(k)
5. i ← select a PE with the smallest working time
6. assign particles for PE(i) from stratum(k)

Fig. 16 Optimization of the work distribution table for the balanced strategy.

Let us consider how this works for the example given in Table 5. Step one selects the stratum leaders by assigning $\mathrm{leader}(k) = PE\ (k - 1 \bmod N_{\mathrm{PE}})$. In our example it simply puts # at the diagonal of table. The stratum leader will always do work on its stratum. In step two, we assign as many particles as possible to the leaders:

$$N(\mathrm{leader}(k), k) = \min\left(t_{\mathrm{ideal}} \times T(\mathrm{leader}(k), k), N_{\mathrm{particles}}(k)\right).$$

At this stage the diagonal of Table 5 is filled. We can see that all the work is assigned for strata 3, 4, and 5. For the other strata, the leaders calculate as many particles as possible in t_{ideal} (see also Table 6, which shows that $t_{\mathrm{ideal}} = 4.293\ s$ work is assigned to the leaders of strata 1, 2, and 6).

The remaining particles are distributed in the loop that starts at step 3 in Fig. 16. We keep track of how much working time is assigned to each PE, and denote it by *t(i)*. We aim to finish processing stratum k by $t_{\mathrm{target}}(k) = t(\mathrm{leader}(k))$, so that the leader does not have to wait for the other PEs. To ensure that the work within a stratum is not fragmented between too many PEs, we select the PE that has the smallest *t(i)* in step 5 (i.e. the PE that has the largest free capacity to do work), and assign as much work as possible to this PE

$$N(i,k) \mathrel{+}= \min\left((t_{\mathrm{target}}(k) - t(i)) \times T(i,k), N_{\mathrm{particles\ remaining}}(k)\right).$$

The heuristic outlined here is a simple and fast method to find a close to optimal solution. It is designed to minimize the total working time as well as the communication within the strata. While building up the work distribution table, we also include estimates for the time of the `CALSTR` subroutine (communication within the stratum, 8th step in Fig. 15) and post processing (9th step in Fig. 15). At every iteration we refine the measurements and recalculate the work distribution table accordingly.

3.3.4. **Performance**

To compare the different parallelization strategies, we used the `AUG3` and `ITER2` test cases which are standalone (Eirene without B2) test cases. The parameters for the test cases are listed in Section 3.5. For both test cases the number of iterations was increased to ten. The code was compiled with the Intel Fortran compiler v16 using `-O3` optimization. The tests were run on the MARCONI supercomputer, where each node has 2x18 cores.

Keeping the problem size constant, the execution time was measured using different numbers of MPI tasks. We considered the execution time of the `MCARLO` subroutine only, which is the main subroutine that loops over all the strata and follows the particles. We calculated the speedup of this subroutine. (Note that Eirene as a whole has lower speedup for these test cases, because during each iteration Eirene broadcasts all the input data, which gives a large overhead).

The speedup for the `AUG1` test case is show in Fig. 17. The original strategy has a load imbalance when the number of MPI tasks is comparable to the number of strata. The APCAS and the balanced strategies have similar performance as shown in the left subfigure, except for the range between 21 and 29 tasks, where the performance of the balanced strategy has a drop. This is reproducible, even with different versions of the Intel MPI library. The other strategies also have drops in performance for

certain numbers of MPI tasks (original for 21 and 25–26 tasks, the APCAS strategy shows a similar problem for the ITER2 test case). Such anomalies were also detected while executing on the HELIOS supercomputer with Intel MPI, but it disappeared when Bullx MPI was used. The problem with Intel MPI will be further investigated.

The black dashed curve shows the theoretical estimate for the ideal speedup based on Amdahl's law. The right side of Fig. 17 shows the speedup for larger numbers of MPI tasks, on a logarithmic scale. As expected, the performance of the APCAS strategy drops when a large number of tasks are used (communication overhead). The original strategy catches up, and it has similar performance to the balanced strategy at large numbers of MPI tasks.

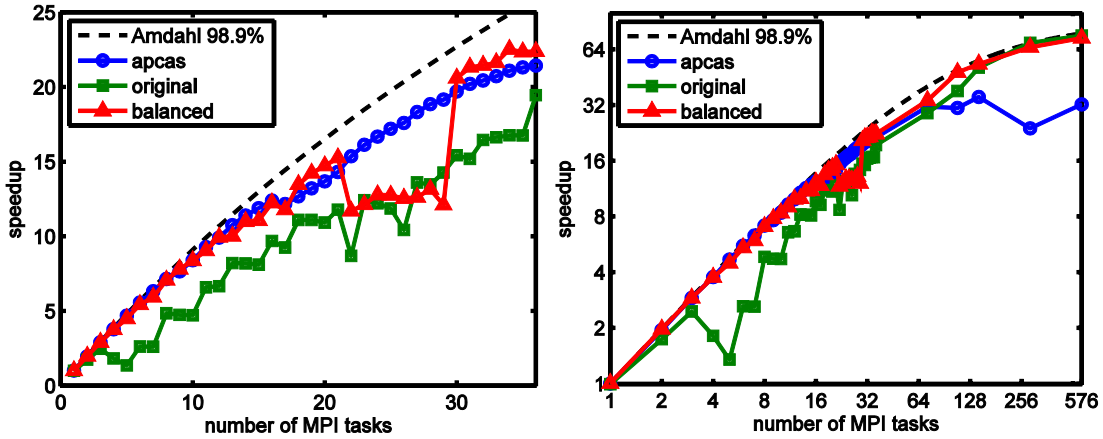


Fig. 17 Speedup of the MCARLO subroutine for the AUG test case.

The speedup for the ITER2 test case is plotted in Fig. 18. The black and green curves are the same that are shown in Fig. 14. For this test case, the post processing overhead for the stratum leaders is significant; therefore the APCAS strategy does not perform well. The balanced strategy is close to optimal. The original strategy start to be efficient above 9 tasks and it can even be slightly faster than the balanced strategy.

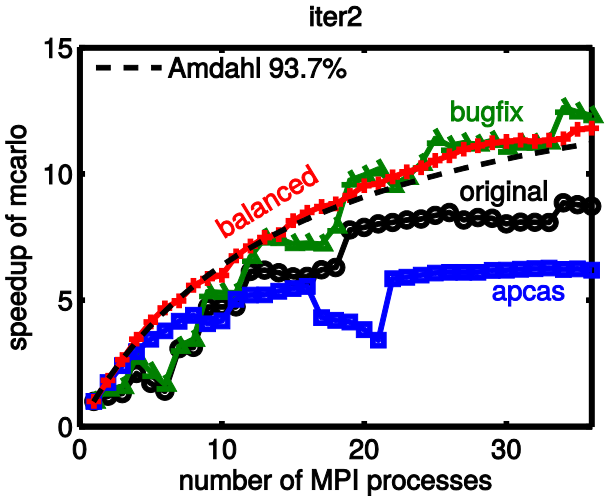


Fig. 18 Speedup of the MCARLO subroutine for the ITER test case.

3.3.5. **Correctness**

All three parallelization strategies were thoroughly tested using the AUG3 and ITER2 test cases (but lower particle numbers and only 4 iterations were used). First, it was confirmed that the serial version gives identical result to ITER-develop.

Second, the debug version of the code (as described in Section 3.2.1) was used to compare serial and parallel simulations. In debug mode, the random seeds are set to

the global particle index; therefore always the same random stream is used even in parallel mode. This allows direct comparison of the serial and parallel result, even if the results are not converged due to the small number of particles.

The MODBGK and MODUSR subroutines were switched off for the parallel comparison; otherwise the small numerical noise coming from the parallel reductions would pollute the plasma parameters for the next iteration and that would make it impossible to compare results with different number of MPI tasks.

The results of the parallel version are in agreement with the serial version. It was checked with different numbers of MPI tasks (1-36) for both test cases and all parallelization strategies. The balanced strategy was tested additionally between 1–144 task with two different optimizations `-O0 –check all` and `–O3`, and found to be in agreement with the serial version.

3.4. *Hybrid B2-Eirene coupling*

In the SOLPS-ITER package, the B2 and Eirene codes are linked together to form a single executable. The B2 code drives the simulation, and it calls Eirene to compute certain source terms. B2 is parallelized using OpenMP and Eirene uses MPI, therefore the coupled B2-Eirene package is a hybrid code. The technical implementation of the coupling is shown in the left side of Fig. 19. We should note that this is an unusual hybrid program: only MPI task 0 is executing the computational part of B2 therefore only MPI task 0 spawns threads. The other MPI tasks wait at a synchronization point until Eirene is called. The synchronization is done using an MPI broadcast call. Afterwards, task 0 continues the calculation with B2, the other tasks will wait until Eirene is called again. Fortunately, the original implementation of the coupling works fine for the hybrid code.

The right side of Fig. 19 of figure shows how the OpenMP threads and MPI tasks are pinned to CPU cores. The MPI tasks execute as separate processes, and they are explicitly pinned to separate CPU cores (blue boxes in Fig. 19). MPI task 0 spawns threads, and these are also pinned to separate CPU cores (red boxes). This pinning is implemented by exporting the following environment variables:

```
export I_MPI_PIN_PROCESSOR_LIST=0-n
export KMP_AFFINITY=norespect,compact.
```

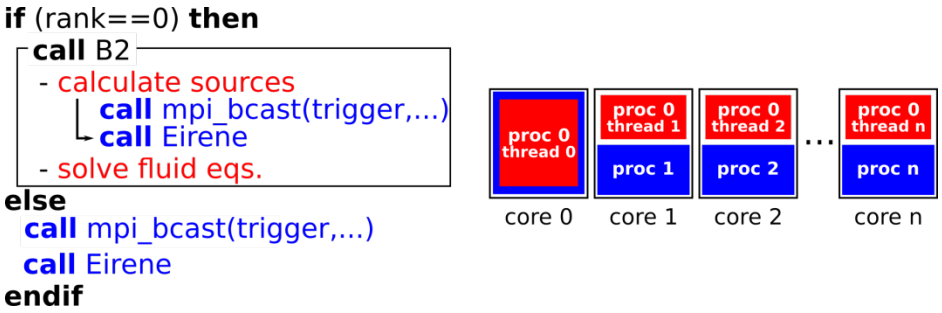


Fig. 19 Coupling between the OpenMP B2 and the MPI Eirene codes. Left: only MPI task 0 is executing the computational part of B2. OpenMP parts are highlighted with red, MPI parts are shown with blue color. Right: the pinning of MPI processes (blue) and OpenMP tasks (red).

We can see that most of the CPU cores are oversubscribed: core *k* executes code both from MPI process 0 and MPI process *k*. Let us consider how this impacts the performance of the coupled code.

The OpenMP threads have a limited lifetime. Once the parallel region is closed, the threads will spin-wait for a certain amount of time, and go to sleep if there is no more work to do. Spin-waiting consumes resources so the waiting time should be short enough to avoid interference with the execution of Eirene. On the other hand, the waiting time should be long enough to keep a low overhead for the creation parallel regions in B2. The variable `KMP_BLOCKTIME` can be used to set how long the

threads are waiting (default is 200 milliseconds). Once the OpenMP changes are merged into SOLPS-ITER one can make tests to find the optimal value.

Another question is whether the waiting MPI tasks have any influence on the performance of the OpenMP threads. We have created a test program to measure this. To simplify the test, the Eirene code was removed, and B2 was replaced with the triad test from the stream benchmark (McCalpin, 1995). The test code and the results are shown in Fig. 20.

```
stream_modified:

if (rank==0) then
  call triad_benchmark
  call mpi_bcast(trigger...)
else
  call mpi_bcast(trigger)
end
```

```
#execution:
export I_MPI_PIN_PROCESSOR_LIST=0-35
export KMP_AFFINITY=norespect,verbose
export OMP_NUM_THREADS=$CORES
# export I_MPI_WAIT_MODE=1
mpirun -np $CORES ./stream_modified
```

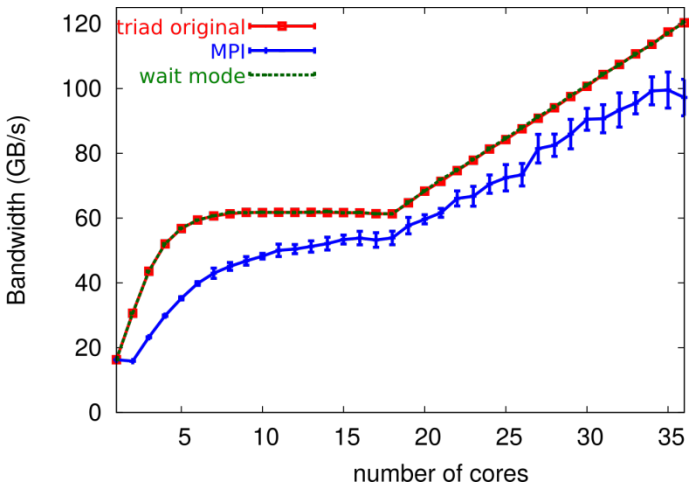


Fig. 20 Testing the performance of the coupling using the stream benchmark. The red curve shows the bandwidth measured by the original stream benchmark. The blue curve is the bandwidth when MPI processes are waiting in the background. Setting I_MPI_WAIT_MODE=1 (green curve) results in the same bandwidth as the original stream benchmark.

It turns out that the MPI processes are also spin-waiting in the background and therefore they steal resources from the OpenMP tasks. The variables `I_MPI_SPIN_COUNT` and `I_MPI_SHM_SPIN_COUNT` can in principle be used to change the spinning time, but in practice they did not change the results. Using Intel MPI, one can fix this problem by enabling the wait mode (using the I_MPI_WAIT_MODE variable). This way the MPI tasks go to sleep state while they are waiting for the message from task 0, therefore they do not use resources, and one can get the same performance as the original test without waiting MPI tasks.

If the MPI library does not support such a waiting mode, then it can be manually implemented as shown in Fig. 21. The waiting MPI tasks periodically check the trigger status, and go to sleep in between. A sleep time of one millisecond or more is enough to keep the MPI overhead negligible. The sleep function is not part of the Fortran standard, but several compilers have implemented such function. In our test the `nanosleep` POSIX function was used with ISO C bindings.

```
if (rank==0) then
  call triad_benchmark
  do k = 1, ntask-1
    call mpi_send(trigger, …)
  end do
else
  call mpi_irecv(trigger, … request, ierr)
  flag = .false.
  do while (.not.flag)
     call mpi_test(request, flag, ..)
     call sleep(SLEEP_TIME)
  end do
endif
```

Fig. 21 Wait mode manually implemented: the waiting MPI tasks use non-blocking receive, and periodically test the status of the receive operation. They sleep between calls to MPI_TEST.

3.5. *Test cases*

Table 7 gives an overview of the main parameters for the test cases used to test Eirene. Test cases AUG1, AUG2, and ITER1 are downloaded from the SOLPS-ITER repository (SOLPS ITER Examples, 2016). The AUG3 test case is the same as AUG1, but ten times as many particles and without time dependent strata. The source of the ITER2 test case is the SVN repository at IPP (SOLPS ITER2 test case, 2016). ITER3 is our target production test case (SOLPS 5.0 98 species test case, 2016), but the input parameters for Eirene are not yet defined for this test, therefore it works only in standalone B2 only mode.

Name	Species	particles	strata	Background species	reactions
AUG1	D	63k	7	5	59
AUG2	D+C+He	67k	9	13	59
AUG3	D	630k	6	5	59
ITER1	D+Ar+He	93k	13	37	34
ITER2	D+C+He	73k	7	25	26
ITER3	D+T+He+Be+Ne+W	?	?	> 98	?

Table 7 Test cases

3.6. *Summary*

The work in 2016 focused on improving the MPI parallelization of Eirene. The parallel code gave slightly incorrect results for certain test cases; this had to be corrected first. A testing framework was constructed to help find the error in the parallelization of Eirene. The framework can automatically generate unit tests for legacy Fortran codes. Using these tests, the error was found in the coupling subroutines between B2 and Eirene. The problem was solved, and the code calculates now correct results in parallel mode. The bug fix has been implemented in the SOLPS-ITER version of Eirene.

The original parallelization strategy was improved by calculating the communicators in advance before the Monte-Carlo loops. This can give significant performance improvement for test cases with high post processing overhead.

The parallel performance of Eirene was further improved by adding two parallelization strategies. The APCAS strategy is a simple method that divides all the

work evenly among the MPI tasks. If the post processing overhead is small, then it usually gives a good performance. However, the MPI communication overhead can be large for the APCAS strategy, specifically if a high number of MPI tasks is used. The Balanced strategy optimizes the total working time (including post processing overhead) and minimizes the MPI communication cost. In general it gives a very good performance for various combinations of input data and MPI task numbers. The new parallelization strategies are now integrated into SOLPS-ITER, and the Balanced strategy is the new default parallelization method.

The technical details of the coupling between the OpenMP B2 code and the MPI Eirene code were investigated. The current implementation is adequate for the hybrid code, and we do not expect any performance degradation in the coupled code.

Lorenz Hüdepohl is currently incorporating the OpenMP parallelization from SOLPS 5.0 into SOLPS ITER, which is the last step in the SOLPS optimization project.

3.7. *Bibliography*

SOLPS 5.0 98 species test case. (2016). Retrieved from https://solps-mdsplus.aug.ipp.mpg.de/repos/SOLPS/trunk/solps_examples/solps5.0/bench_mark/ITER_D+T+He+Be+Ne+W

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

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

Fehér, T. (2015). Report on HLST project SOLPSOPT.

McCalpin, J. D. (1995). Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE TCCA Newsletter, December.

Parr, T., & Harwell, S. (2016). ANTLR. Retrieved from http://www.antlr.org

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

4.1. ***Introduction***

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

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

The recently proposed field line map approach (also called flux-coordinate independent approach) addresses the geometrical issues: A cylindrical grid (*R,Z,phi*) is used, which is Cartesian within poloidal planes. The discretisation of perpendicular operators is straight forward and simple as their stencils remain within (Cartesian) poloidal planes. Parallel operators are discretised with finite differences along magnetic field lines. Field line tracing from plane to plane is performed and required values on a magnetic field line are thereby obtained by interpolation within poloidal planes. A grid sparsification in the toroidal direction is employed to exploit the flute mode character. Ultimately, simulations in tokamaks (and even stellerators) with an arbitrary poloidal cross section are possible. In particular simulations across the separatrix in diverted devices pose no problems any more.

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

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

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

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

4.2. *Discretization of the elliptic problem in GRILLIX*

In the GRILLIX code, we need to solve the following elliptic partial differential equation (PDE) on poloidal planes as shown in Fig. 22:

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

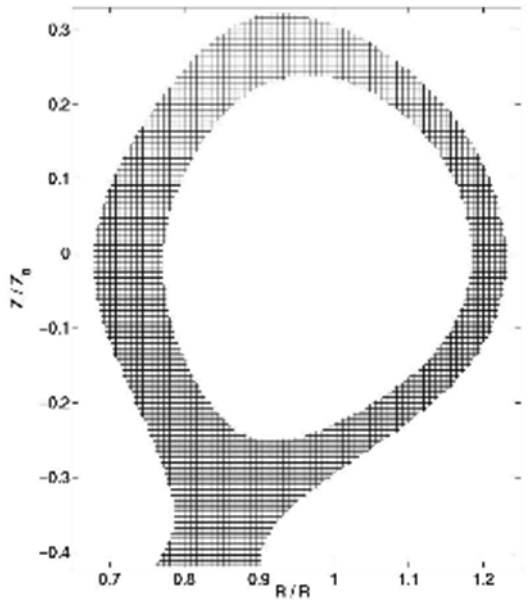


Fig. 22 The Cartesian coordinate of the GRILLIX code.

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

$$-\frac{c_0}{h_y^2}u_{i,j-1} - \frac{c_0}{h_x^2}u_{i-1,j} + \left(\frac{2c_0}{h_x^2} + \frac{2c_0}{h_y^2}\right)u_{i,j} - \frac{c_0}{h_x^2}u_{i+1,j} - \frac{c_0}{h_y^2}u_{i,j+1} = f_{i,j}.$$

These discretisations are well developed and well analyzed. Also, the derived linear system can be solved by the multigrid method with very good performance and scaling properties:

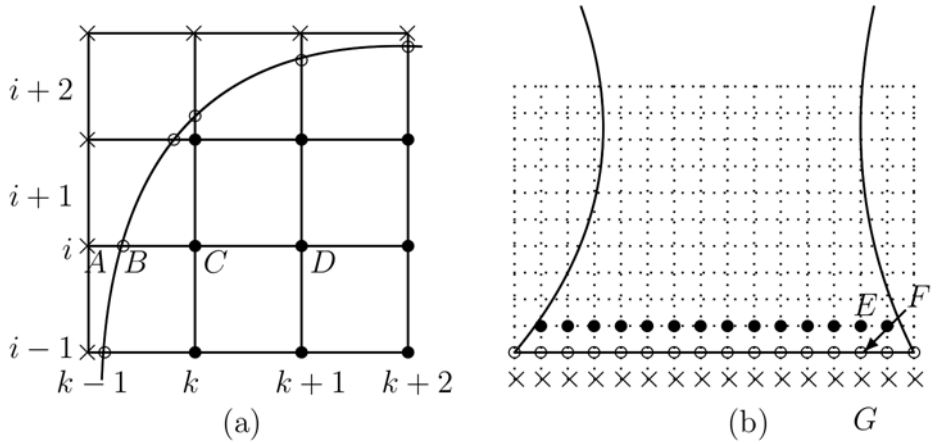


Fig. 23 The Cartesian coordinate of the GRILLIX code near the boundary with a Dirichlet boundary condition (a) and a Neumann boundary condition at the boundary perpendicular to the *y*-axis (b).

As an example of how to handle a Dirichlet boundary condition in

Fig. 23 (a), we consider the point *C* where the five stencil notation has to use the boundary point *B* instead of a point *A* which is out of the domain. There are two ways to generate the linear system, one is by directly using the point *B* and the other to approximate the value at the point *A* with the value of the points *B* and *C*.

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

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u \approx -\frac{2c_0}{(h+h_x)h}u(B) + \frac{2c_0}{hh_x}u(C) - \frac{2c_0}{(h+h_x)h_x}u(D)$$

by using the boundary point *B*. If we use the first order approximation value for *A*

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

then we have

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u \approx -\frac{c_0}{h_x h}u(B) + \frac{c_0}{h_x^2}\left(1 + \frac{h_x}{h}\right)u(C) - \frac{c_0}{h_x^2}u(D).$$

If we use the second order approximation value for *A*

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

then we have

$$-\frac{\partial}{\partial x}c_0\frac{\partial}{\partial x}u \approx -\frac{2c_0}{h(h+h_x)}u(B) + \frac{2c_0}{h_x h}u(C) - \frac{2c_0}{h_x(h_x+h)}u(D).$$

We can get similar results for

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

For the Neumann boundary condition at the point *F* (Fig. 23 (b)), we calculate the approximating value at point *G* using the normal derivative at the point *F* and the value at the point *E*, i. e.,

$$g_n(F) = \frac{\partial}{\partial n}u(F) = -\frac{\partial}{\partial y}u(F) = \frac{u(E)-u(G)}{2h_y},$$

i.e.,

$$u(G) = u(E) - 2h_y g_n(F).$$

If we put these values in the five-point stencil scheme, we get the finite difference approximation which has still a five-point stencil.

Now, we consider the Neumann boundary conditions on general shaped boundaries. A typical boundary shape near the divertor is shown in Fig. 24. A typical boundary condition is a Dirichlet boundary condition on the outer boundary except at the divertor, and a Neumann boundary condition on the inner boundary and the divertor.

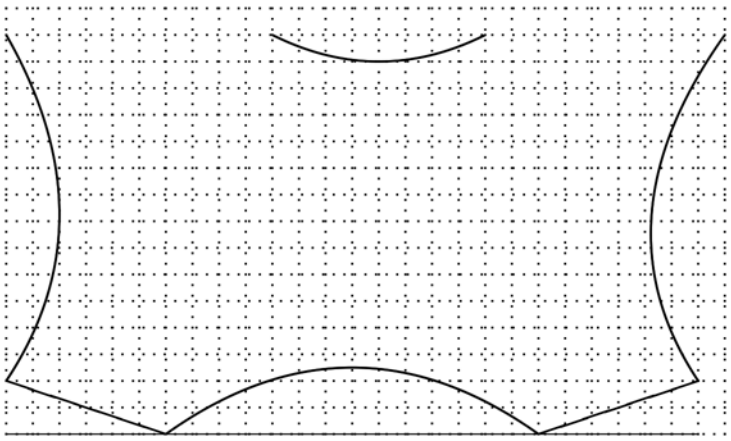


Fig. 24 The typical boundary of the GRILLIX code near the divertor.

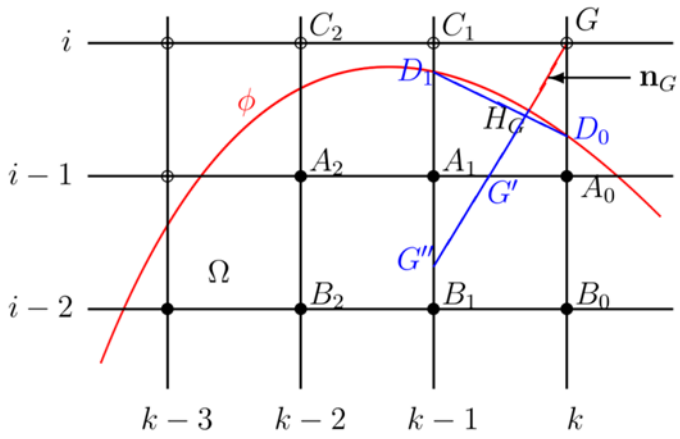


Fig. 25 The stencil is in the upwind direction with respect to the normal vector $\boldsymbol{n}_G$ computed in H_G, while the discretization is performed along the co-normal direction.

To handle the Neumann boundary condition on a general shaped boundary including the region near the divertor, we may choose zero-, first-, or second-order approximation schemes. To get the value at a ghost point G, we compute the outward unit normal at G, that is $n_G = (n_G^x, n_G^y) = \boldsymbol{\nabla\phi}/|\boldsymbol{\nabla\phi}|$, using a discretization for $\boldsymbol{\nabla\phi}$ (see Fig. 25). Now we can compute the closest boundary point to G, that we call H_G, by the signed distance function:

$$H_G = G - \mathbf{n}_G \cdot \phi(G).$$

We discretize the Neumann boundary condition

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

where u and ϕ are bilinear or biquadratic interpolants on the following upwind stencil St_m:

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

with $s_x = \mathrm{sgn}(x_{HG}\text{-}x_G)$ and $s_y = \mathrm{sgn}(y_{HG}\text{-}y_G)$.

For practical reasons, we approximate the boundary by the line D_0D_1 where GH_G is the perpendicular line to D_0D_1. For the zero-order approximation ($m = 0$), we may

choose a point G_1 to either G' or G'', which are the crossing points of the stencil and the line GH_G. We approximate the point G_1 as the reflection point of G, i.e., $|GH_G| = |G_1H_G|$. Then, we have

$$g_n(H_G) \approx g_n(G) = \frac{\partial u_h}{\partial n}(G) = \frac{u_G - u_{G_1}}{|GG_1|}$$

i.e.,

$$u_G = |GG_1| g_n(H_G) + u_{G_1}.$$

For the first-order approximation (m=1), we have

$$g_n(H_G) = g_n(G) = \frac{\partial u_h}{\partial n}(G) = \frac{(u_G - u_{C_1})|n_x|}{h_x} + \frac{(u_G - u_{A_0})|n_y|}{h_y},$$

i.e.,

$$u_G = \frac{h_x h_y}{|n_x| h_y + |n_y| h_x} \left(g_n(H_G) + \frac{|n_x|}{h_x} u_{C_1} + \frac{|n_y|}{h_y} u_{A_0} \right).$$

For the second-order approximation (m = 2), we need to use the nine-point stencil St_2. For the typical case of the nine-point stencil as shown in Fig. 25, we have

$$g_n(H_G) = g_n(G) = \frac{\partial u_h}{\partial n}(G) = \frac{(3u_G + u_{C_2} - 4u_{C_1})|n_x|}{2h_x} + \frac{(3u_G + u_{B_0} - 4u_{A_0})|n_y|}{2h_y},$$

i.e.,

$$u_G = \frac{2h_x h_y}{3(|n_x| h_y + |n_y| h_x)} \left(g_n(H_G) + \frac{2|n_x|}{h_x} u_{C_1} + \frac{2|n_y|}{h_y} u_{A_0} - \frac{|n_x|}{2h_x} u_{C_2} - \frac{|n_y|}{2h_y} u_{B_0} \right).$$

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

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

Here we consider how to calculate the Laplace equation on each point. To do this, we classify the value on the ghost (ϕ_g) and the real (ϕ_r) points. In the current GRILLIX version, we use the following numbering: a negative integer for the ghost points and a positive integer for the real points with dummy value zero, i.e., from **ghost%lpnti** (< 0, negative number of the ghost points on the grid, *-ng*) to **ghost%lpntf** (= -1) and from **grid%lpnti** (= 1) to **grid%lpntf** (number of the real points on the grid, *nr*). Then, Eq. (4.2.1) will be the following system

$$\begin{pmatrix} B_g \\ 1 \\ A_r \end{pmatrix} \begin{pmatrix} \phi_g \\ 0 \\ \phi_r \end{pmatrix} = \begin{pmatrix} f_g \\ 0 \\ f_r \end{pmatrix}$$

where ϕ_g is a ng dimensional vector, ϕ_r is a nr dimensional vector, B_g is a $ng \times (ng+nr+1)$ matrix which is for computing the values on the ghost points, and A_r is

a *nr*×(*ng*+*nr*+1) matrix for the five-point stencil on the real points. For the homogeneous boundary condition (zero values for the Dirichlet or Neumann boundary), f_g will be zero. If we rearrange the ghost points according to the boundary condition near the ghost points (first the Dirichlet boundary condition then the Neumann boundary condition). Then we have the following system;

$$\begin{pmatrix} I_D & \mathbf{0} & \mathbf{0} & B_D \\ \mathbf{0} & I_N & \mathbf{0} & B_N \\ \mathbf{0} & \mathbf{0} & 1 & \mathbf{0} \\ \mathbf{0} & A_N & \mathbf{0} & A_{rr} \end{pmatrix} \begin{pmatrix} \phi_D \\ \phi_N \\ 0 \\ \phi_r \end{pmatrix} = \begin{pmatrix} f_D \\ f_N \\ 0 \\ f_r \end{pmatrix}$$

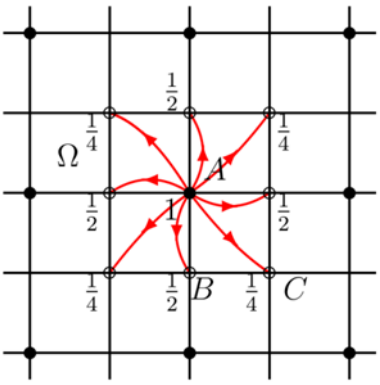
where ϕ_D is a *nD* dimensional vector, ϕ_N is a *nN* dimensional vector (*nD*+*nN* = *ng*), B_D is a *nD*×*nr* matrix, B_N is a *nN*×*nr* matrix, A_N is a *nr*×*nN* matrix and A_{rr} is a *nr*×*nr* matrix. From this system, we know that ϕ_D is not needed during the computation and can be removed.

4.3. *Multigrid algorithm*

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

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

The intergrid transfer operators on the Cartesian grid are well known, but we need a modification near the boundary. For the prolongation operators, we can use a typical linear interpolation, as shown in Fig. 26, after all values at the ghost points of the coarser grid are computed.



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

Fig. 26 The stencil notation of the prolongation transfer operator

For the projection operators, we use the typical linear projection operator which is well defined and tested. This operator is shown on the left of Fig. 27. But, we need some modifications near the boundary when some of the neighbor points are ghost points. Usually, we restrict the residual on the finer grid and cannot impose the boundary condition. However, we need contributions from ghost points. One way is to set the value on the reflected point in relation to the point in a coarse grid as the value at the ghost point. Then we have the stencil notation as shown in the second and third Fig. 27.

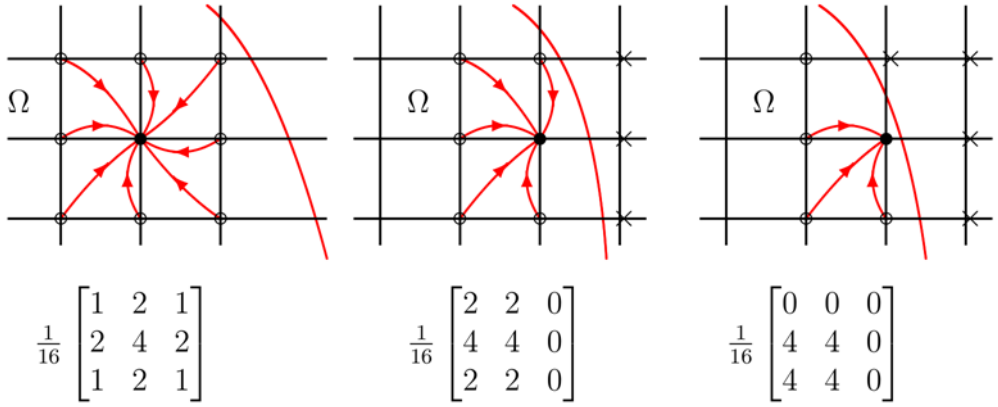


Fig. 27 The stencil notation of the projection transfer operator near the boundary.

4.4. *Conclusions and outlook*

The contribution of the project coordinator was to implement the matrix generation routine for the Neumann boundary condition with zero-order approximation. As a result, GRILLIX can handle the Dirichlet and Neumann boundary conditions on a general shaped domain. This might be improved in a further step by adding first- and second-order approximations for the Neumann boundary condition.

An unreasonable implementation of the restriction operator was found which used only the value of one point at the finer grid to get the value at the coarser grid. This caused a five time pre- and post-smoothing with the Jacobi smoother in order to achieve convergence. Therefore, we had to modify the restriction and prolongation operators to achieve faster convergence. In addition, we implemented Jacobi and Gauss-Seidel smoothers. The multigrid solver itself was rewritten. Also, a multigrid solver as a preconditioner for GMRES was implemented. In the future we will have to investigate which of the two algorithms is the more efficient one.

As a next step we need to verify the modified code and to tune some numerical parameters, e.g. an optimal number of smoothing iterations for the Jacobi and Gauss-Seidel smoothers.

5. **Final report on HLST project MGBOUT3D**

5.1. Introduction

First principle simulations of Scrape-Off Layer (SOL) turbulence can provide important insight in the mechanisms governing plasma exhaust. The BOUT++ code has proved over the years to be able to tackle several aspects of the problem, thanks to its user-friendly, robust approach to defining models and geometry. However, to prepare for full 3D simulations in realistic geometry and with saturated turbulence, the code needs to be optimized also in terms of performance.

A significant step in this direction was taken in 2015, when multigrid techniques were introduced in the 2D linear elliptic solver of BOUT++ with the help of the HLST. This solver has applications to several parts of the code, in particular calculating electric and magnetic potentials, and preconditioning of the implicit time step. In particular, to derive the electrostatic potential from the vorticity, which is solved for the Shear-Alfven law, a Laplace problem must be inverted in each iteration of the implicit time-stepper (this takes a substantial fraction of the total simulation time and strongly limits the scaling performance). Its form is as follows:

$$d\nabla \cdot \nabla_\perp u + \frac{1}{c_1}\nabla c_2 \cdot \nabla_\perp u + au = f$$

where a, c_1, c_2 and d are constants and u is the potential to be solved for.

In the current version of the BOUT++, the inversion is performed on independent planes at constant poloidal angles, approximating the second order operator by removing cross terms connected with the parallel derivatives. In the original version of the code, the solver was direct: Fast Fourier Transforms in the “toroidal” direction reduced the problem to the inversion of a tridiagonal matrix, which was solved with the Thomas algorithm. The “toroidal” direction was not parallelised since FFTs are best performed on contiguous data, but achieving this requires communication over the entire grid to rearrange storage. This constraint survives in the 2015, multigrid version of the 2D solver.

The incomplete formulation of the Laplacian problem due to the lack of parallel derivatives entails another important limitation of the current approach: the decoupling of the planes. This implies, for example that the 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 perpendicular boundary conditions that can be applied; 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 behaviour. This issue is of paramount importance in the SOL: the presence of sheath boundaries induces strong parallel variation of the plasma parameters that exposes the limitations of decoupled 2D solvers.

One task in this project is to develop and to implement a multigrid scheme to allow a pseudo 3D (2D+1D) Laplace inversion. This task is the natural extension of the work performed by HLST in 2015 on the implementation of multigrid algorithms for the 2D solvers. The modularity of BOUT++ restricts the modification of the multigrid algorithm only to the Laplace solver part of the code (only ~1000 lines out of ~85,000).

Due to the more onerous nature of the pseudo 3D problem, extending the parallelisation of the code to the “toroidal” direction will ensure that the extra physics will be implemented without loss of performance of the code. The absence of such parallelisation is a legacy feature, which could be removed by using the MPI protocols already implemented in the other two directions. Therefore, the next task is to parallelise the direction along the field line label coordinate using MPI. In addition,

parallelisation in the third direction will enable full exploitation of the power of the multigrid approach in medium size simulations that are important for parameter scans, where the present 2D approach saturates in efficiency at a relatively small number of processors. Other tasks are to allow for different equations on different-sized grids to help to obtain non-Boussinesq solutions.

In the long term, we will upgrade the code, by refactoring the mesh implementation, to be able to use multigrid methods more widely. Application to the implicit time-stepper should reduce the number of iterations per time-step and/or enable longer time steps to be taken; in either case this will reduce the number of right hand side evaluations needed in a fixed amount of simulation time, which should decrease in direct proportion the run-time. The re-design process should identify, with a view to efficient multigrid implementation, things like: appropriate representation in memory of the fields; methods to iterate over and operate on fields; optimizing the topology of distribution of the grid(s) across processes to minimize communication overhead; and how to simultaneously maintain the flexibility to alter easily the physics model (i.e. the differential equations to be solved).

5.2. *BOUT++ structures: discretization and parallelization*

To construct the solver, we need to know how to discretize and handle the data. So in this section, we summarize the current structures of BOUT++ which are related to the second order partial differential equation (PDE) on 2D for 3D (2D+1D parallel) problems and explain the requirements of the 3D parallel solvers.

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. 28 (2D+1D parallel). Each slice can be transformed to a reference domain (rectangular or unit square) with periodic boundary condition along the *z*-direction by the conformal mapping *g* as shown in Fig. 29.

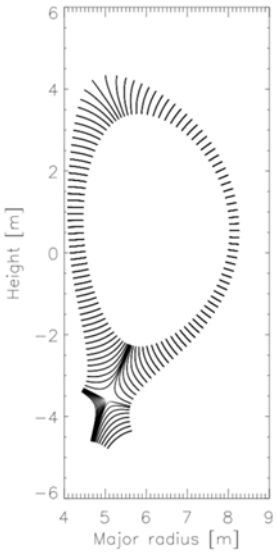


Fig. 28 The coordinates in the poloidal plane of a tokamak in the BOUT++ code. The x- and y- coordinates lie in the poloidal plane. The solid lines denote the x-coordinate lines.

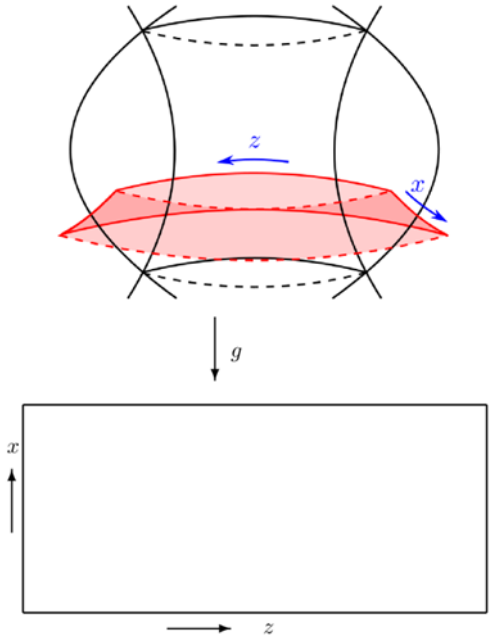


Fig. 29 Computational domain and its conformal mapping

We consider the following second order PDE

$$d\nabla \cdot \nabla_\perp u + \frac{1}{c_1}\nabla c_2 \cdot \nabla_\perp u + au = f \tag{5.2.1}$$

where

$$\nabla_\perp \equiv \left(\frac{\partial}{\partial x}, 0, \frac{\partial}{\partial z}\right).$$

By using the conformal mapping *g*, we get the following relations on the reference domain

$$\begin{aligned}\nabla \cdot \nabla_\perp &= g^{xx}\frac{\partial^2}{\partial x^2} + g^{zz}\frac{\partial^2}{\partial z^2} + 2g^{xz}\frac{\partial^2}{\partial x \partial z} + g^{ij}\Gamma^x_{ij}\frac{\partial}{\partial x} + g^{ij}\Gamma^z_{ij}\frac{\partial}{\partial z} \\ &\quad + g^{yx}\frac{\partial^2}{\partial x \partial y} + g^{yz}\frac{\partial^2}{\partial y \partial z} \\ \nabla c \cdot \nabla_\perp u &= g^{xx}\left(\frac{\partial c}{\partial x}\right)\left(\frac{\partial u}{\partial x}\right) + g^{zx}\left(\frac{\partial c}{\partial z}\right)\left(\frac{\partial u}{\partial x}\right) \\ &\quad + g^{xz}\left(\frac{\partial c}{\partial x}\right)\left(\frac{\partial u}{\partial z}\right) + g^{zz}\left(\frac{\partial c}{\partial z}\right)\left(\frac{\partial u}{\partial z}\right).\end{aligned}$$

With the assumption that the plasma is strongly magnetized, the terms in red are small and can be neglected and are able to decouple the 3D problem in the toroidal direction. As a result the PDEs can be solved independently on each poloidal plane.

To get the discretized solution of Eq. (5.2.1), BOUT++ uses cell centered second order finite difference schemes on the reference domain (Fig. 30). By combining the computations, we have a linear finite system with the following entries in stencil notation

$$\begin{bmatrix} a_{i-1,k+1} & a_{i,k+1} & a_{i+1,k+1} \\ a_{i-1,k} & a_{i,k} & a_{i+1,k} \\ a_{i-1,k-1} & a_{i,k-1} & a_{i+1,k-1} \end{bmatrix}.$$

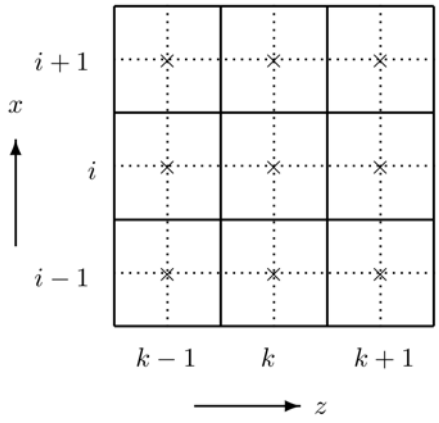


Fig. 30 Notation for discretization

From now on, we consider the parallelization concept to solve Eq. (5.2.1) in BOUT++. The domain has periodic boundary conditions in the *z*-direction and Dirichlet or Neumann boundary conditions in the *x*-direction and is discretized with $N_{x\_global} \times N_{z\_global}$ mesh-cells as shown in Fig. 31.

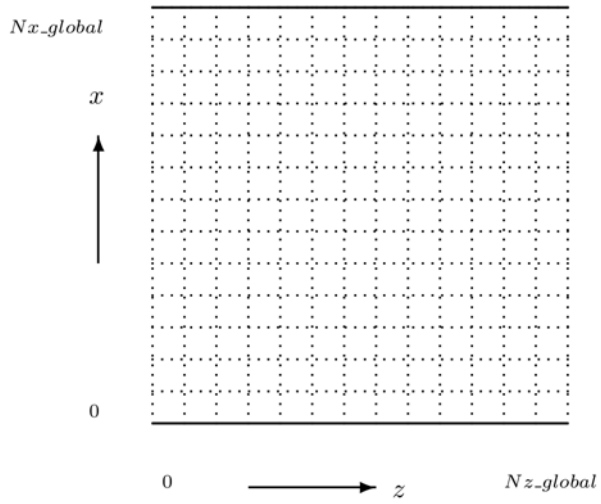


Fig. 31 Global numbering

The current BOUT++ is parallelized only in the *x*-direction and needs to communicate data between neighboring MPI tasks in this direction. Even without being parallelized in the *z*-direction, it still needs ghost nodes at the ends of the *z*-direction due to the periodic boundary condition. At the project MGBOUT in 2015, we developed a parallel multigrid solver according to this 1D parallelization. In the beginning of 2016, the BOUT++ team finished the implementation of the OpenMP parallelization for the 2D PDE problem, i.e., the OpenMP/MPI hybrid parallelization. The new version of the parallel multigrid solver also relies on the OpenMP/MPI hybrid parallelization.

The above mentioned 2D-1D approach becomes problematic near the X-point where the coefficients g^{yx} and g^{yz} cannot be neglected. One way to solve this difficulty is to use a 3D solver to solve the full 3D problem. However, the disadvantage is that the parallelization is not as trivial as for the decoupled 2D+1D case. In addition, the number of degrees of freedom (DoF) is much larger for the 3D problem than for the separated 2D problem, which increases the computational costs of the solver significantly. The best solution is a combination of both approaches. Therefore, we will implement a 3D parallel multigrid solver on a cubic reference domain to be able

to solve correctly near the X-point. How to combine the two approaches will be further discussed with the BOUT++ team.

5.3. *Parallel implementation of the multigrid method*

In this section, we explain how to implement the multigrid method to solve the discretized system of Eq. (5.2.1) in a general framework. According to the object oriented framework of BOUT++, we define C++ classes to do so.

5.3.1. **Basic class for multigrid algorithm**

We define a basic multigrid algorithm class for the parallel multigrid solver which stores the basic multigrid algorithm. The basic multigrid cycle can be given in the following form:

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

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

2: solve $A_h u_h = f_h$ by a parallel direct solver or Krylov iteration solver

3: **else**

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

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

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

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

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

9: **end if**

To implement it in a C++ class, we construct a basic abstract class **KMGhlst_ab()**. We include different implementations of the In functions in this C++ class, according to how to use the memory allocation and pointers. The typical way is that a function returns the pointer to the memory as shown in Fig. 32 (a). This implementation needs to allocate the memory inside the functions and deallocates it. The next possibility is that the pointer of the memory is transferred to the functions as shown in Fig. 32 (b). This implementation needs to allocate the memory before the functions are called. These two implementations will allocate the memory when required, but they may cause irregular memory allocation. A third way is that all required memory is allocated when the multigrid classes are constructed by calling the function as shown in Fig. 32 (c). This method will reduce the time to allocate memory during solving the equations and result in a regular memory allocation. However, such a implementation needs a full awareness of the algorithm and its memory usage a priori and may lead to more cache misses when the allocated memory is large. In spite of the obstacles of the third implementation, we chose it and were able to reduce the solution time on each time step as shown in section 5.4.

We show the definition of **KMGhlst_ab()** and the function **KMGhlst_ab::cycleMG()** which is the main function of the multigrid algorithm in Fig. 33 and Fig. 34. In the class **KMGhlst_ab()**, we combine the functions which do not depend on discretization and parallelization. Other functions are defined as abstract functions which will be implemented in derived sub-classes.

```
kVec *cycleMG(int level,kVec *rhs);
kVec *pGMRES(kVec *rhs,int level, int iplag);
```

(a) Return pointer

```
void cycleMG(int level,kVec *rhs,kVec *sol);
void pGMRES(kVec *rhs,kVec *sol, int level, int iplag);
```

(b) Transfer pointer

```
void cycleMG(int level);
void pGMRES(int level, int iplag);
```

(c) Preallocate memories

Fig. 32 Three implementations of a function in Class **KMGhlst_ab()**

```
class KMGhlst_ab{
public:
  KMGhlst_ab(int ,int ,int ,MPI_Comm );
  virtual ~KMGhlst_ab();
  void getSolution(const int );
  int mglevel,solvMe;
  kmgMat *matf;
  kVec *xx,*rr;
protected:
  double rtol,atol,dtol,omega;
  char mlog[128];
  kmgMat *matmg[20];
  kVec *xp[20],*yp[20],*rp[20],*vg[MAXGM+1],*vx[MAXGM+2],*x0,*r0;
  kRd *rdmg[20];
  int *ldim,*gdim;
  MPI_Comm commMG;
  void cycleMG(int );
  void pGMRES(int level, int iplag);
  void pCGM(int level, int iplag);
  void solveMG(int ); // MG as a solver
  virtual void precond(int ,int );
  virtual void projection(int )=0;
  virtual void prolongation(int )=0;
  virtual void lowestSolver(int );
};
```

Fig. 33 Class **KMGhlst_ab()**

```
void KMGhlst_ab::cycleMG(int lv) {
  if(lv == 0) lowestSolver(0);
  else {
    xp[lv]->init(0.0);
    int ll = lv-1;
    for(int i = 0;i<nsm;++i) {
      if(mgsm == 1) matmg[lv]->smooth(xp[lv],rp[lv],omega);
      else  matmg[lv]->smooth(xp[lv],rp[lv]);
    }
    matmg[lv]->residA(xp[lv],rp[lv],yp[lv]);
    projection(lv);
    cycleMG(ll);
    prolongation(ll);
    xp[lv]->add(yp[lv]);
    for(int i = 0;i<nsm;++i) {
      if(mgsm == 1) matmg[lv]->smooth(xp[lv],rp[lv],omega);
      else  matmg[lv]->smooth(xp[lv],rp[lv]);
    }
    xp[lv]->communicate();
  }
  return;
}
```

Fig. 34 Function **KMGhlst_ab::cycleMG()**

Usually Krylov subspace methods (CGM, GMRES, …) and direct methods can be used as coarsest level solvers. These are well known fast solvers for small problems but are not scaling well with respect to the number of DoF. However the Krylov subspace methods need only a matrix-vector multiplication and a norm calculation for vectors and the multigrid method does not guarantee convergence as a solver for some cases. Hence, we should consider the Krylov subspace methods. For this purpose, we implement the Preconditioned GMRES method which guarantees the convergence even for nonsymmetric problems. In a hybrid method the multigrid method can be used as a preconditioner for the Krylov subspace methods, resulting in a solver which converges with a small number of iterations.

The multigrid cycle and the Krylov subspace method can be implemented independently from each other for parallelization and discretization. The abstract class **KMGhlst_ab()** has implementations of these routines. From this basic class, we define sub-classes as shown in Fig. 35. The implementation of these functions has to follow the discretization of the PDEs and the parallelization for different dimensions. First, we define additional abstract sub-classes for the multigrid algorithm on 2D (**KMGhlst_2db()**) and 3D (**KMGhlst_3db()**). Then, we define further subclasses for the multigrid algorithms for the parallel cell-centered finite difference schemes on the rectangular domain for 2D (**KMG2D_ccfd()**) and on the cubic domain for 3D (**KMG3D_ccfd()**). In the project MGBOUT in 2015, we derived 1D parallel, 2D parallel, and serial versions of the multigrid algorithm from the basic parallel multigrid solver class **MultigridAlg()** (which is replaced by **KMG2D_ccfd()**) for 2D by assigning the number of MPI tasks in each direction.

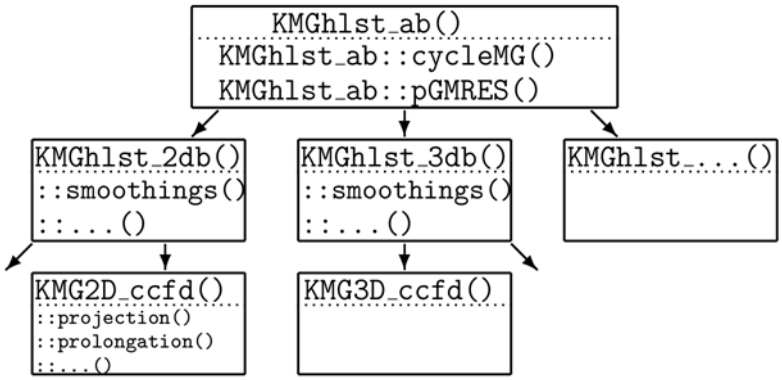


Fig. 35 The basic multigrid class and its subclasses

5.3.2. **The CCFD MG: Intergrid operators**

The intergrid transfer operators, i.e., the restriction and prolongation operators, are one of the key elements in the multigrid algorithm and depended on the discretization scheme for the geometric multigrid method. As shown for the basic multigrid cycle, we need coarse grid operators A_H on each grid level. A_H can be obtained by direct computations on each grid level, but to do so we need the conformal mapping value g on each grid level. To avoid computing the conformal mapping values on each coarser grid, we use an algebraic relation for the coarse and fine grid

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

where c can be chosen to get good performance. In this implementation, we choose c to make A_H the same operator as for the geometric version of the Poisson problem with constant coefficient on a uniform mesh.

For the cell centered discretization method, we have to use a zero-order intergrid transfer operator, i.e., the average value of four fine cells in 2D and of eight fine cells in 3D for one coarse cell. We can write according to the notation in Fig. 36 for 2D CCFD:

$$(I_h^H\phi)_{I,K} = \frac{1}{4}\{\phi_{i,k} + \phi_{i+1,k} + \phi_{i,k+1} + \phi_{i+1,k+1}\}$$
$$(I_H^h\varphi)_{i,k} = (I_H^h\varphi)_{i+1,k} = (I_H^h\varphi)_{i,k+1} = (I_H^h\varphi)_{i+1,k+1} = \varphi_{I,K}$$

and for 3D CCFD

$$\begin{aligned}(I_h^H\phi)_{I,K} = \frac{1}{8}\{&\phi_{i,j,k} + \phi_{i+1,j,k} + \phi_{i,j,k+1} + \phi_{i+1,j,k+1}\\ &+ \phi_{i,j+1,k} + \phi_{i+1,j+1,k} + \phi_{i,j+1,k+1} + \phi_{i+1,j+1,k+1}\}\\ (I_H^h\varphi)_{i,j,k} = (I_H^h\varphi)_{i+1,j,k} &= (I_H^h\varphi)_{i,j,k+1} = (I_H^h\varphi)_{i+1,j,k+1} = (I_H^h\varphi)_{i,j+1,k}\\ = (I_H^h\varphi)_{i+1,j+1,k} &= (I_H^h\varphi)_{i,j+1,k+1} = (I_H^h\varphi)_{i+1,j+1,k+1} = \varphi_{I,K}.\end{aligned}$$

In the stencil notation we can write

$$I_H^h = \begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}, \qquad I_h^H = \begin{bmatrix}\frac{1}{4} & \frac{1}{4}\\ \frac{1}{4} & \frac{1}{4}\end{bmatrix}$$

for 2D and in similar way for 3D.

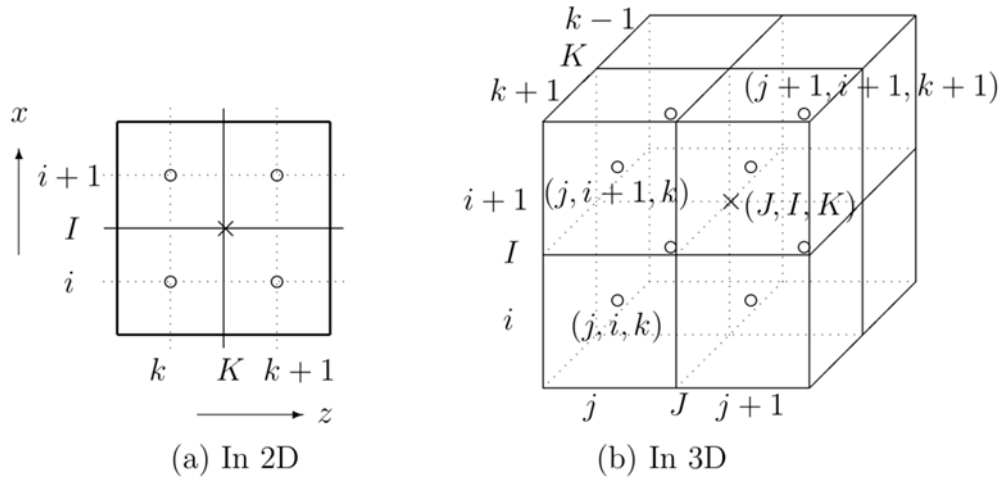


Fig. 36 Notation for the intergrid transfer operators

5.3.3. **The CCFD MG: Parallelization**

Some operations in the multigrid algorithm and the Krylov subspace method have to be implemented according to the parallelization of the discretized problem. The matrix-vector multiplication, the residual computation, and the smoothings are examples of such operations. To implement them, we have to consider how to parallelize the domain. In this subsection, we consider the parallelization and the smoothing of the multigrid algorithm. We restrict our explanation to the *x*-*z* 2D domain, but it would also apply to the *x*-*y*-*z* 3D domain by adding just the *y*-direction.

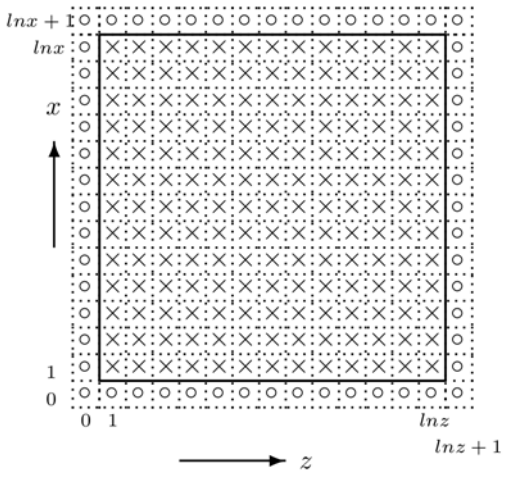


Fig. 37 Local numbering on each MPI task

We assume that each MPI task handles a subdomain as shown in Fig. 37. To parallelization, we divide the domain with the *xNP* × *zNP* subdomains where *xNP* and *yNP* are the number of divisions along *x*-direction and *y*-direction. Each domain has *lnx* × *lnz* cells as shown in Fig. 37 (×) and is handled by one MPI tasks. The **KMG2D_ccfd()** class is defined on this parallelization. For the data communication, we add a guard (ghost) cell (○) at each end which is needed for the matrix-vector multiplication, smoothing operations, and intergrid transfer operators. We plot the data communication pattern in Fig. 38.

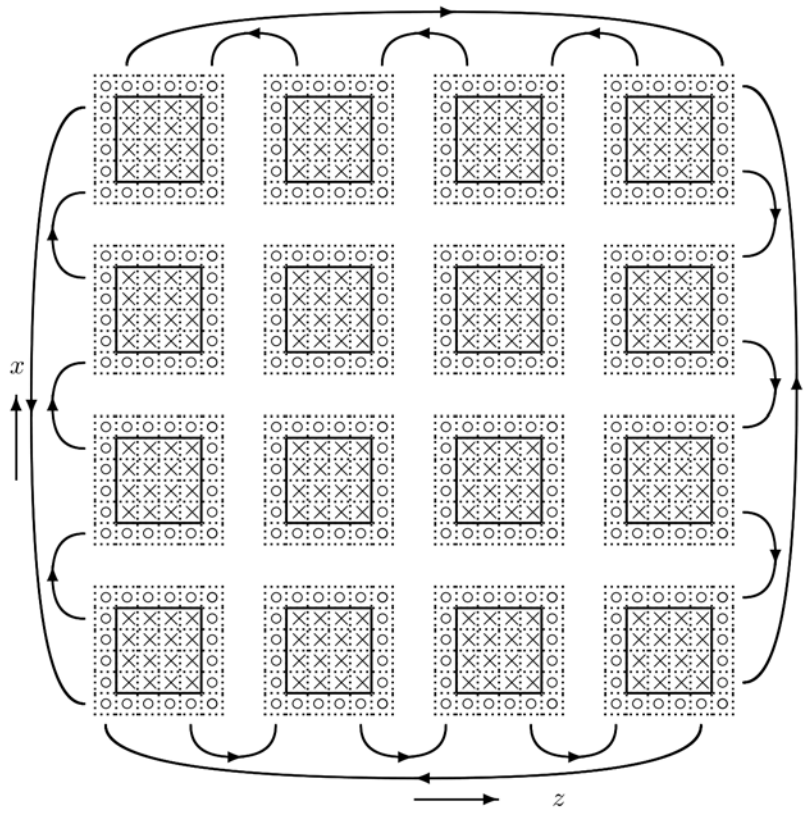


Fig. 38 Basic data communication for the parallel multigrid algorithm

The smoothing iteration has to be simple but must reduce the high frequency error component. The damped Jacobi and Gauss-Seidel iterations are well-known smoothers. The damped Jacobi iteration is slower than the Gauss-Seidel iteration, but its result does not depend on the number of MPI tasks. Therefore, we can use the damped Jacobi iteration for debugging purposes of the parallel code. In production we use the Gauss-Seidel iteration as a smoother.

First of all, we consider the 1D parallelization which is used in the current version of BOUT++. To match the notation of the class, we add extra ghost cells in the first and last domain as shown in Fig. 39.

For the parallel multigrid method, the coarsest level has at least one real cell for each MPI task. This limits the selection of the coarsest level as the number of DoF on the coarsest level has to be greater than or equal to

$$\frac{N_z}{N_x} N_c^2$$

where N_c is a number of MPI tasks. We display the number of DoF on the coarsest level and estimate (theoretically) the required number of iterations of the GMRES solver for $N_z = N_x$ in Table 8. It shows that the number of DoF on the coarsest level is four times larger when the number of MPI is doubled, i.e., the possible number of levels is decreased by one. For example, if the number of DoF on the finest level is 1024×1024 (1M) and the number of MPI tasks is 1024, then we have to solve the problem directly, i.e., we cannot use the multigrid

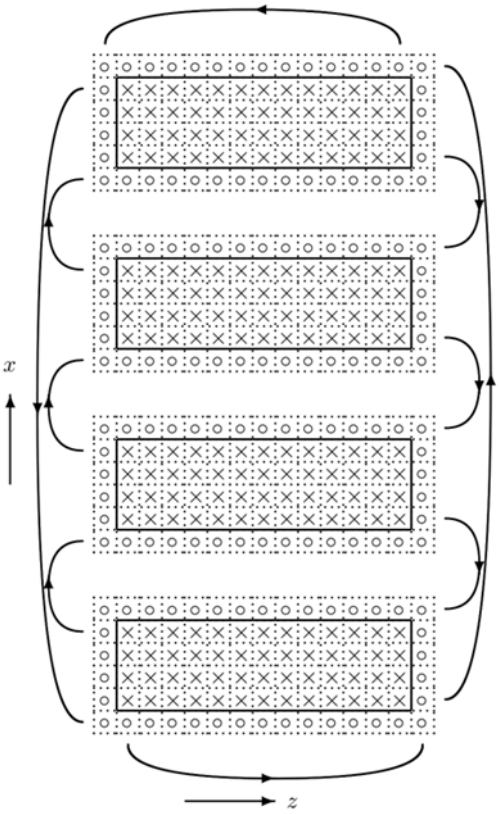


Fig. 39 One dimensional parallelization and its data communication for the multigrid algorithm

Table 8 The number of DoF on the coarsest level and the required number of iterations of GMRES

# MPI	16	32	64	128	256	512	1 024	2 048	4 096	8 192
# DoF	256	1K	4K	16K	65K	260K	1.2M	5M	20M	80M
# GMRES	16	32	64	128	256	512	1 024	2 048	4 096	8 192

The number of iterations required in the GMRES solver increases with the square root of the number of DoF on the coarsest level and the number of DoF increases with the square of the number of MPI tasks. On the coarsest level we have either to use a serial algorithm after gathering the data or to use a 2D parallel algorithm to involve more levels in the computation which would lead to a better scaling property. The serial algorithm achieves good results for small size problems in combination with a small number of MPI tasks, but will have a heavy workload for large size problems which may be the case for large numbers of MPI tasks.

Therefore the desired algorithm would result in a 2D parallelization starting from the finest level, but it would need a 2D parallelization of the whole BOUT++ code which would imply a lot of code refactoring. Instead of using a 2D parallelization beginning from the finest level, we consider to use a 2D parallelization or a serial multigrid algorithm from a certain coarser level which is in practice close to the coarsest level. To do this, we modify the lowest level solver of the 1D parallel multigrid solver.

The limitation of the coarsest level, even though alleviated, occurs also for the 2D parallel multigrid algorithm in case of a large number of MPI tasks. So, if the coarsest level limitation does occur, we will use a serial algorithm after gathering the data on the coarsest level of the 2D parallelization multigrid algorithm. This will be done in the same way as for the 1D parallelization.

In implementation of the multigrid class **KMG2D_ccfd()** is implemented based on a 2D parallelization for this project. The 1D parallel and the serial version are easily obtained by setting the number of MPI tasks (*xNP* and *zNP*) for each direction (*zNP* =

1 for the 1D parallel version and *xNP* = *zNP* = 1 for the serial version), so we need to implement only the transformation for a matrix and a vector.

First, the matrix and data transformation from the parallel algorithm to the serial one is just a gather from all MPI tasks and a reduction to all MPI tasks. To do this, we use **MPI_allreduce()**.

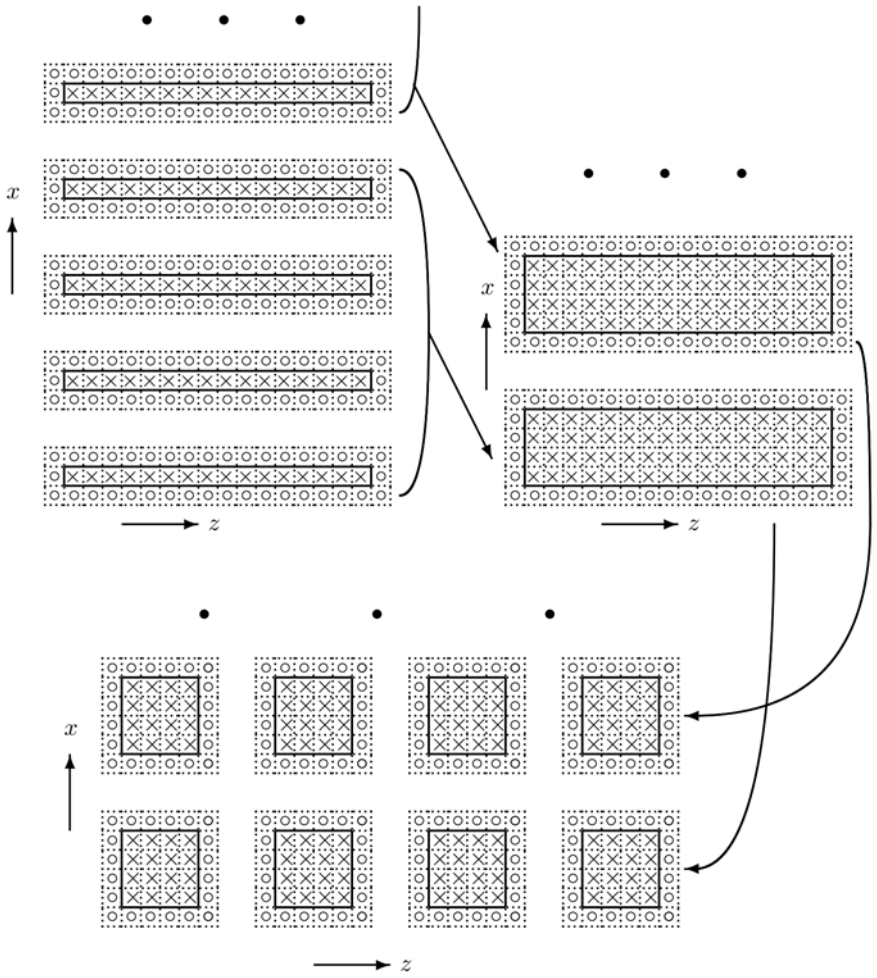


Fig. 40 Change from 1D parallelization to 2D parallelization

Next, we consider the 2D parallelization on the coarsest level. To obtain a 2D parallelization from a 1D parallelization, we choose subsets of all the sub-domains in the *x*-direction to form groups. Within each group, we gather and distribute data from and to group members as shown in Fig. 40. To do this, we use **MPI_allreduce()** and collect data which will be handled by each MPI task. For the vector elements, we have to perform the **MPI_allreduce()** twice, one is 1D to 2D and the other is 2D to 1D. We show such a data transformation (1D to 2D) for 16 MPI tasks to 4 × 4 (the number of entries per subgroup is four) MPI tasks in Fig. 40.

OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems. It uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications. It is an implementation of multi-threading, a method of parallelization whereby a master thread forks a specified number of slave threads and a task is divided among them. The threading then runs concurrently, with the run time environment allocating threads to different processors. Using the OpenMP programming model, multiple threads share tasks which can be parallelized. So, we use multi-threading on the computation intensive parts of the program. The computation intensive routines of the multigrid method are the matrix-vector multiplication, the computation of the residual norm, the smoothing iterations, and the intergrid transfer operators including

the restriction and the prolongation. We have modified these routines by adding OpenMP pragmas.

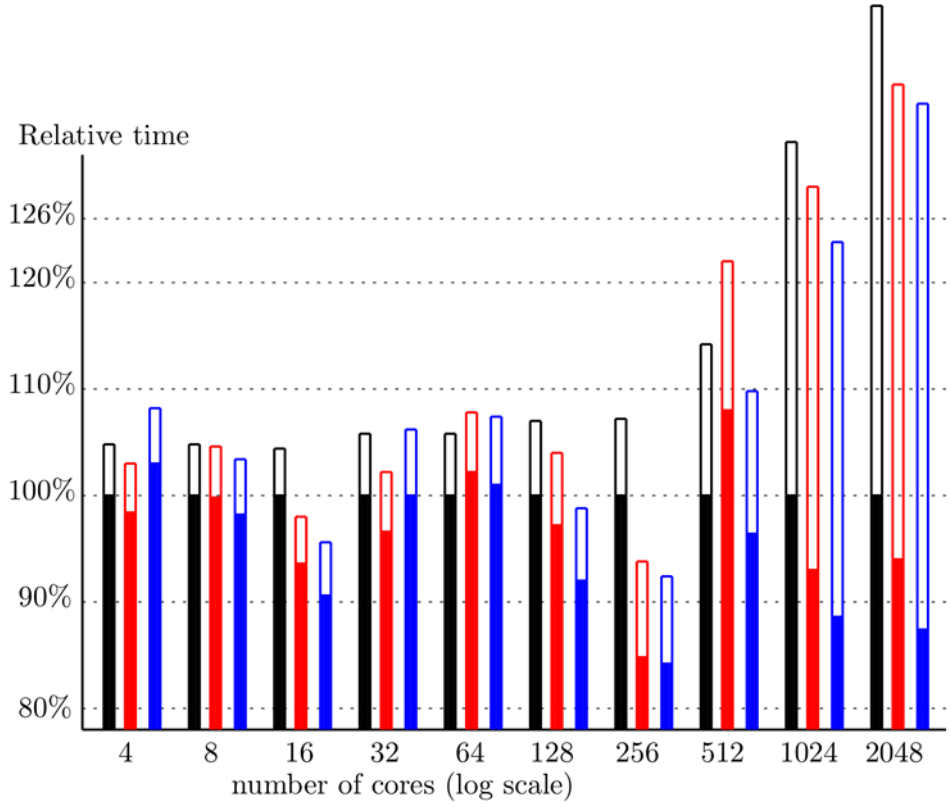
5.4. *Numerical results*

For the accessment of the multigrid method we consider the Poisson problem on the unit square domain with a uniform discretization with the same number of cells in each direction. First, we investigate the required number of iterations for convergence according to the number of DoF or total number of levels, respectively for both a multigrid solver and a PGMRES solver having a multigrid preconditioner. For the multigrid solver we use a Gauss-Seidel smoother (MG(GS)) and for the PGMRES solver a Gauss-Seidel (PGMRES(GS)) and a Jacobi smoother (PGMRES(JA)). Table 9 shows that the required number of iterations does not depend on the number of levels, as expected for the multigrid method.

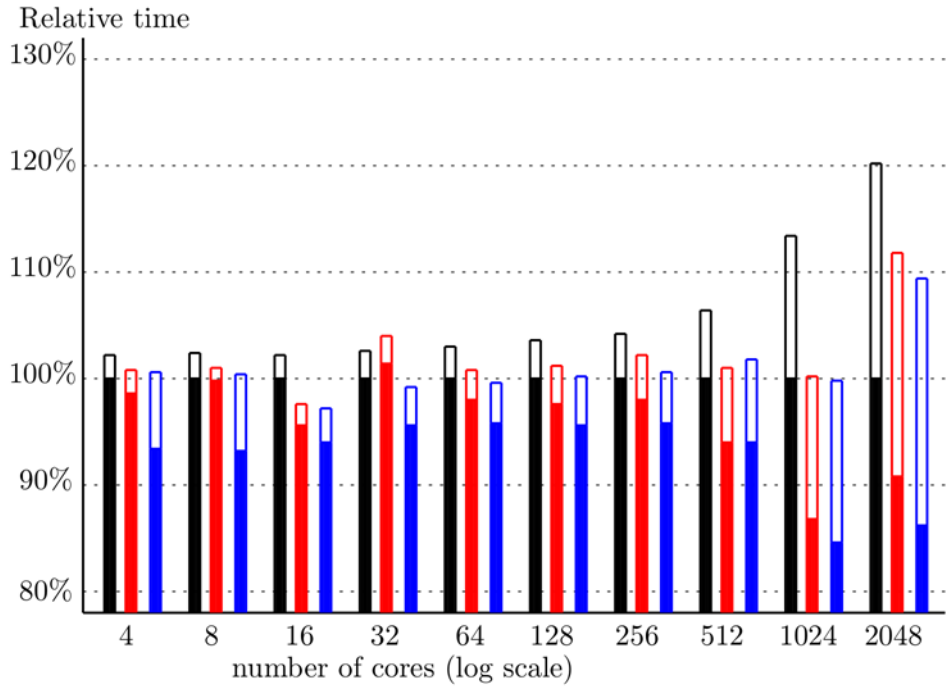
Table 9 Number of required iterations for convergence of three different iterative solvers according to the number of DoF (the total number of levels): multigrid solver with Gauss-Seidel smoother (MG(GS)); PGMRES solver with a multigrid preconditioner using a Gauss-Seidel smoother (PGMRES(GS)) and a Jacobi smoother (PGMRES(JA)).

Finest	Levels	MG (GS)	PGMRES(GS)	PGMRES(JA)
$2^7 \times 2^7$	5	10	7	13
$2^8 \times 2^8$	6	10	8	13
$2^9 \times 2^9$	6	10	8	13
$2^{10} \times 2^{10}$	7	10	8	14
$2^{11} \times 2^{11}$	8	10	8	15
$2^{12} \times 2^{12}$	9	10	9	15
$2^{13} \times 2^{13}$	10	11	9	16
$2^{14} \times 2^{14}$	11	11	9	16
$2^{15} \times 2^{15}$	12	10	9	17
$2^{16} \times 2^{16}$	13	10	8	17

We compare three different implementations in handling the memory allocation as explained in section 5.3.1 (black for (a), red for (b), and blue for (c)). To do this, we consider a problem with 4096 cells in each direction, i.e., 4096^2 DoF. We select two cases, one case is the multigrid solver with the Gauss-Seidel smoother (i) and the other case is the PGMRES solver with the multigrid preconditioner using a Jacobi smoother (ii). We choose the solving time of the first case (a) as a base case and plot the relative times of the solving (filled box) and the setting (unfilled box) times. The resulting time is the time that is needed to generate the matrices on all levels. In Fig. 41 the results are shown according to the number of MPI tasks. Each MPI task is located exclusively on a core. The solving time of the case (c) is almost always the fastest. Also, the summation of the solving time and the setting time has a similar behaviour. In practice, the multigrid class setting is needed once and the solving is repeated several times within each time steps. So the solving time is more important. The relative setting time increases as the number of MPI tasks increases and is significantly larger for the multigrid solver because this method needs less iterations than PGMRES(JA).



(i) The multigrid solver with a Gauss-Seidel smoother



(ii) The PGMRES using a multigrid preconditioner with a Jacobi smoother

Fig. **41** The relative times of the solving (filled box) and the relative setting (unfilled box) times of three different cases (black for (a), red for (b), and blue for (c)) according to the number of used MPI tasks (For details of memory allocation cases see text).

To evaluate the performance of the described parallel algorithms, we measure two quantities: the solution time (black) and the matrices setting time (red). We perform a strong scaling of these quantities for a problem size of 4096^2 DoF. The corresponding results are shown in Fig. 42. First, we use the 1D parallelization algorithm, denoted by — and - - - (case one). For this case, the problem size on the

lowest level increases in the same way as the number of MPI tasks. Second, we use the 1D parallelization and the serial algorithm, denoted by • • • and * * * (case two). And last, we use the combined 1D and the 2D parallelization algorithm, denoted by ○ ○ ○ and ◊ ◊ ◊ (case three).

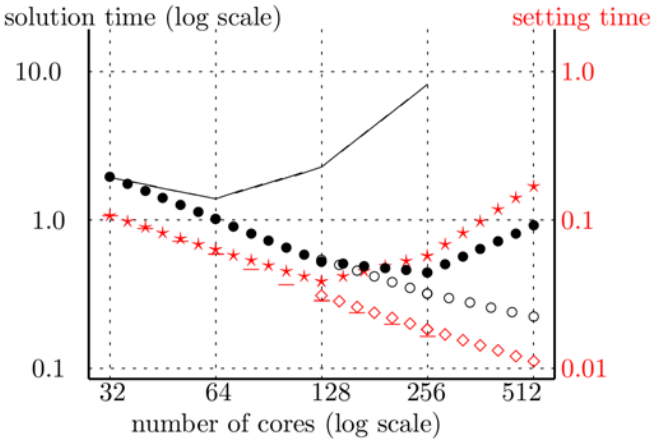


Fig. 42 The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver (in black) and matrices setting time (in red) as a function of the number of cores for a domain of 4096^2 DoF (DoF of the lowest level is 1024) with three different parallelization concepts.

For case one, the number of levels decreases as the number of cores is increased, i.e., 8 levels for 32 MPI tasks, 7 levels for 64 MPI tasks, 6 levels for 128 MPI tasks, ... This means that the number of DoF on the lowest level increases, i.e., 1024 for 32 MPI tasks, 4096 for 64 tasks, 16384 for 128 MPI tasks, ... Therefore, we need more time to solve on the lowest level and consequently also for the total solve, as shown in Fig. 42 —. As expected, the setting time (matrix building times) is reduced as the number of MPI tasks is increased (Fig. 42 - - -).

For cases two and three, we use the same number of levels (which is 8 levels) and the same lowest level (1024 DoF). As expected, we have an improved performance for both cases. But, the solution time • • • and the setting time * * * increased for a large number of MPI tasks (more than 128 MPI tasks for the setting and 256 MPI tasks for the solution time) because too many work accumulates on each single MPI task after the data are gathered. We have a good performance improvement for case three up to 512 MPI tasks, as shown in Fig. 42 ○ ○ ○ and ◊ ◊ ◊.

From Fig. 42, we conclude that the 2D parallelization algorithm has to be used on a large number of MPI tasks and after gathering the data the serial algorithm may be used on mid-range numbers of MPI tasks.

Next, we consider the strong and weak scaling properties of the third algorithm which uses a 2D parallelization and/or serial version on the coarsest grid level. We use a multigrid preconditioner with a Gauss-Seidel smoother for the PGMRES solver which has the best performance among several solvers. In Fig. 43 we consider a strong scaling for six cases, with 2048^2 DoF (—), 4096^2 DoF (•), 8192^2 DoF(+), 16384^2 DoF(○),32768^2 DoF(□) and 65536^2 DoF(×) on the finest level, respectively. For each case, the solution time and relative setting time have a good strong scaling property up to 8192 MPI tasks. Here, we also consider the 2D parallelization from the finest level and plot in Fig. 43 in red for three large cases. We can see that for a relatively small number of MPI tasks the performance is very similar, but a 2D parallelization starting at the finest level has a better scaling property on large number of MPI tasks.

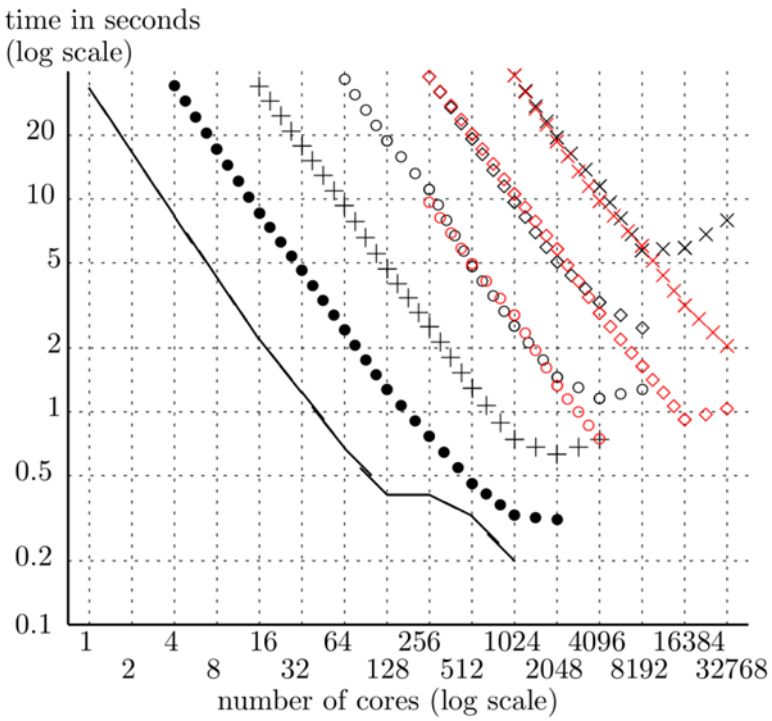


Fig. **43** (Strong scaling) The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver as a function of the number of cores.

In Fig. 44 we consider a weak scaling for nine cases with 16384 DoF(—), 32768 DoF (•), *), 65536 DoF (+),131072 DoF (○), 262144 DoF (◊), 524288 DoF(×), 1048576(∆) DoF, 2097152(□) DoF, and 4194304 (- - -) DoF per MPI task on the finest level, respectively. Over all cases we have a good weak scaling property, especially, for large numbers of DoF per MPI task which is the typical property of the parallel multigrid algorithm.

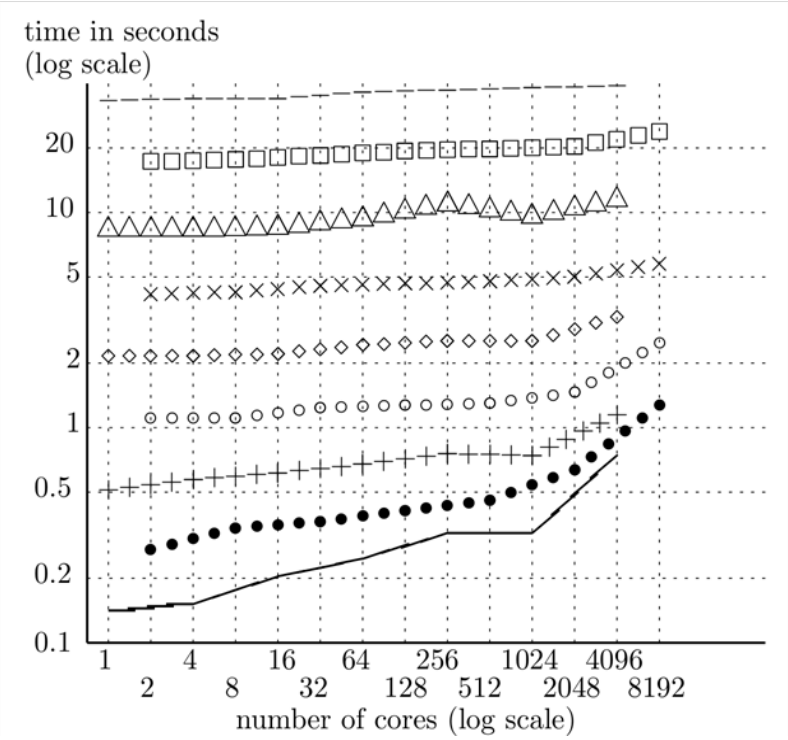


Fig. **44** (Weak scaling) The solution time of the multigrid method in seconds with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver as a function of the number of cores. The curves show cases of different DoF (see text).

5.5. *Conclusions and outlook*

We developed and tested multigrid algorithm classes for 2D problems. Especially, we compared three different types of memory allocation and selected the memory pre-allocation as it had the best performance. The numerical results show that the multigrid method has good strong and weak scaling properties up to more than 16K MPI tasks.

The BOUT++ team finished the 2D parallelization with OpenMP and on our side we finished the OpenMP/MPI hybrid multigrid solver. One has to consider the benefit from multithreading in comparison to the penalty overhead time for using the threads in the OpenMP/MPI hybrid code. So, one should further investigate which setting is the optimal in the hybrid algorithm.

For the Poisson problem with constant coefficients, the basic solver (FFTW) in the BOUT++ has better performance than the multigrid solver. However, we expect this to change for complicated problems. To confirm this, the project coordinator would need to construct a reasonable but demanding example to compare both solvers.

In addition, we prepared the implementation of the multigrid classes for the 3D problem. The BOUT++ team is further investigating how the multigrid solver for 3D problems can be efficiently used. The multigrid classes for 3D problems might be finished in a follow up project.

6. Final report on HLST project FWTOR-16

6.1. *Introduction*

FWTOR is a Fortran 95 full-wave code which solves Maxwell's equations for the propagation of electromagnetic wave beams in tokamak plasmas using the *Finite-Difference Time Domain* (FDTD) method. FWTOR has been developed at the National Technical University of Athens (NTUA), originally as a serial code.

Subsequent HLST projects have assessed parallelization possibilities while obtaining an OpenMP-enabled version first ([F14]), and an MPI-enabled one afterwards ([F15]). While the first project improved execution speed of FWTOR by responding to the current technological necessity of exploiting shared memory based multi-threading, the second one enabled support for much larger cases by means of distributed memory parallelism. Notably larger sized test cases expose the limits of the current MPI version, in that computed results are stored to disk by means of (serial) Fortran-based formatted I/O. While not being a bottleneck in the serial version, I/O became such in the MPI version when running large cases (especially ITER-sized, which start being approachable).

The project resulted in an optimization of the existing MPI parallelization by reducing the latency impact. This was achieved by an aggregation of successive array exchanges and by trading off exchanges for computation via subdomain overlap. In addition, partial support for 3D communications was implemented in the HMF module.

Due to the limited time budget available it has not been possible to also upgrade the I/O from serial to parallel; the focus of the work has been instead on the optimizations shortlisted during the last project [F15]. Most of the changes are concentrated on the HLST communication module for FWTOR, named 'HMF'. The following sections describe in detail the ideas behind them, activities to achieve these goals, and performance experiments results.

6.2. *Aggregate Communication optimization*

The FWTOR source code contains locations where the ghost data of up to three arrays need to be communicated in no particular order. In the original setting this required three completely independent exchanges, also because of the unevenly sized arrays (the grid is staggered; see [F15]). One can observe that aggregating the three ghost communications into one can save two `MPI_Sendrecv` calls out of three, at the cost of using a larger exchange buffer. One can expect that a decrease in the number of MPI messages (while keeping the total data payload constant) shortens the overall execution time due to reduced latency impact.

We have generalized the interface of the original ghost exchange routines in the HMF module to support this 'aggregated' exchange, and implemented the internals; please see Section 6.6 for performance experiments results.

6.3. *Subdomain overlap optimization*

Currently the entire domain extends on a rectangular region and is being partitioned among the MPI ranks using process-local variables with global indexing; each MPI process updates its own subdomain at each step. Such a partitioning foresees no overlap between the owned subdomains, except for an extra 'ghost' layer of thickness *g=1* being allocated around the local subdomain delimiting variables, like e.g. in: `allocate(eys(lx_1-g:lx_2+g,lz_1-g:lz_2+g))`.

This allows the time step computation to update the cells that are local within the subdomain and still access locally the data computed at the previous time steps on neighboring subdomains. In order for the ghost layer to be valid, it has to be fetched before each time step from the neighboring subdomains using MPI communication.

Now let's consider using a thicker ghost layer, say *g=(1+e)>1,* and after the arrays' allocation, adjusting the FDTD subdomain variables to extend by *e*, and to initialize this larger subdomain locally. This introduces an overlap of thickness *e>0* between the local ghost region and the neighboring process subdomain. After computation of the first time step, one might update the index variables so to shrink the *current* subdomain rectangle on each side by one unit (without changing the array allocation status), and use this perimeter from the previous step as 'ghost area'. This allows skipping costly MPI ghost data communication across adjacent subdomains. This *borders shrinking* step can be performed *e* times without the need of MPI communication; then the 'original' (non-overlapping) values for the subdomain limits are reached, and each subdomain would have to communicate its entire *g*-thick inner border to the neighbors around. This would revalidate their now *g*-thick 'ghost' area for further *e* steps without communication.

With this technique, after *g=1+e* time steps only one ghost communication of 'thickness' *g* would be necessary. This spares *e* ghost exchanges every *g* time steps, so a fraction of:

e /(1+e), that is (*1/2* for *e=1*, *2/3* for *e=2*, etc ...)

in the messages count, with roughly the same overall communicated data volume. The trade-off is that we also have to extend FDTD computation to the additional 'overlapping' subdomains, with the additional complication that the current subdomain area changes at each time step. In regimes where the ghost communication time is dominated by latency (when the per-process data amount is small) this leads to smaller MPI communication costs. Another precondition for efficiency is that the extra computation cost shall also be negligible, thus requiring the overlap area to be proportionally small with respect to the rest of the subdomain.

Fig. 45 and Fig. 46 give an example of applying this technique on a 11x6 grid with 3x2 processes.

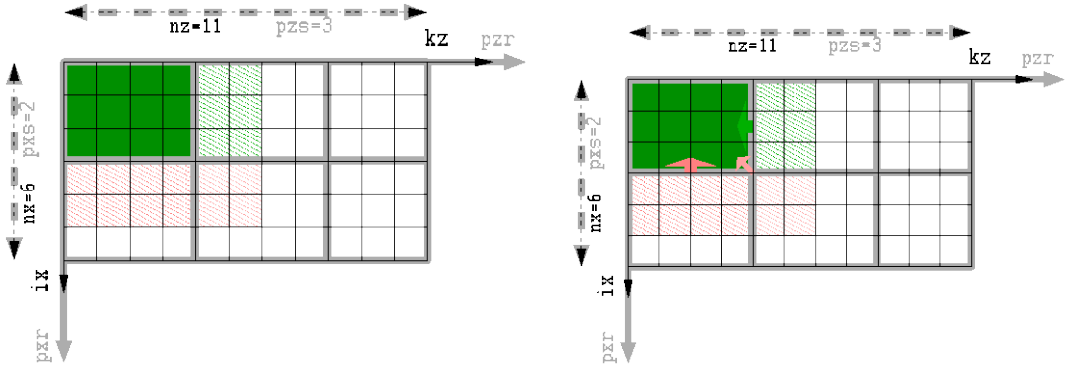


Fig. 45 Overlap technique on an example grid of 11 x 6 (*nx* x *nz*, in black) domain distributed across a 3 x 2 (*pxs* x *pzs*, in gray) process grid. In green, one process subdomain with a *g*=2 thick ghost layer (left). At initialization time, this data has to be either computed locally or to be communicated from neighbors (right).

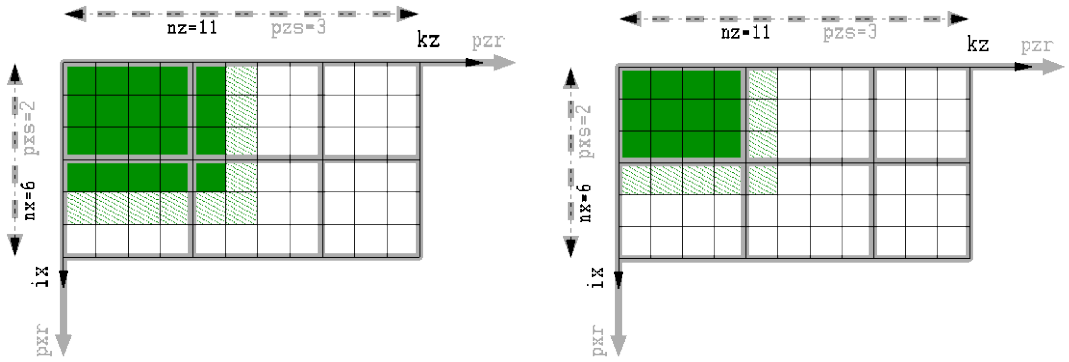


Fig. 46 Overlap technique. Left: a first time step update can be extended by one (*g*-1) beyond the own subdomain, because the outermost cells contain valid data from the previous step and can act as ghost (pattern). The second step (right) cannot use the outermost cells because they are not anymore up to date, however the original subdomain can be updated entirely. Before a third step a *g*-thick exchange will be necessary.

Please see Section 6.6 for performance experiments results related to this optimization.

6.4. *Extension to 3D physics*

A version of FWTOR with 3D physics is being developed at NTUA using the 2D serial version as a basis. It is expected to be computationally more demanding than the 2D version, so its parallelization would be highly desirable. On HLST's side, a restructuring of the HMF module began to accommodate exchanges of either 2D or 3D arrays by invoking the original existing communication primitives on the FWTOR side. For now we keep a 2D MPI domain distribution and a further 1D OpenMP level computation distribution. The technical means to achieve this include preprocessor macros and modern Fortran.

In order to proceed with this restructuring it was necessary to have access to a full reference serial 3D implementation. Unfortunately due to unforeseen workforce shortage on the PI group side, this could not be provided to us in time, so we had to keep the 3D functionality limited to the HMF module.

6.5. *Parallel I/O in Checkpoint-Restart*

The current FWTOR saves results using serial formatted Fortran I/O, which is not efficient for large datasets. It also lacks a restart feature. The MPI version of FWTOR enables the use of supercomputers to run cases much larger than in the past, eventually still requiring long run times; this while computing centers usually impose job runtime limits of e.g. 12 or 24 hours. Moreover, the larger the run, the larger the chance of a hardware failure leading to a job crash; and unless data residing in a node memory is periodically secured to the file system, it is at risk of loss.

All of the above motivates the need for an efficient checkpoint-restart (or, 'save and load' of simulation data) feature. The HLST has already experience with similar requirements, and addressed them in the 'ADIOS' series of projects [HAC]. A deliverable outcome of these projects has been the HAC ('HLST-ADIOS-Checkpoint') Fortran module giving trivially easy access to the functionality of the ADIOS library, and is currently used by e.g. the GENE code. We deemed this solution to be viable in FWTOR as well and the Project Investigator (PI), Christos Tsironis agreed to provide a proof-of-concept checkpoint-restart enabled serial version of the code, so to identify the overall impact and necessary helper functionality. Unfortunately, due to unexpected manpower shortage in his group he could not meet this goal in time. This means a project extension or a separate effort shall address this aspect in the future.

6.6. *Performance experiments*

The optimizations mentioned in sections 6.2 and 6.3 aim at mitigating the communication throughput loss at high latency regimes, namely when many nodes are being deployed to run smaller cases. We have made experiments in measuring ghost communication times in order to quantify the benefit of these optimizations (HMF side only).

From the following picture, one can see that ghost exchange can be improved by a factor of up to 8x on 512 nodes on the 1812-sided square grid. Considering that communication latency was dominating ghost exchange duration at these regimes, this improvement shall translate to a considerable savings in overall running times.

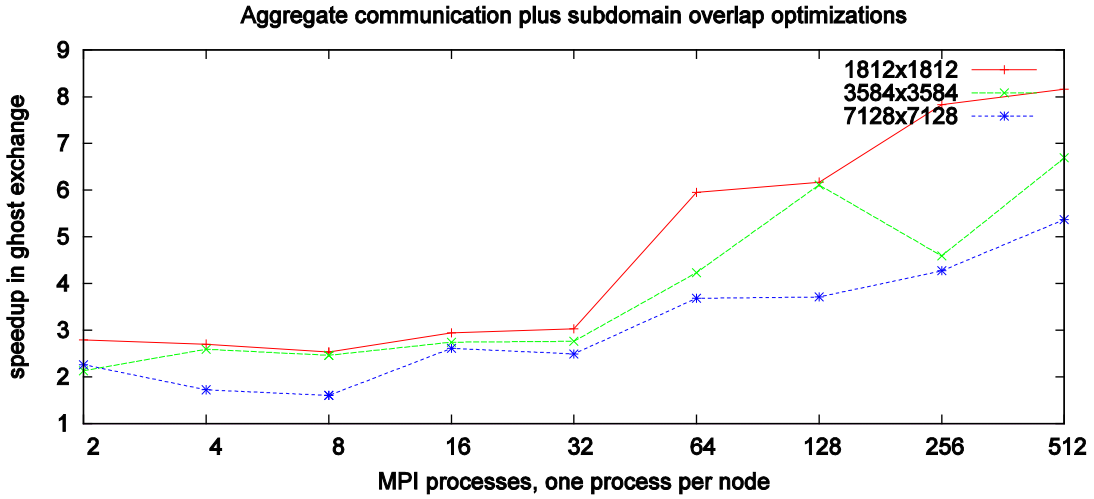


Fig. 47 Ghost exchange speedup when applying the aggregate communication and subdomain overlap optimizations. Note how highest latency loads (the smallest case, at maximum of the nodes count) improves the most.

Breaking down the ghost exchange improvement to the two optimizations, one can get the following picture for the aggregate communication technique:

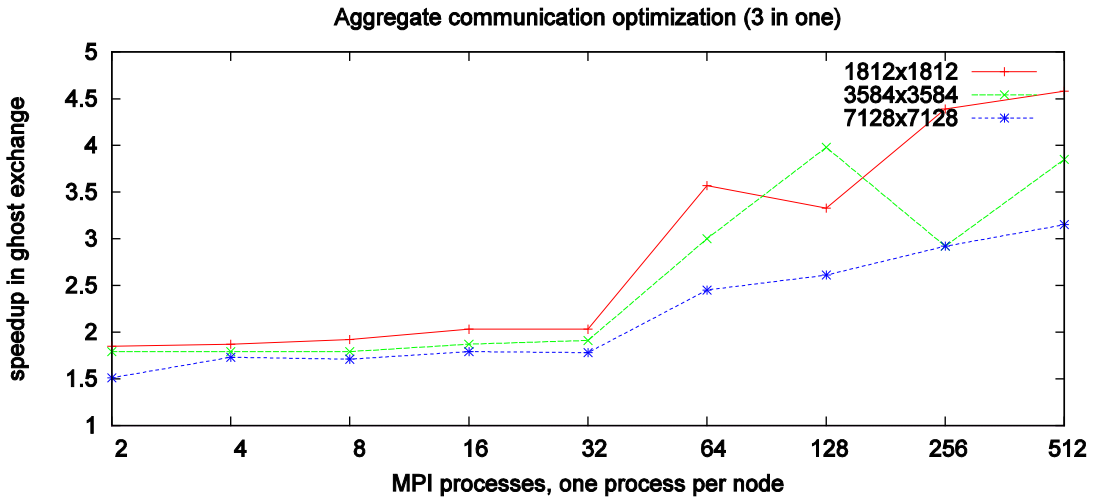


Fig. 48 Ghost exchange speedup when applying the aggregate communication optimization alone. This optimization applies by grouping similar staggered arrays ghost exchange (please see Section 6.2).

And the following one for the overlap optimization:

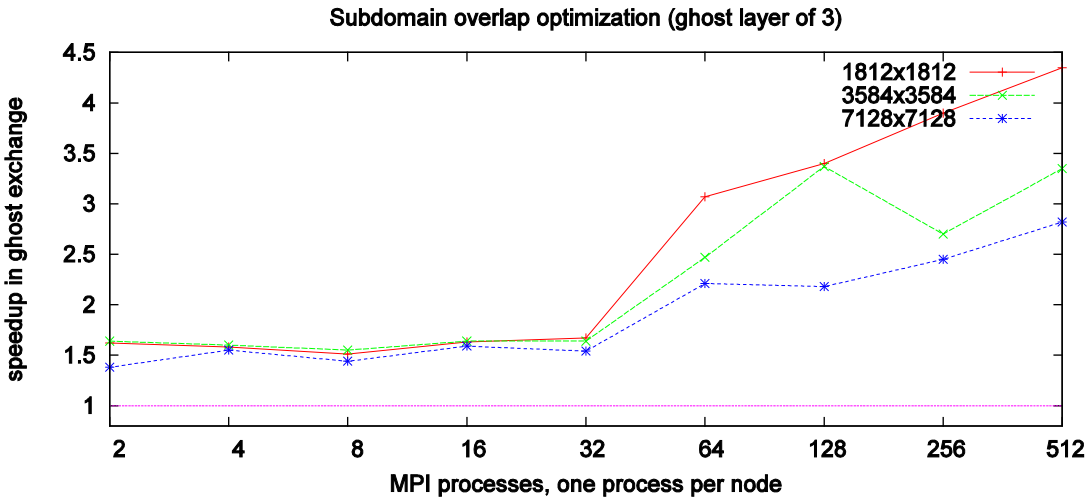


Fig. 49 Ghost exchange speedup when applying the subdomain overlap optimization alone. This optimization (please see Section 6.3) requires adjusting local subdomain indices at every step and has been therefore postponed.

Since both optimizations aim at reducing the number of MPI messages while enlarging their size, it is of no surprise that both contribute roughly to half of the improvement. The substantial difference in the nature of the two optimizations requires a few words of caution: a) While application of the aggregate communication optimization is easy and immediate at a code level, not all of the ghost exchanges can be grouped this way, but only a fraction of them. b) Regarding the subdomain overlap optimization, this can almost nullify necessity of MPI communication for several steps, but apart from requiring a certain amount of extra computations (recall Section 6.3), it also requires intervention on several variables in the HMF code itself. This step had been postponed due to shortage in the time budget; we recommend it to be applied in close collaboration with the PI.

Both mentioned optimizations are exploitable also in the 3D case, on the HMF side.

6.7. *Summary and outlook*

This project contemplated two categories of goals: those requiring reference implementations in serial as input from the project coordinator (PI) and those we could work out on our own.

Due to unforeseen workforce shortage the PI could not provide us in time neither with the necessary references for serial checkpoint-restart functionality nor with a 3D physics model. Nevertheless, we provide partial support for 3D communications in the HMF module. These communication routines have been added to serve as parallelization infrastructure for a 3D physics model to be developed by the PI. As the parallel I/O based checkpoint-restart functionality within our HAC module had to be postponed, we nevertheless advise it to be addressed with priority before planning a campaign of large jobs: I/O is the current performance bottleneck.

The HMF version we have obtained is capable of serving the post-2015 FWTOR code while supporting any combination of the two proposed MPI-related optimizations on the original 2D physics. The combined performance improvement of these optimizations can reach a factor of eight. To benefit from both optimizations, however, certain code adaptions are still necessary. These might be best achieved by means of an integrative HLST project addressing also the other pending interventions, in strict collaboration with the PI.

6.8. *References*

All reports are available from http://www.efda-hlst.eu.

[F14] *Final Report on HLST Project FWTOR* part of the HLST core team report 2014
[F15] *Final Report on HLST Project FWTOR* part of the HLST core team report 2015
[HAC] *Final Report on HLST Project ITM-ADIOS2* part of the HLST core team report 2014

7. **Final report on HLST project VIRIATIO**

7.1. ***Introduction***

VIRIATO is a hybrid fluid-kinetic code that solves the Kinetic Reduced Electron Heating Model: an asymptotically exact analytical reduction of gyrokinetics, obtained in the limit when the electron beta (the ratio of the electron internal energy to the magnetic energy) is comparable to the electron-ion mass ratio.

VIRIATO has already been object of an HLST project focused on parallel scalability improvement, assigned to Tiago Ribeiro in 2015; see [V15]. As a result, now larger runs are foreseeable, and I/O requirements (checkpoint/restart) have increased consequently.

Until now the checkpoint/restart functionality was being carried out serially via formatted I/O, gathering data in turn from each MPI rank. This method is not appropriate for large runs; in the project proposal document ([VP]) the authors report checkpoint times of several hours on a "256^4 x 200" case running on the Stampede supercomputer. This indicates a rather poor throughput in the order of MiB of array data per second.

The present project (VIRIATIO) aims at improving this by means of reusing a technical solution developed during HLST project ITM-ADIOS back in 2014 [A14], that is the *HLST ADIOS Checkpoint* (HAC) FORTRAN module. This module makes usage of the ADIOS I/O library, allowing I/O throughputs at a good rate on HELIOS' LUSTRE partitions, like e.g. ≈ 20 GiB/s on sufficiently large runs.

Section 7.2 describes VIRIATO and HAC; Sec. 7.3 describes our modifications to VIRIATO. We continue in Sec. 7.4 by analyzing the current scalability and performance properties.

7.2. ***VIRIATO I/O requirements and HAC***

Of the several dozens of arrays used by VIRIATO, only four need to be saved (*checkpointed*) in order to enable *restart* functionality. On a case with *np=256* points in all three directions and *nh=200* Hermite moments one can expect the fraction of the total allocated memory to be saved not to exceed ≈ 1/6 of the total one; this fraction decreases with *np*.

Typically, three of the mentioned arrays are sized np^3 while the fourth array size is $np^3 \times nh$. So the typical checkpoint can consist of $np^3 \times (3+nh)$ DOUBLE PRECISION elements being written, and the magnitude of *nh* can span between *1* and *np*.

The HAC module has been written for ease of use and maintenance on the application side, i.e. for the codes GENE, GEMZA, NEMORB (see [A14]). The application programming interface (API) of HAC is minimal: six subroutines only. A limiting aspect of HAC is that the dimension count of the supported arrays has to be chosen at compilation time. VIRIATO's checkpoint arrays are either all 3D or all but one, which is 4D. To cope with this VIRIATO saves the 3D arrays as 4D with one of the dimensions unitary.

7.3. ***HAC in VIRIATO***

As mentioned before, the typical checkpoint consists of $np^3 \times (3+nh)$ numerical elements (typically 8 byte DOUBLE PRECISION ones) being written. If *np=nh* then the one array sized np^4 would dominate the entire I/O, in that the other three arrays are much smaller. Our experience with the sizes in question shows that aggregating all four arrays into one 4D I/O array can avoid the smaller arrays I/Os from being latency-ridden and thus inefficient. This aggregation is possible because of the conforming size and distribution of the array dimensions. At restart time one "ADIOS checkpoint" is being read and the four arrays are recovered from the slices.

The overall impact on the VIRIATO code can be summarized as follows:

- the original load and save routines are being reused, just slightly modified in their signature and internals
- the HAC source listing is now included as part of VIRIATO sources; the ADIOS library instead has to be provided by the system side
- one entry has been added to the input file specified *namelist* to activate/deactivate the HAC based checkpoint: the original VIRIATO output format is always available
- use of HAC can be turned off at build time via a preprocessor switch

The amount of code impacted by our changes is small: a few hundred of code lines. The code of HAC is meant to be maintained by HLST; users are discouraged from modifying it.

HAC uses the ADIOS own file format ('.bp') to save data; the use of this format has proven to offer efficient operation on HELIOS. For interoperability with the other tools, as well as for longer term storage we recommend using a different format, like e.g. HDF5. A small program reusing the internal VIRIATO checkpoint/restart subroutines exemplifies how a converter or data post-processor might be written. Alternatively, conversion can be done with the **bp2h5** command line converter provided by ADIOS.

7.4. *Performance and scalability of VIRIATO + HAC*

We have run performance experiments to find a match with the findings of the ITM-ADIOS project (see [A14]).

We have chosen to test cases with as many Hermite moments (*nh*) as resolution points, and resolutions from 64 up to 320 in each direction. As a consequence 4D arrays are dominant, and one of them constitutes the major part of the checkpoint (but as mentioned in Sec. 7.2, the I/O data constitutes only a fraction of the entire memory occupation of a VIRIATO run). We ran our case using 16 MPI processes per node, on 1 up to 256 nodes (16 to 4096 MPI processes). Because of the memory constraint such a parameter range would hit against, we had to keep certain extremal combinations out; foremost the largest resolutions on fewer nodes.

We arrange results in four different plots to comment on the different performance factors.

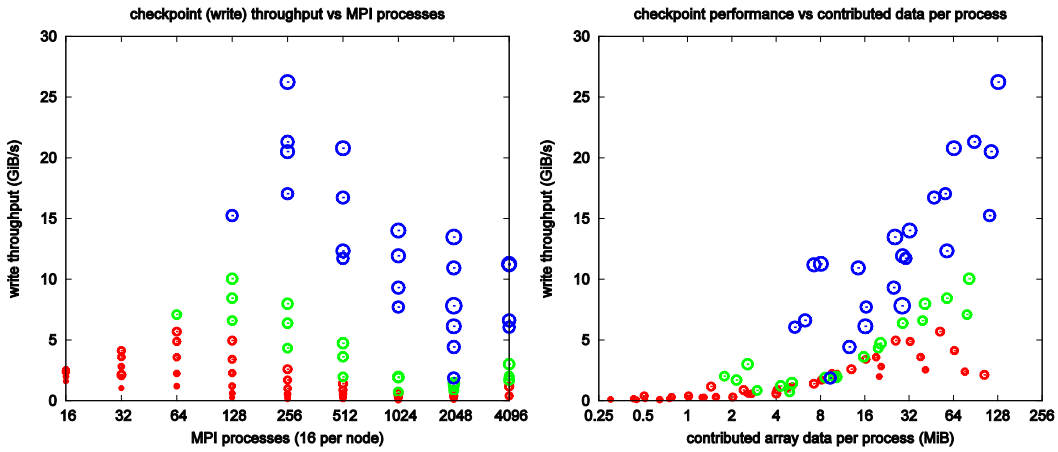


Fig. 50 Measured write throughput. For clarity, samples of small (red), middle (green), and large (blue) resolutions have different colors. On the left against MPI process counts, on the right against the amount of written array data contributed per process. Larger resolutions (here larger circles) led to more data being written and therefore higher throughput. Notice how the left plot resembles the mountain-like shape we have also encountered in our ITM-ADIOS investigation ([A14]).

The first plot in (Fig. 50, left) presents write throughput as a function of the MPI processes count. This reaches the ≈ 20 GiB/s which we have already encountered in

[A14], and then decreases when the I/O data per process ceases to be significant (as it is distributed among too many processes). The second (Fig. 50, right) arranges throughput on an axis with measured written data. This matches our past experience, where below a few dozens MiB per process latency dominates the I/O time.

In the third plot (Fig. 51, left) we can observe how larger resolutions, leading to more data usage, tend to yield higher throughput, but too many MPI processes (larger circles), although necessary to enable such cases, can make I/O latency-ridden and thus slow. Observed absolute elapsed checkpoint times (Fig. 51, right) did not exceed 7 s and are usually close to the time step computation time on the significant cases.

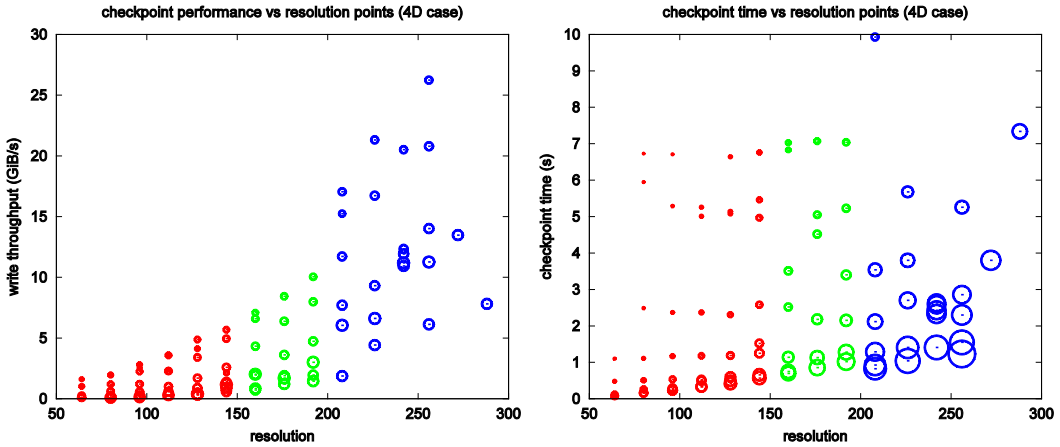


Fig. 51 Left: Measured write throughput versus resolution points signify more MPI processes (larger circles) are not a guarantee for higher I/O performance. Right: Measured checkpoint times did not exceed the range of a few seconds. Here the larger the circles, so the throughput.

A standard run of VIRIATO would checkpoint seldom, say, every hundred time steps, if not less. As long as this holds, the cost of checkpointing with HAC would be negligible. The speedup over what mentioned by the VIRIATO authors in the proposal [VP], in the range of hundreds to a thousand.

7.5. *Conclusions and outlook*

The original checkpoint / restart technique in VIRIATO was inefficient. Now, after having interfaced VIRIATO to our HAC module based on the ADIOS I/O library, checkpointing a case with 256 points in each direction with over 200 Hermite moments should take a few seconds and write at over 20 GiB/s to the LUSTRE file system of the HELIOS machine. This is a speedup approaching the order of a thousand over the original. Further experiments on the same machine align with our experience and findings with the ITM-ADIOS project ([A14]). The performance problem of checkpoint/restart in VIRIATO on HELIOS (and similar machines) seems to be resolved at the moment.

Since our solution uses the ADIOS file format for storing checkpoints, we recommend the VIRIATO users to convert to their own formats of choice for post-processing and archiving. Apart from the ADIOS-provided converter, they can use our converter example program.

In order to keep the HAC-based I/O of VIRIATO in good operation it is necessary for the VIRIATO users to keep an eye on the I/O statistics. Diagnostic information is being provided at each run. In case of e.g. incompatibilities requiring maintenance they are encouraged to contact the HLST.

7.6. *References*

[V15] *Final Report on HLST Project VIRIATO*, part of the HLST core team report 2015

[A14] *Final Report on HLST Project ITM-ADIOS*, part of the HLST core team report 2014

Both reports are accessible through http://www.efda-hlst.eu.

[VP] *Proposal for the Use of High Level Support Team resources*, VIRIATIO project

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

The CINCOMP project is dedicated to provide support for the European scientists who use the MARCONI machine located at CINECA. The MARCONI supercomputer was launched in July 2016 before an official production phase that was planned on mid-October 2016. During the months of pre-release phase both the hardware and the software of the system were tested by a variety of benchmarks. As it was expected for a new supercomputer many issues were found and reported to the MARCONI support team via the ticket system.

8.1. *The MARCONI supercomputer architecture*

The MARCONI supercomputer is located in Bolonia in the largest Italian computing centre named CINECA. The machine is planned to be gradually built during 16 months of operation, between April 2016 and July 2017, according to three series of upgrades. A preliminary system already went into production in July 2016, based on the Intel Xeon processor E5-2600 v4 product family (*Broadwell*). A total of 1512 computing nodes were assembled providing a computational power of 2 Pflop/s. The second upgrade should be fulfilled until the end of 2016, during which a new section equipped with the last generation of the Intel Xeon Phi product family (*Knights Landing*) based on the many integrated core architecture (68 cores) will be added. The new section with about 250 thousand cores will provide an additional computational power of about 11 Pflop/s. The last upgrade is planned for July 2017 with Intel Xeon *Skylake* processors. This update should allow to reach an overall computing power of about 20 Pflop/s. The European fusion community will have only access to the so-called MARCONI-Fusion partition of MARCONI.

8.2. *The Intel Broadwell processor architecture*

The Intel Xeon processor E5-2600 v4 with the code name *Broadwell* was chosen for the first configuration of the MARCONI supercomputer. Table 10 shows the comparison of the main hardware characteristics between Intel *Sandy Bridge* and Intel *Broadwell* processors, which were installed on HELIOS [1] and MARCONI, respectively.

Processor	**Intel Sandy Bridge**	**Intel Broadwell**
Number of cores	8	18
Memory	32 GB	64 GB
Frequency	2.6 GHz	2.3 GHz
FMA units	1	2
Peak performance	173 GFlop/s	633 GFlop/s
Memory bandwidth	68 GB/s	76.8 GB/s

Table 10 Hardware characteristics comparison between Intel *Sandy Bridge* and Intel *Broadwell*.

In spite of the lower CPU frequency of the Intel *Broadwell* the peak performance is about 3.6 times higher in comparison to the Intel *Sandy Bridge* due to the large number of the cores per CPU and the two FMA (fused multiply-add) units. The memory bandwidth increased also by a factor of 1.13.

8.3. *Memory bandwidth test*

The theoretical memory bandwidth according to the vendor specification for the Intel *Sandy Bridge* and the Intel *Broadwell* is 68 and 76.8 GB/s, respectively. However, this value is not reachable in practice, but we can measure the real memory bandwidth by means of different benchmarks. The Stream benchmark [2] is one of the most popular in the high performance computing community for doing so.

Fig. 52 presents the results obtained when the Stream benchmark was launched on a single node (two CPUs and two memory chips) of MARCONI. We found out that all four numerical benchmarks have a similar behaviour with a bit smaller bandwidth yielded for the *Scale* test. For this test a *compact* thread pinning (threads are mapped to neighboring cores as closely as possible) was used. The distribution is typical for NUMA node shared memory system [3]. The bandwidth grows up to 7 cores and saturates afterwards until all cores on one CPU are filled (18 cores). Including the second CPU and its memory bank (starting from the core number 19) the memory bandwidth starts to increase again up to 36 cores. However, the behaviour is not identical to the first CPU (it increases linearly and does not saturate). The reason arises from the fact that the Stream benchmark was developed in such a way that each thread performs the same amount of work and afterwards waits for the remaining threads during synchronization at an *omp_barrier.* A load imbalance can appear whenever there are threads which access the memory with different bandwidth values. Such situation happens for instance when we saturate the memory interface of the first CPU and use less than 18 cores from the second CPU. These cores will benefit from a higher memory bandwidth per core than cores on the first socket. Therefore, because the Stream algorithm is memory bound they will finish their tasks earlier, but will have to wait for the other 18 threads from the first socket at the *omp_barrier.* As result, the total bandwidth being consumed with each new additional core of the second socket increases gradually about 1/18 of the full bandwidth of a single socket [6], because each thread will have 1/18 less work to perform overall.

A maximum bandwidth of ~61 GB/s can be reached on a single CPU and ~118 GB/s on a single node. Such results are about 80% of the theoretical memory bandwidth specified by the vendor, which is a typical ratio for such a system.

As it was written above, the *compact* method of thread pinning was used for the benchmark presented in Fig. 52. Another method, named *scatter,* is also available in the OpenMP standard and can be used to pin threads alternatingly on distinct CPUs of one node. Initially, the PBS job submission system on MARCONI was ignoring such specification and always assigned threads as *compact,* even with the *scatter* pinning method involved. However, this issue was reported to the MARCONI support team and resolved accordingly.

The comparison of the Stream triad benchmark for the *scatter* and the *compact* pinning methods is shown in Fig. 53. For this test the Stream array size was chosen, as recommended in the Stream documentation, to consist of four times the (L_3) cache size of the computing node (90 MB). Up to eight cores (four on the first CPU four on the second) the scaling is almost linear because the memory interfaces are not saturated. When the bandwidth on both sockets is saturated, the number of threads does make a difference and a load imbalance occurs again whenever the number of threads on each CPU is different, which is the case when the total number of threads required is odd. In this situation, even though the threads on both CPUs are able to draw the full bandwidth from their sockets, there is less work to do for the second CPU, because there is one less thread, and the workload is divided evenly amongst all threads. Therefore, the threads on the second CPU finish their work before the threads on the first CPU and need to wait at the barrier which results in a small degradation in performance. When one more thread is required, thus resulting in a total even number of threads, then the load imbalance is lifted and small jumps up in performance arise [6].

However, an unexpected gradual bandwidth drop was also found (Fig. 53, red line). The bandwidth decreased from ~180000 MB/s using 16 threads to ~100000 MB/s with 36 threads. In order to prove the statement that both pinning methods should provide the same maximum bandwidth by using a full node (36 threads for a MARCONI node) the same test was performed on the HELIOS supercomputer and the results are shown in Fig. 54. As one can see, if one uses the full node (16 threads) the bandwidth is identical for both the *scatter* and the *compact* pinning, also

showing that the unexpected bandwidth drop on MARCONI does not exist on HELIOS. The test was then repeated on MARCONI (Fig. 53, blue line) using an array size of forty times the L_3 cache size for the *scatter* pinning method which yields the maximum bandwidth identical to the *compact* method. It was found that this holds when using the Stream array size larger than 20 caches. The reason why a smaller array size (as it is written in the Stream documentation) works on HELIOS and does not work on MARCONI is still under investigation. The issue was reported to the CINECA support team.

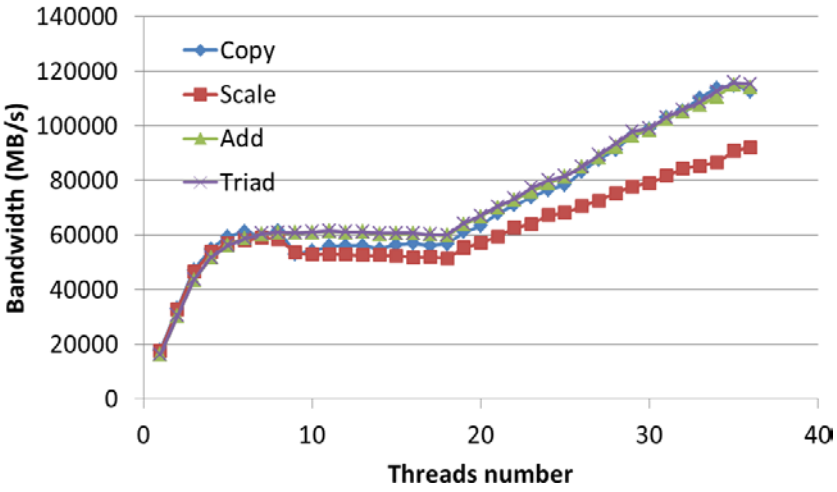


Fig. 52 Result of the Stream benchmark on a single MARCONI computing node.

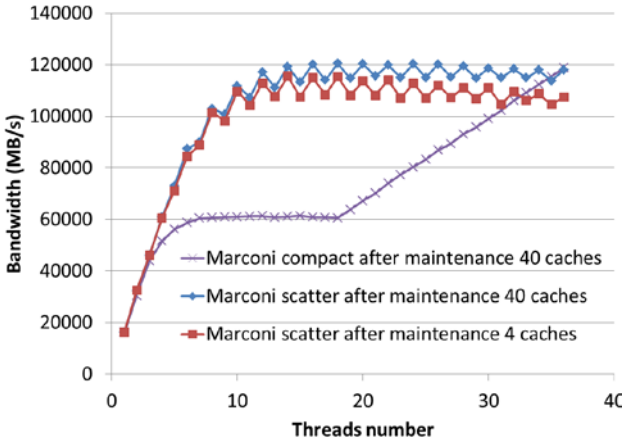


Fig. 53 Result of the Stream *triad* benchmark on a single MARCONI computing node for the *scatter* and the *compact* pinning methods.

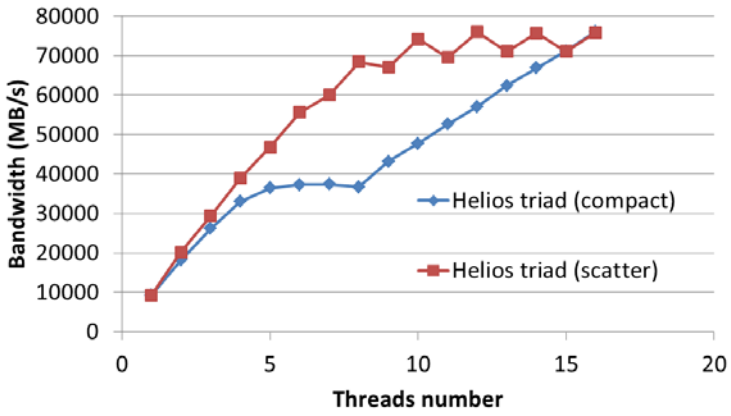


Fig. 54 Result of the Stream benchmark on a single HELIOS computing node for the *scatter* and the *compact* pinning methods.

We used also the Stream benchmark to compare the memory bandwidth between the computing nodes of MARCONI and HELIOS, respectively. The achieved memory bandwidths for the *triad* Stream test are shown in Fig. 55 for both machines. One can see that MARCONI's memory bandwidth of a single CPU (~61 GB/s, 80% of the theoretical) is about 1.6 times higher than the one for a single CPU of HELIOS (~38 GB/s, 56% of the theoretical). Almost the same factor (1.5) is yielded for a complete node (2 CPUs).

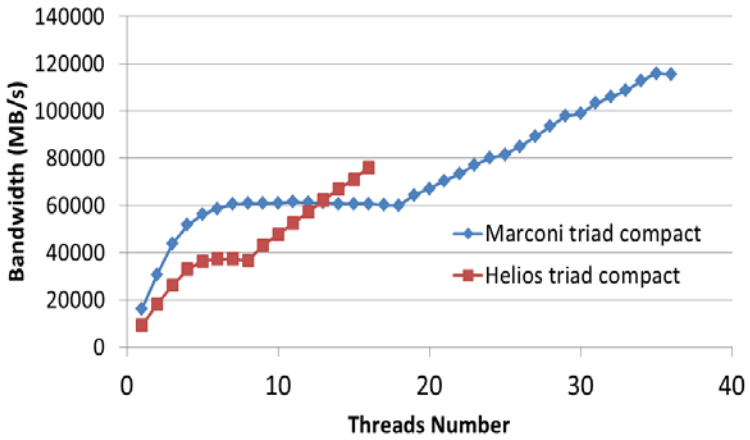


Fig. 55 Comparison of the Stream *triad* benchmark on a single MARCONI and HELIOS computing node (compact affinity).

8.4. *Roofline model*

The roofline model is an intuitive performance model that is used to estimate characteristic bounds (memory or CPU) of a code [4]. The performance is estimated as $P = \min\{P_{\max}; AI \times b_s\}$, where $P_{\max}$ is the applicable peak performance (Flop/sec), AI is the arithmetic intensity (Flops/Byte) and b_s is the applicable memory bandwidth (Bytes/sec). Fig. 56 shows the roofline model for a computing node on MARCONI using the theoretical peak performance (green line) and both the theoretical (blue line) and measured with the Stream benchmark (read line) memory bandwidth. One can see that on Intel *Broadwell* CPUs a code with up to eight flops/byte arithmetic intensity is memory bound. With higher arithmetic intensity the program will be compute bound due to the finite peak performance of the node. Comparatively, for the Intel *Sandy Bridge* CPUs installed on HELIOS the roofline model shows that code becomes compute bound after ~4 flops/byte.

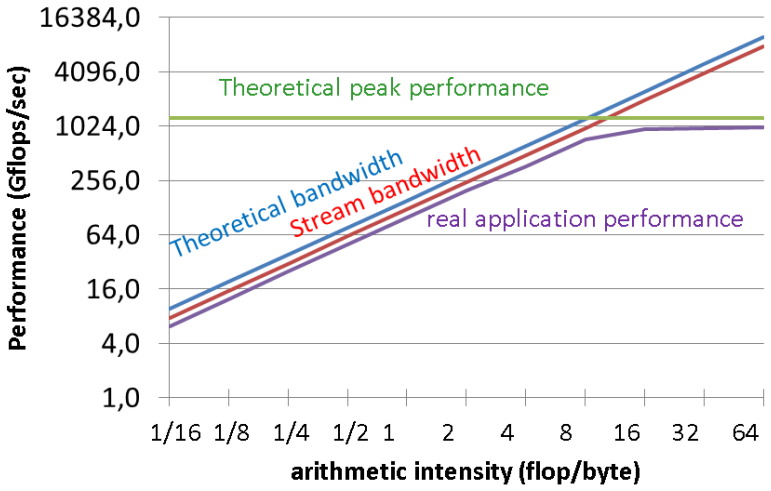


Fig. 56 Roofline model for a MARCONI computing node.

8.5. *Peak performance test*

From the section above we understand that in order to calculate the real peak performance we have to use a code which performs more than eight flops per byte of memory transfer. Therefore, we developed a simple test code which executes arithmetical operations (add and multiply) inside a large loop. We made a parametric study for different arithmetic intensities and calculated the performance of the code (Fig. 56, violet line). The performance increases gradually with increasing arithmetic intensity up to eight flops/byte as we expected since we are in the memory bound region. Afterwards the calculation becomes compute bound and the performance saturates with a value of about 1 Tflop/sec. Thus, we were able to reach almost 80 percent of the theoretical peak performance of a MARCONI computing node.

8.6. *Scalability test*

The strong scaling of a compute bound calculation within one MARCONI computing node was tested next. For this purpose a simple test code that performs different arithmetic operations and calculates a global sum of a given array was developed. The amount of the arithmetic operations was chosen in such a way to be sure that our calculation is compute bound. The speed-up of such a calculation is shown in Fig. 57 for different array sizes. The results are also compared to the ones obtained from a HELIOS node and to the ideal scaling. One can see a very good scalability (close to the ideal one) for all tested array sizes when they are distributed over different numbers of MPI tasks. In some cases the scaling is even superlinear due to cache effects.

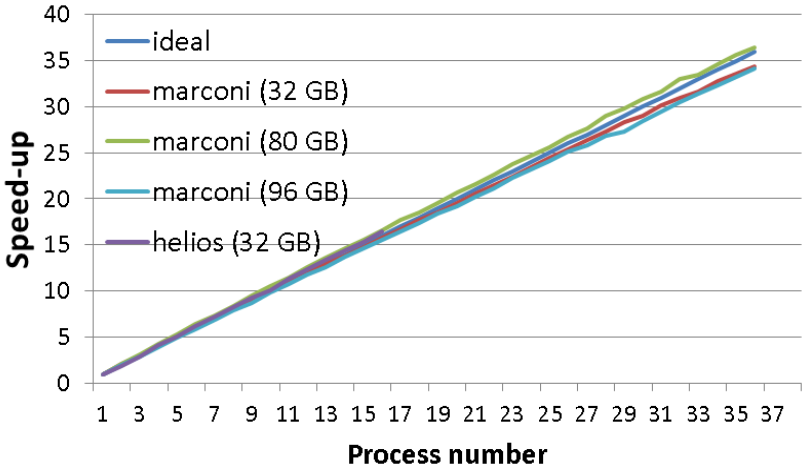


Fig. 57 Speed-up of the scaling test code versus number of MPI tasks within one computing node.

We have shown above that the strong scaling of a compute bound calculation works properly on MARCONI. The logical next step was to compare the performance (the wall clock time) on MARCONI and HELIOS. Fig. 58 presents the computational time of the scaling test code discussed above versus the number of MPI tasks. In spite of the higher CPU frequency of the Intel *Sandy Bridge* nodes installed on the HELIOS supercomputer, the code performance is between five to ten percent better on the MARCONI machine using the same number of MPI tasks. Such benefit is due to the two FMA units of the Intel *Broadwell* CPU in comparison to one on the Intel *Sandy Bridge*. Moreover, by using a whole node for the computation the total wall clock time is a factor of about 2.32 smaller on MARCONI due to a higher number of the physical cores (eight on *Sandy Bridge* and 18 on *Broadwell*).

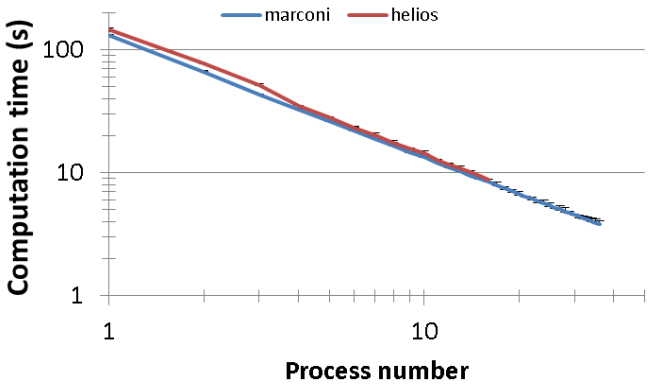


Fig. 58 Wall clock time for the scaling test code versus number of MPI tasks involved in the computation. Blue line represents results obtained on the MARCONI supercomputer; red line corresponds to the HELIOS machine.

8.7. *Porting the STARWALL code on MARCONI*

The next step was to transfer and run a real code on the MARCONI supercomputer. For this test we chose the STARWALL [5] code from the previous JORSTAR project. The compilation of the code was relatively easy and straightforward due to the convenient module system installed on MARCONI. However, the code crashed during runtime with the following error message: "*buffer overflow detected*". After debugging it was found that such an error appeared only for codes written in Fortran, in particular during the operation of reading from the standard input (*stdin*) or printing to the standard output (*stdout*). The issue, that prevented the execution of most Fortran codes on MARCONI, was reported to the support team. After contact with Intel support a temporary solution was proposed. By using an auxiliary environment variable named *FOR_PRINT* it was possible to redirect *stdout* to a file which finally allowed to launch our Fortran code on MARCONI. Afterwards the real cause of the malfunction was found to be in the interaction between the Intel 2016 compiler installed on MARCONI and the PID (process identity number) provided by the operating system. Each Fortran code that started on MARCONI and which got a PID higher than five digits crashed due to a bug in the Intel compiler. The solution found was to limit the PID on MARCONI to five digits, allowing Fortran codes to be launched on MARCONI without any additional setups and manipulations.

8.7.1. **Scalability of the STARWALL code**

After resolving the bug discussed above a strong scaling was performed using the STARWALL code. Fig. 59 shows the execution time on MARCONI and HELIOS versus a) the number of nodes and b) the number of cores. One can see that for the given problem size the minimum number of nodes needed is four on HELIOS and two on MARCONI (Fig. 59, a), due to the twice larger memory capacity of the *Broadwell* node installed on MARCONI (128 GB) in comparison to the memory of the *Sandy Bridge* (64 GB) installed on HELIOS.

For a small number of nodes <16 the performance on MARCONI is higher than on HELIOS due to the higher number of cores within the node and thus higher peak performance. Above 16 nodes the scalability on MARCONI saturates and the simulation time becomes larger for higher node numbers (Fig. 59, a), red and violet curves. The code scales much better on HELIOS where the simulation time decreases up to 64 nodes (Fig. 59, a) blue line.

We further checked if the amount of cores per node plays a significant role for the scaling on MARCONI having in mind that the HELIOS nodes have a total core number which is a power of two. Therefore, we performed a scaling on MARCONI using only 32 of the 36 cores available on each node, emulating a power of two scaling (Fig. 59, a) green line. The performance became better for 16 nodes in

comparison to the tests with 36 tasks per node; however performance degraded again for higher node numbers.

Another reason for the poor scaling on MARCONI could have been the different numbers of cores per node. Hence, the results presented in Fig. 59, a) were re-plotted versus the number of cores (Fig. 59, b). One can see again that using a low number of cores (<64) the performance on MARCONI is better than on HELIOS. However, starting with 128 cores the performance on HELIOS becomes better and scales up to 1024 cores. Scaling breaks already down with 512 cores on MARCONI. The reason of such a poor scalability detected on MARCONI will be explained in the next section.

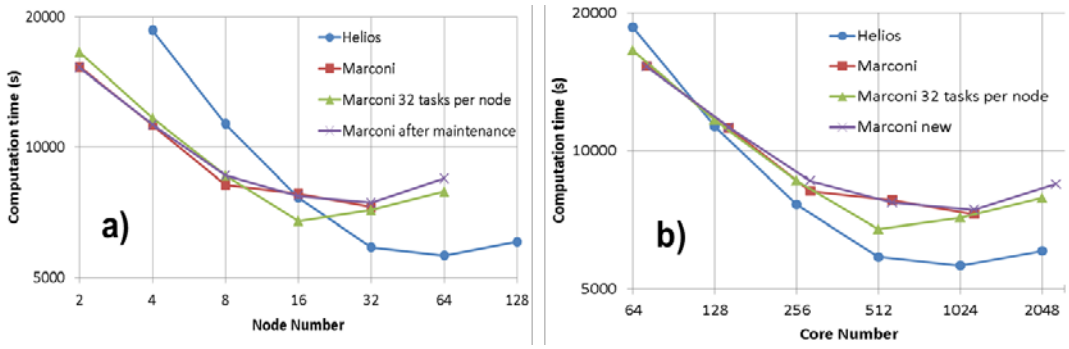


Fig. 59 Scalability of the STARWALL code on the MARCONI and HELIOS supercomputer a) versus number of computing nodes and b) versus number of cores.

8.8. *MARCONI network performance*

The PingPong test from the Intel MPI Benchmark suite [7] 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 *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 $time = \Delta t/2$, where Δt is the consumed time for the transaction from one process to another. The test is repeated 100–1000 times and averaged values are estimated. The latency is defined as the time spent to send a zero byte message. The bandwidth is estimated from the time needed to send a message of 4.2 MB between two processes.

First, the intra node network performance was tested on MARCONI and the results were compared with the ones obtained on HELIOS. The latency time between two processes running on the same CPU on MARCONI (0.61 µs) is more than two times higher than on HELIOS (0.25 µs). In spite of the different latency the memory bandwidth is very similar on both (9045 MB/s – MARCONI; 9130 MB/s – HELIOS).

When the two MPI tasks were pinned on the same node but on different CPUs, as expected, the latency increases for both machines in comparison to the single CPU results, because the message has to pass also the QPI (QuickPath Interconnect). However, the latency on MARCONI is still a factor of two slower in comparison to HELIOS (1.09 µs – MARCONI; 0.64 µs – HELIOS). The memory bandwidth decreases for both computers in comparison to the single CPU results but this effect is more pronounced for HELIOS (8537 MB/s – MARCONI; 6389 MB/s – HELIOS). Such results can be explained by a higher throughput (9.6 GT/s) of the new QPI for the *Broadwell* in comparison to 8 GT/s for the *Sandy Bridge* processor.

In the next step the inter node network performance was tested. For this test two MPI tasks were pinned on two different nodes. As expected the latency increases for both computers, namely to 1.45 µs for MARCONI and 1.14 µs for HELIOS, because a message should pass in addition through the Intel OmniPath interconnect on MARCONI or the Infiniband interconnect on HELIOS. Unexpected results were obtained during the inter node bandwidth test depicted in Fig. 60. The bandwidth on HELIOS decreases to 3200 MB/s in comparison to the intra node test results but the

distribution is as expected (Fig. 60, blue line) – a grow of the bandwidth with increasing message size until saturation is reached. For MARCONI a growth of the bandwidth is also seen up to 8 kB message sizes with a maximum bandwidth of 2393 MB/s. However, afterwards the bandwidth dramatically drops to 36 MB/s and slowly increases up to 350 MB/s for a message size of 4.2 MB.

The bandwidth test was repeated on MARCONI for different MPI libraries including Intel MPI 2015, Intel MPI 2016, Intel MPI 2017, OpenMPI and different communication fabrics (*tmi*, *shm:tmi*). It was observed that using the *shm:tmi* fabrics instead of *tmi* shifts the bandwidth drop to 32 kB (Fig. 61). The same shift was observed with OpenMPI. The reason might be that the OpenMPI library uses the *shm:tmi* fabric as default configuration.

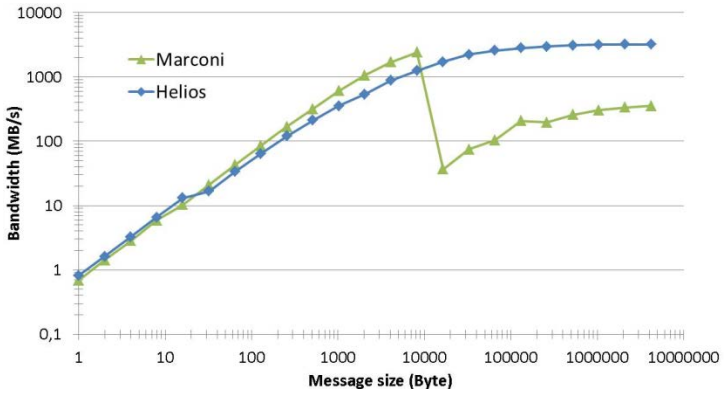


Fig. 60 Comparison of the memory bandwidth measured with the inter node ping pong test on the HELIOS and MARCONI computers.

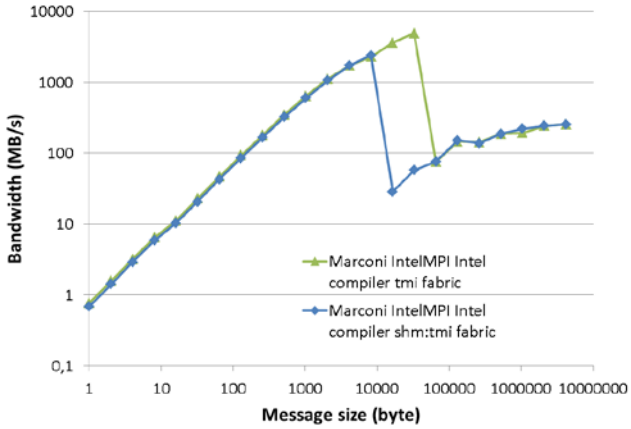


Fig. 61 Comparison of the memory bandwidth measured with the inter node ping pong test on the MARCONI supercomputer for different communication fabrics.

8.9. *Results after maintenance of 2016.09.30*

Many important updates were installed during a maintenance scheduled on 2016.09.30 for both the operating system and the internode network including the edge switches software. Therefore, we repeated all tests described above. The results are presented in this section.

8.9.1. **Inter node memory bandwidth**

After the maintenance the PingPong test from the Intel MPI benchmark suite was executed again to check the issue with the bandwidth drop described in detail in section 8.8. Fig. 67 shows a comparison of the memory bandwidth before (blue line) and after (green line) the maintenance. One can see that the unexpected bandwidth drop disappeared after the software of the supercomputer was updated. The memory bandwidth for the inter node connection can reach now a value of around 11304

MB/s which is more than three times higher than the inter node bandwidth on the HELIOS supercomputer (3209 MB/s).

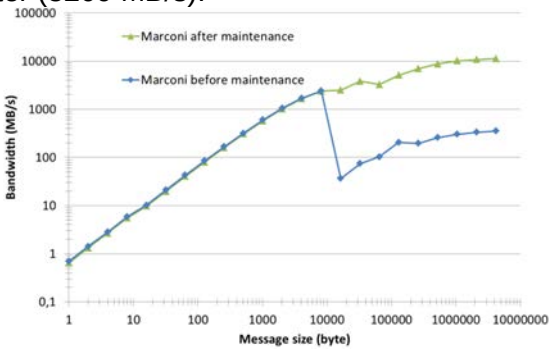


Fig. 62 Comparison of the memory bandwidth for the inter node PingPong test on the MARCONI supercomputer before (blue line) and after (green line) the maintenance.

8.9.2. **Intra node memory bandwidth**

The intra node memory bandwidth was tested after the maintenance as well. The results are presented in Fig. 68 for HELIOS a) and MARCONI b) supercomputers. On HELIOS the maximum bandwidth can be reached when two MPI tasks are pinned on one CPU inside one node (9158 MB/s – green line). Such a result is in good agreement with the bandwidth obtained from the Stream benchmark during a test with just one thread (9338 MB/s). The bandwidth decreases to 6372 MB/s when two MPI tasks are pinned on two different CPUs inside one node as we would expect. In such a case a message has to pass additionally through the QPI bus. The bandwidth decreases further when two MPI tasks are pinned on two different nodes (3209 MB/s) because the message has to go through the inter node connection.

Unexpected behaviour was detected during the intra node bandwidth test on MARCONI (Fig. 68 b)). When two MPI tasks are pinned on one CPU inside one node the bandwidth can reach a value of about 9295 MB/s that is only 57% of the bandwidth obtained from the Stream benchmark (16229 MB/s) during the test with just one thread. Moreover, the bandwidth increases when we move to a potentially slower connection. It reaches 9371 MB/s when two MPI tasks are pinned on different CPUs inside one node and even 11307 MB/s when two MPI tasks are assigned on two different nodes. We assume that the problem is with the software for the intra node communication because the bandwidth for the inter node connection seems to be reasonable. This issue was reported to the MARCONI support team.

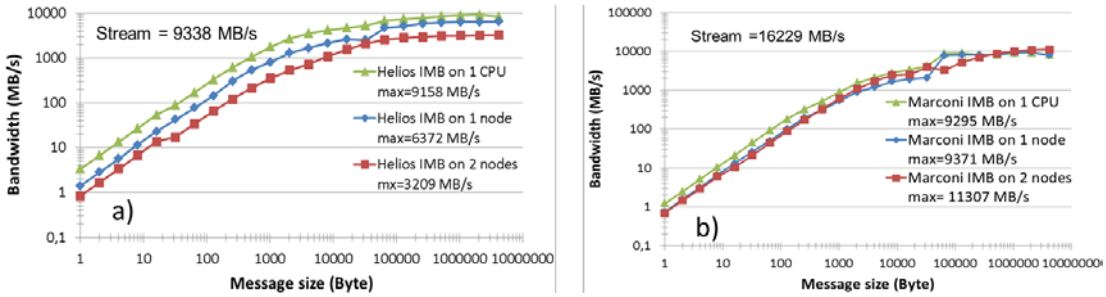


Fig. 63 Comparison of the inter and intra node memory bandwidth for the HELIOS a) and MARCONI b) supercomputer.

8.9.3. **Performance of the STARWALL code after maintenance**

After resolving the issue with the inter node memory bandwidth discussed in section 8.10.1 we repeated the test of the STARWALL code performance scalability detailed in section 8.7.1. Fig. 69 presents the execution time of the STARWALL code versus the number of nodes (a) and number of cores (b) after the maintenance that was

done on 2016.09.30. One can see that the computation time became much faster after the maintenance (blue and orange lines) in comparison to all previous tests on both HELIOS and MARCONI supercomputers. It is also worth to mention that for our calculations there was almost no difference between the execution time of the STARWALL code using 36 tasks per node (blue line) and 32 tasks per node (orange line).

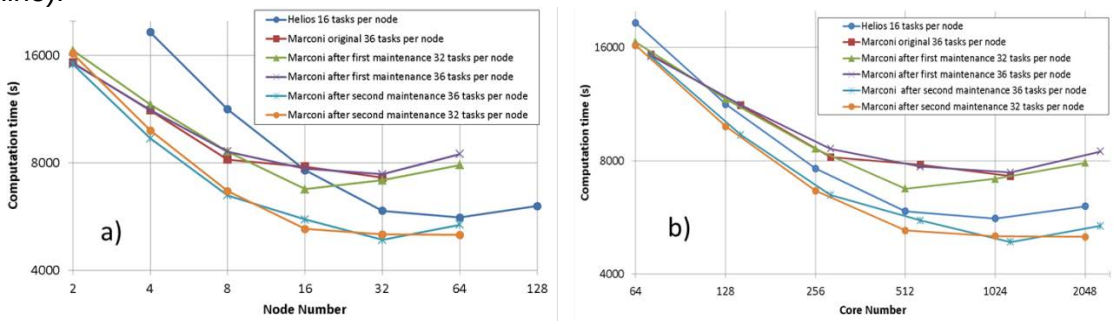


Fig. 64 Scalability of the STARWALL code on the MARCONI and HELIOS supercomputer a) versus number of computing nodes and b) versus number of cores.

8.10. *MPI libraries evaluation*

Fig. 71 shows the time needed to setup the MPI environment (a) and the total memory footprint per MPI task (b) for different MPI libraries. The initialization time is calculated using the formula from [9]: $t_{init} = t_{MPI\_init} + t_{all\_to\_all\_1} - t_{all\_to\_all\_<5:10>}$, where t_{MPI_init} is the duration of the *MPI_Init* call, $t_{all\_to\_all\_1}$ the execution time of the first *MPI_Alltoall* call and $t_{all\_to\_all\_<5:10>}$ the average execution time of the last five *MPI_Alltoall* calls. The obtained results are quite moderate. The initialization time is less than 11 s for all three libraries using the maximum possible number of cores on MARCONI-Fusion (8244). The distribution of the execution time for both Intel MPI libraries (2016, 2017) is very similar. It is more than a factor of two faster than for OpenMPI when using more than a hundred cores. The memory consumption during the initialization of the MPI environment is also moderate and takes not more than 45 MB per task for all three tested libraries.

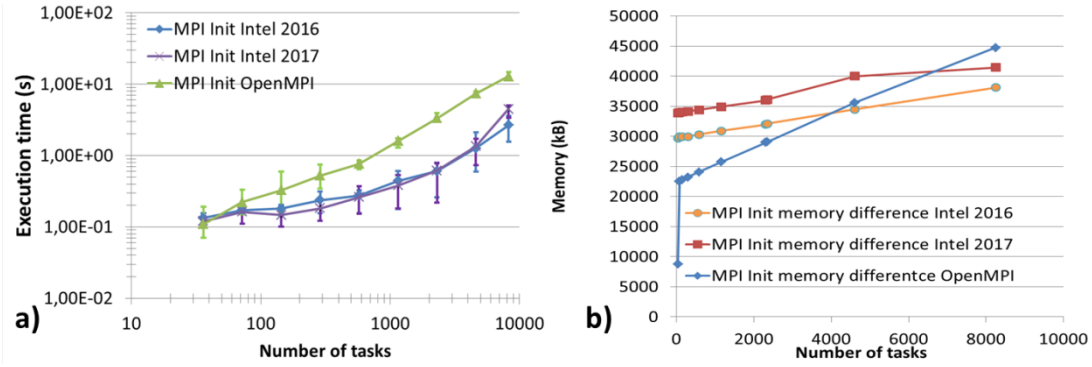


Fig. 65 a) *MPI_Init* initialization time and b) total memory footprint per task.

8.11. *Conclusions*

MARCONI went into operation in July 2016 as the new supercomputer for the fusion community. It should replace the outdated HELIOS machine at IFERC-CSC which will be decommissioned at the end of 2016.

Different benchmarks and tests were made to determine the performance of the new system. Issues were found that significantly limited its use. Some of them were resolved by the MARCONI support team, however others are still under investigation.

The Stream benchmark shows a good memory bandwidth inside a *Broadwell* node. The test code that computes the global reduction shows also good performance

scalability inside one node. However, the Intel MPI PingPong test for the inter node benchmarks had shown a drastic bandwidth drop for message sizes larger than 8–32 kB, which was finally resolved by the maintenance on 2016.09.30.

Scalability tests of codes like STARWALL and GYGLES [8] were also performed. It was found that in spite of remaining unresolved issues the performance on MARCONI is higher than on HELIOS for any node numbers.

8.12. ***References***

[1] https://www.iferc-csc.org/index.php/homepage
[2] https://www.cs.virginia.edu/stream/
[3] https://en.wikipedia.org/wiki/Non-uniform_memory_access
[4] Samuel Webb Williams, “Auto-tuning Performance on Multicore Computers”, Technical Report No. UCB/EECS-2008-164, December 17, 2008.
[5] S. Mochalskyy, M. Hoelzl, R. Hatzky, MPI Parallelization of the Resistive Wall Code STARWALL - Report of the EUROfusion High Level Support Team Project JORSTAR (2016).
[6] Private communication with Dr. G. Hager.
[7] https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[8] M. Fivaz, S. Brunner, G. de Ridder, O. Sauter, T.M. Tran, J. Vaclavik, L. Villard, K. Appert: Finite element approach to global gyrokinetic Particle-In-Cell simulations using magnetic coordinates, Computer Physics Communications, Volume 111, pp. 27–47, 1998.

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

Large scale plasma instabilities inside a tokamak can be influenced by the currents flowing in the conducting vessel wall. This involves non linear plasma dynamics and its interaction with the wall current. In order to study this problem the code that solves the magneto-hydrodynamic (MHD) equations, called JOREK, was coupled with the model for the vacuum region and the resistive conducting structure named STARWALL [1]. The JOREK-STARWALL model has been already applied to perform simulations of the Vertical Displacement Events (VDEs), the Resistive Wall Modes (RWMs), and Quiescent H-Mode.

At the beginning of the project it was not possible to resolve the realistic wall structure with a large number of finite element triangles due to the huge consumption of memory and wall clock time by STARWALL and the corresponding coupling routine in JOREK. Moreover, both the STARWALL code and the JOREK coupling routine are only partially parallelized via OpenMP. The aim of this project is to implement an MPI parallelization in the model that should allow to obtain realistic results with high resolution.

9.1. ***STARWALL code analysis***

It was important to determine the most critical data structures and subroutines that consume most of the memory and execution time before starting the implementation of the MPI parallelization. The memory consumption and the execution time for individual subroutines concerning different problem sizes can be controlled by tuning three knobs, which directly influence the problem size:

- Number of triangles within the plasma:
  ntri_p = 4**nv***n_points**2*(*n_R*+*n_Z*-2)
- Number of triangles in the wall: *ntri_w* = 2**nwu***nwv*
- Number of sin/cos harmonics: *n_harm*

We changed the problem size by varying the following parameters independently: (i) *n_R* and *n_Z* for *ntri_p*, (ii) *nwu* and *nwv* for *ntri_w*, and (iii) *n_harm.* A large scale production run should finally correspond to the parameters: *ntri_p*=$2*10^5$, *ntri_w*=$5*10^5$, *n_harm*=11.

9.1.1. **Memory consumption analysis**

Fig. 72 shows the memory consumption of the most important individual subroutines during the scan of the parameter *ntri_w* by varying the variables *nwu* and *nwv*. For this test case we fixed *n_harm*=1*, n_R*=*n_Z*=15, *nv*=32, and *n_points*=10. One can see that three subroutines (*matrix_wp*, *matrix_ww*, and *resistive_wall_response*) are the most memory demanding in this scan. Moreover, if we further scale our problem to a production size run with *nwu*=*nwv*=500 five additional subroutines (*matrix_rw, solver, dsygv, matrix_ew, matrix_pe*) will consume more than 50 GB memory. Therefore, all these subroutines must be parallelized in the final version of the code.

Fig. 73 represents the memory consumption of the same subroutines as it was shown in Fig. 72, however, this time with a parametric scan in the number of triangles within the plasma (*ntri_p*). In this test we kept the following parameters constant *nwu*=*nwv*=110, *n_harm*=1 but changed *n_R*=*n_Z*. The memory consumption increased mainly in three subroutines (*matrix_pp, matrix_wp, and matrix_ep*), which should be parallelized for a production run with *ntri_p*=$2*10^5$.

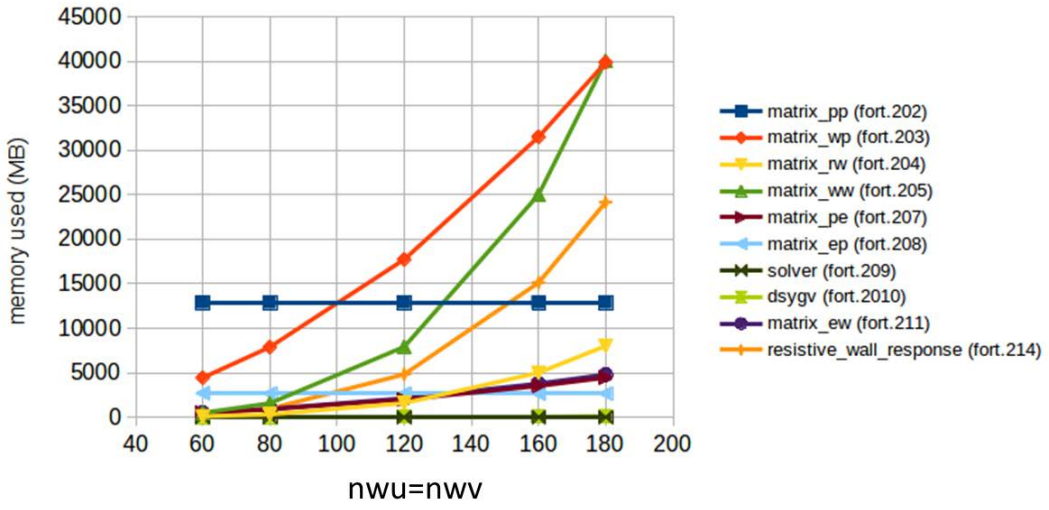


Fig. 66 The memory consumption of individual subroutines of the STARWALL code during the scan over the number of the triangles discretizing the wall (*ntri_w*=2**nwu***nwv*).

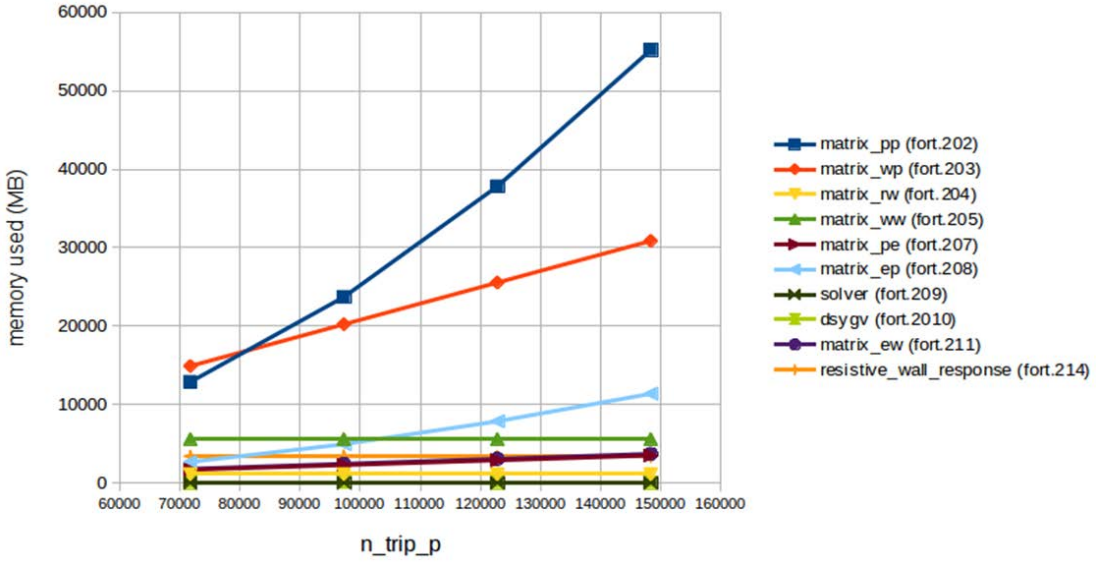


Fig. 67 The memory consumption of individual subroutines of the STARWALL code during the scan over the number of the triangles within the plasma (*ntri_p*).

The last parameter tested was the number of sin/cos harmonics (*n_harm*). Fig. 74 shows the memory consumption per subroutine versus *n_harm,* which varies from one to eleven. The value *n_harm*=11 corresponds to a production run. For this testcase we kept the following parameters constant: *nwu*=*nwv*=80, *n_R*=*n_Z*=15. All subroutines stay almost at the same level of memory consumption with only an insignificant growth for some subroutines. In order to prove that the number of sin/cos harmonics will not have a large influence on the memory consumption, whilst the number of triangles is increased, we performed an additional test with *nwu*=*nwv*=110. Indeed, as in the test above, the memory usage did not change much during the *n_harm* scan.

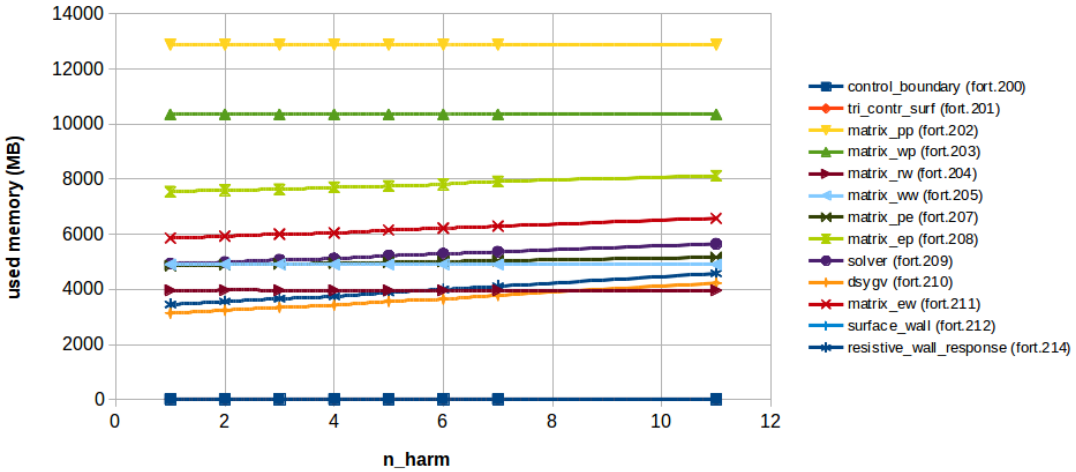


Fig. 68 The memory consumption of individual subroutines of the STARWALL code during a scan over the number of sin/cos harmonics.

STARWALL uses six subroutines (*dpotrf*, *dpotrs*, *dgemm*, *dsygv*, *dgetrf*, *dgetri*) from the linear algebra package LAPACK that is part of Intel the MKL library. It was important to check both, the size of the input matrices of these subroutines and the additional memory allocation inside the subroutines in order to determine if we should also replace these sequential subroutines by their parallel analogues. A dedicated script was developed for this propose, which measures the time spent executing the LAPACK subroutines and their memory consumption. It was found that only the *dsygv* LAPACK subroutine requires additional allocation of memory, which however, is negligible (~50–100 MB). Finally, the size of the input matrices for the production will range between 20 GB and few TB. Therefore, all LAPACK subroutines must be replaced by their parallel versions from other libraries like ScaLAPACK in order to distribute the input/output matrices, and hence reduce the size of the local sub-matrices.

Summarizing our tests, the complete STARWALL code must be rewritten in order to distribute the memory consumption. We estimated that the production run will require about six to seven TB of physical memory that can be allocated by using about 100 computing nodes of the HELIOS cluster.

9.1.2. **Computational time analysis**

The memory analysis has already shown the necessity of a complete domain decomposition of the whole code. Additionally, it was also important to determine the wall clock time for the production run and find the hot spots in the code. Fig. 75 shows the STARWALL execution time for different amounts of triangles in the wall and within the plasma (red and green lines). For a large scale production simulation on a single CPU the wall clock time would be in the range of a year.

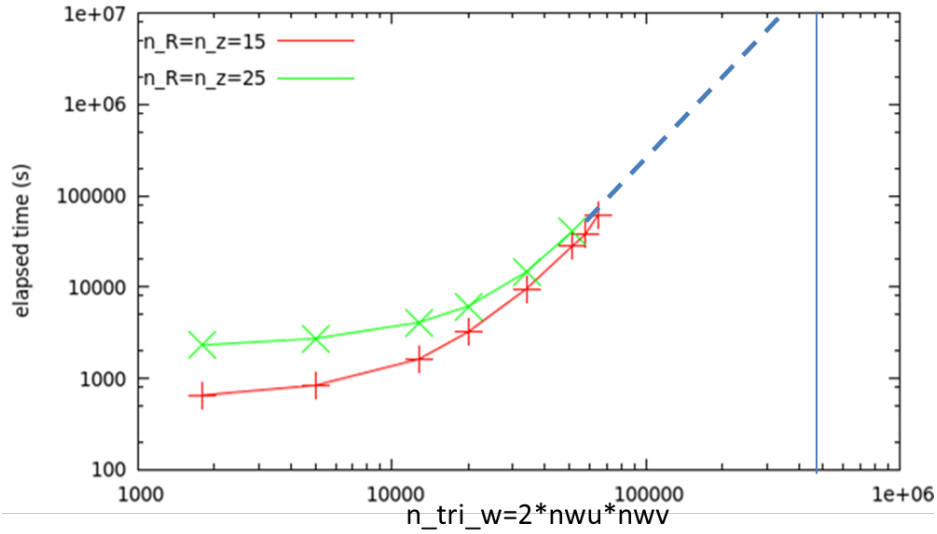


Fig. 69 The wall clock time versus the number of triangles in the wall (ntri_w) for different numbers of triangles within the plasma: *n_R*=*n_Z*=15 shown as red line, *n_R*=*n_Z*=25 shown as green line. The solid blue line shows the targeted numbers of triangles for a production run, while the dashed blue line presents the extrapolated scaling.

The next step was to determine the most time consuming subroutines in the code. This analysis was performed by means of the Allinea Forge profiling package. Depending on the problem size different subroutines contribute to a different percentage of the total execution time. However, among all subroutines, one (*dsygv*) consumes in all cases more than 40% of the total wall clock time. For the largest problem size we could run, the percentage was > 70%. Hence, this subroutine became the first candidate for parallelization effort and improvement.

9.1.3. **OpenMP parallelization analysis**

STARWALL is partially parallelized by means of OpenMP directives. Its parallelization efficiency is shown in Fig. 76. The wall clock time decreases by a factor of 1.4 when 16 threads are involved in comparison to the sequential run. Such poor performance can be explained by Amdahl's law, which shows the maximal possible speed-up of a program only partially parallelized. According to this law the maximal speed-up factor we can expect is around two. For this estimate we have taken into account that all LAPACK routines are sequential. With this assumption the sequential parts of STARWALL add up to about 45 percent of the total execution time.

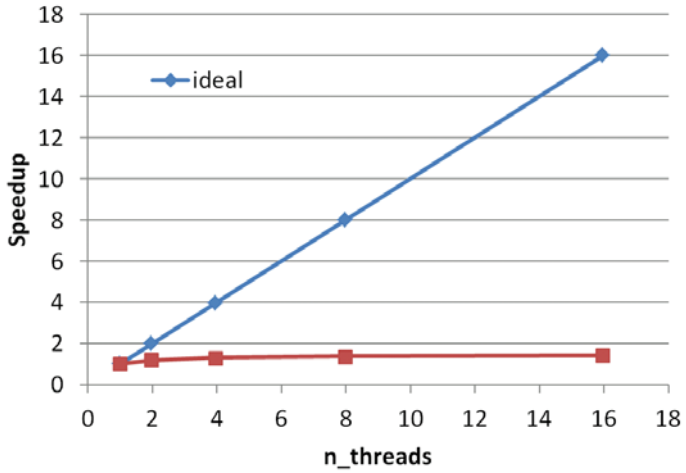


Fig. 70 Speed-up of the code versus number of OpenMP threads.

In order to confirm poor OpenMP parallelization scalability our model was checked via the Intel Vtune performance profiler. The basic hot spots analysis is presented in

Fig. 77. One can see that for most of the time only one thread is performing calculations (brown color), while the other 15 threads stay idle, as expected. Such results confirm the necessity of a replacement of all sequential LAPACK subroutines with their parallel analogues.

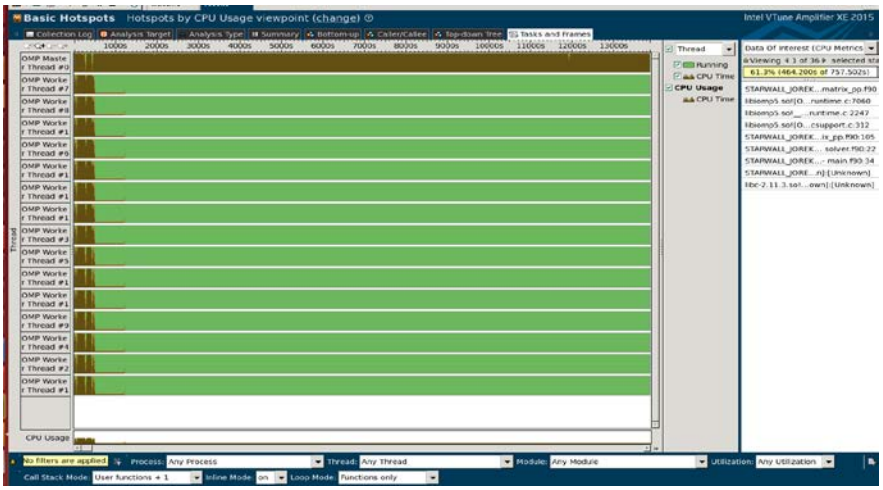


Fig. 71 Basic Hotspots analysis from the Intel Vtune amplifier using 16 OpenMP threads. Brown color shows the working status of the process, while green color corresponds to the idle state.

9.1.4. **LAPACK subroutines**

As it was discussed earlier the code spends most of the computational time in the execution of the LAPACK subroutines. In this subsection we summarize all LAPACK subroutines which are used in STARWALL:

- *dpotrf* – computes the lower-upper (LU) factorization of a tridiagonal matrix;
- *dpotrs* – solves a system of linear equations with a Cholesky factored symmetric positive defined matrix;
- *dgemm* – computes a matrix-matrix product for general matrices;
- *dsygv* – computes all eigenvalues and corresponding eigenvectors of a real generalized symmetric definite eigenproblem;
- *dgetrf* – computes the LU factorization of a general matrix;
- *dgetri* – computes the inverse of the LU factored general matrix.

9.1.5. **Bug check**

Before starting the optimization and parallelization the code was checked for correctness. The run time debugging was performed with two different compilers: *Lahey* and *Intel*. Afterwards the source code was also analyzed by the *Forcheck* static analyzer.

Three uninitialized variables were found that could produce unexpected behaviour of the code:

1) In file solver.f90: *nd_w=**ncoil**+npot_w*
2) In file matrix_ec.f90: alv=pi2***fnv***
3) In file resistive_wall_respones.f90: **ntri_c**

These problems were reported to the project coordinator and resolved afterwards.

The code was running mainly on a LINUX cluster called *TOK-P,* which is located at RZG, Garching. During parallel simulations a bug was detected in the standard input (*stdin*) system of this cluster. Within the default configuration only the process with *rank*=0 reads data from the *stdin*. Adding the flag *'-s all'* to *mpirun* should allow all processes being involved in the computation to read data from standard input. However, this flag was working only on a single node with all MPI tasks pinned. For tests with two or more nodes the code got stuck at the *stdin* reading. The same tests

were performed on *HELIOS* using the same compiler and compile flags. In this case the *std* reading worked properly. This bug was reported to the support team of the *TOK-P* cluster at RZG.

9.2. *MPI parallelization*

9.2.1. **Parallelization of the eigenvalue solver**

The LAPACK subroutine used for the calculation of the eigenvalues and the corresponding eigenvectors got the priority for parallelization. This subroutine consumes more than 70% of the total STARWALL execution time and uses two large matrices as input parameters. The subroutine is called *dsygv* and a more detailed description can be found in Ref. [2]. This subroutine was replaced by its parallel version *PDSYGVX* from the ScaLAPACK library that includes subroutines for linear algebra computation on distributed memory computers supporting MPI [3].

The PDSYGVX subroutine includes 34 input/output parameters by means of which the user can specify: the eigenvalue problem type to be solved, which eigenvalues and eigenvectors must be computed, the calculation precision, etc. Prior the calculation all global matrices must be distributed on process grid using a so called block-cycling scheme [3].

In order to test the correctness of the implementation of the *PDSYGVX* subroutine the calculated eigenvalues and the eigenvectors were compared with the results from the original (sequential) subroutine *dsygv.* Fig. 78 shows the calculated eigenvalues from both the *dsygv* (red points) and the *PDSYGVX* (green points) subroutines. In the case of the ScaLAPACK subroutines 16 MPI processes distributed over 16 computational nodes (1 per node) were used. A very good agreement was found for different problem sizes.

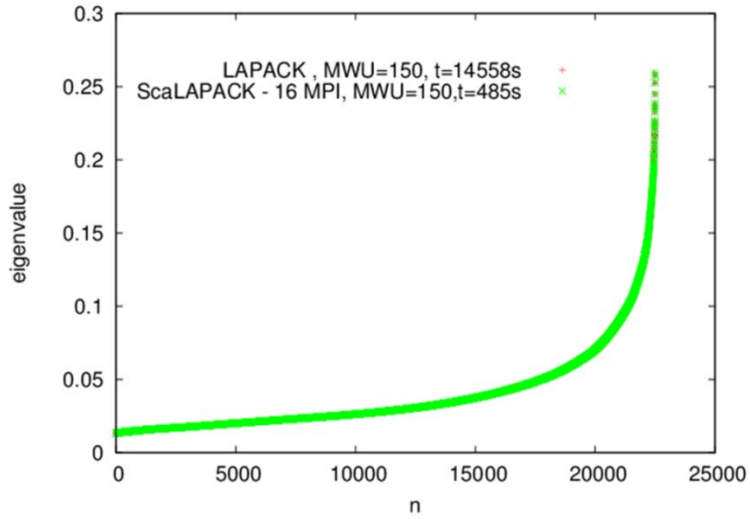


Fig. 72 Eigenvalues from the sequential LAPACK *dsygv* (red points) and the parallel ScaLAPACK *PDSYGVX* (green points) subroutine.

In spite of the perfect agreement of the eigenvalues the calculated eigenvectors are somehow unpredictable. For some problem sizes they are identical between the *dsygv* and *PDSYGVX* subroutine. In other cases some eigenvectors have the same length but point in opposite direction i.e. all their components are with opposite sign (Fig. 79 on the left). They are still correct eigenvectors as can be seen in Fig. 79 on the right, where the absolute values of all eigenvector components are shown. However, sometimes eigenvectors have even different values of their components. Such behaviour can be explained by a not unique solution of the eigenvector problem. If some eigenvalues are not distinct, i.e. the solution of the characteristic equation has multiple roots, we say that these eigenvalues are degenerated. Different bases of eigenvectors exist for these degenerate eigenvalues. Therefore,

LAPACK and ScaLAPACK can deliver different components for eigenvectors which correspond to degenerate eigenvalues, but they still represent the right eigenvector.

In addition, the correctness of the new subroutine was checked by a comparison of the physical solution for the eigenvectors from LAPACK and ScaLAPACK library. The STARWALL results were in very good agreement within an absolute error of 10^{-13}.

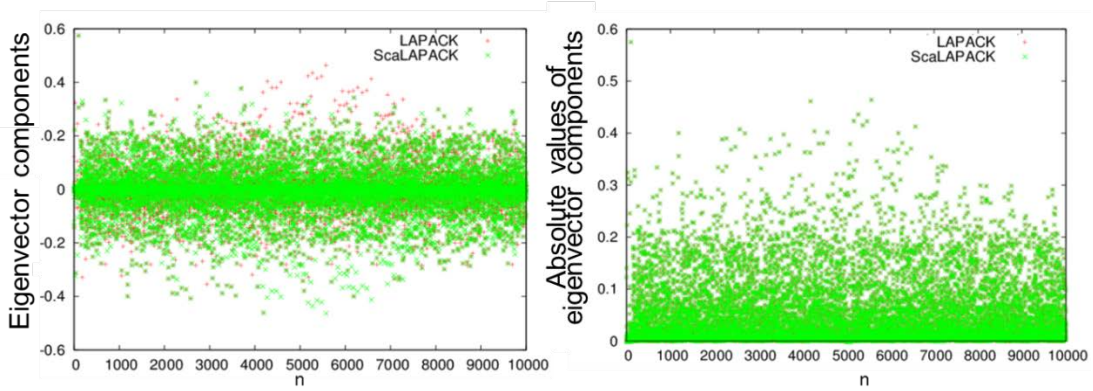


Fig. 73 Eigenvector components on the left, and their absolute values on the right, from the sequential LAPACK routine *dsygv* (red points) and the parallel ScaLAPACK routine *PDSYGVX* (green points).

The advantage of the ScaLAPACK library in comparison to LAPACK is that it benefits from the IEEE ±∞ arithmetic to accelerate the computations of the eigenvalue solver. Such improvement can be seen in Fig. 80 where the execution time of the ScaLAPACK subroutine *PDSYGVX* obtained from the simulations using one task is compared to the execution time of the LAPACK *dsygv* subroutine for different problem sizes. The ScaLAPACK solver works faster than LAPACK for all problem sizes and gains a factor more than two for large matrices.

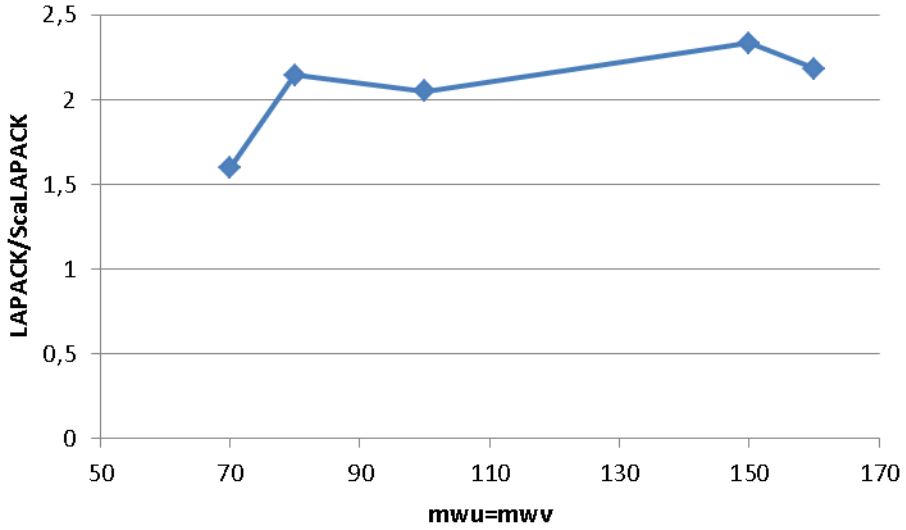


Fig. 74 Comparison of the eigenvalue solver execution time between ScaLAPACK using one process and the LAPACK library for different problem sizes.

The parallelization efficiency of the *PDSYGVX* subroutine is shown in Fig. 81 on the left for a small problem size (*nwu*=*nwv*=70) and on the right for large matrices (*nwu*=*nwv*=160). For an efficient ScaLAPACK performance the matrix size should be large enough relative to the amount of processes being involved in the simulation [3]. Therefore, the parallelization efficiency is almost saturated with 16 processes for a small problem size with an execution time of only a few seconds. However, when large matrices are used the problem scales almost linearly. An even better performance is expected for a production run in which *nwu*=*nwv*=500.

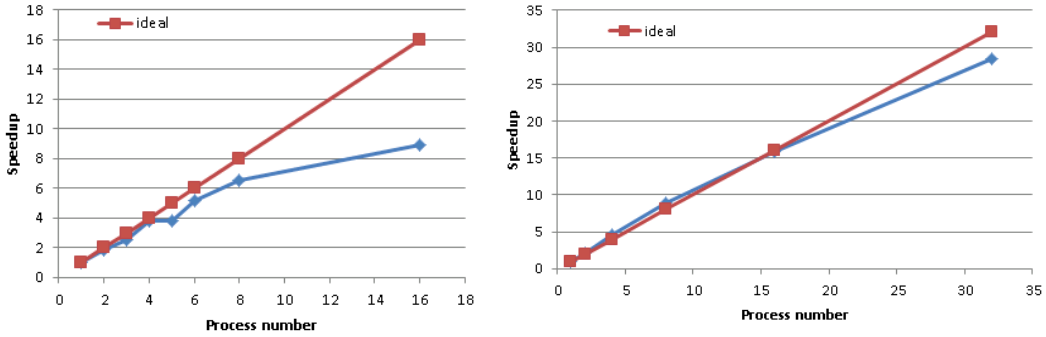


Fig. 75 *PDSYGVX* parallelization efficiency. On the left, small problem size with *nwu=nwv*=70*;* on the right, large problem size *nwu=nwv*=160*.*

9.2.2. **Parallelization of the *matrix_ww* subroutine**

The eigenvalue solver described above uses two large matrices (*a_ww(npot_w,npot_w)* and *b_rw(npot_w,npot_w)*) as input parameters. The size of these matrices for a large production run will be (250,000 × 250,000) that is 500 GB for double precision components. Therefore, these matrices have to be distributed over MPI tasks. We started the parallelization with the subroutine *matrix_ww* where the matrix *a_ww* is built.

In this subroutine the matrix *a_ww* is calculated from another matrix, which is named *dima(ntri_w,ntri_w).* The size of this additional matrix is even larger than the size of the matrix *a_ww,* namely (500,000 × 500,000), that is 2 TB for the double precision components. Thus, *dima* matrix must be also distributed over the MPI processes.

The original kernel loop that corresponds to the creation of the matrix *a_ww* is shown in Fig. 82. One can see that the indexes of the matrix *a_ww* and *dima* are not linked. The first one gets its indexes from the additional array *ipot_w* where values range from 1 to *npot_w,* while the *dima* indexes can run from 1 to *ntri_w.*

We tried to find some patterns between the *a_ww* and *dima* matrices in order to determine which components of the *dima* matrix will be used for calculating the equally distributed *a_ww* matrix. The *a_ww* matrix was distributed among 16 processors (Fig. 83 left). Each pink rectangle represents the global *a_ww* matrix, and the yellow rectangles depict the sub-matrices assigned to each of the 16 new tasks. The *dima* matrix indexes that were used to calculate the local distributed matrix *a_ww* are shown in Fig. 83 on the right. Now, the pink rectangles stand for the global *dima* matrix, whereas the yellow represent those indexes which are needed to calculate the local part of sub-matrices *a_ww* (yellow rectangles on the left figure). One can see that the *dima* components, which are used to build the distributed part of *a_ww* are not localized and spread across the whole matrix. Hence, it will be very difficult to efficiently distribute the matrix *dima*.

```
do i =1,ntri_w
     do k =1,3
            j = ipot_w(i,k) + 1
       do i1=1,ntri_w
        do k1=1,3
            j1 = ipot_w(i1,k1) + 1
          temp  = .5*(dxw(i,k)*dxw(i1,k1)                              &
                     +dyw(i,k)*dyw(i1,k1)                              &
                     +dzw(i,k)*dzw(i1,k1))                             &
                     *(dima(i,i1)+dima(i1,i))
           a_ww(j+ncoil,j1+ncoil) = a_ww(j+ncoil,j1+ncoil) + temp

        enddo
       enddo
     enddo
 enddo
```

Fig. 76 Original kernel loop that builds the matrix *a_ww*.

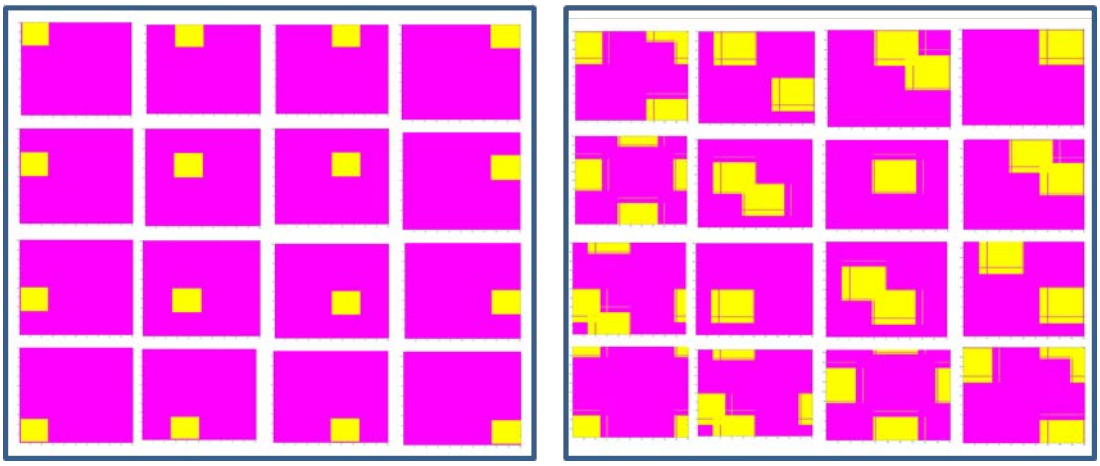


Fig. 77 Distributed matrix *a_ww* on 16 processors (left) and the corresponding indexes of the matrix *dima* that are used to calculate the local part of *a_ww* (right).

9.2.2.1. *Matrix free "dima" computation*

As the distribution of the matrix *dima* could not be performed efficiently, we decided to rewrite the code in such a way that components of the *dima* matrix will be calculated directly in the place where they should be used.

In the original code version the matrix *dima* was pre-calculated by means of the subroutine *tri_induct,* where three nested loops take place. If this subroutine would be straightforwardly implemented in the kernel loop (Fig. 82), where it has already four nested loops, computational time would be years even on computer clusters. Therefore, we split this subroutine in three parts: *tri_induct_1, tri_induct_2, tri_induct_3.* Two subroutines (*tri_induct_1, tri_induct_2*) are called outside the kernel loop and have no significant effect on the total computational time. Inside the kernel loop only one more nested loop with an index running over seven points was added. A code fragment of the new version of the kernel loop is shown in Fig. 84. One can see that the *dima* matrix is absent there. Instead, there is the function call *tri_induct_3,* where *the* necessary value of *dima* is calculated and stored in the variables *dima_sca* and *dima_sca2.*

The drawback of such a modification is the increase of the computational time. Fig. 85 shows the elapsed time of the kernel loop for different problem sizes using the old version of the code with the matrix *dima* and the new version with the *dima* free format. The computational time increases in about two times for all problem sizes. For a large production run with *ntri_w*=500,000 it was estimated to be around 111

hours on one CPU. The advantage is naturally the possibility to distribute the array and run in parallel.

```
do i =1,ntri_w
 do i1=1,ntri_w
   do k =1,3
      j = ipot_w(i,k) + 1
      ! If index is inside the local part of distributed matrix a_ww
      if (j>= j_loc_b .AND. j<= j_loc_e ) then
           counter=0
           do k1=1,3
              j1 = ipot_w(i1,k1) + 1
              if (j1>= j1_loc_b .AND. j1<= j1_loc_e ) then
                  dima_sca=0
                  dima_sca2=0
                  if ( counter<1 ) THEN
                       dima_sca=0
                       dima_sca2=0

                       call tri_induct_3(ntri_w,ntri_w,i,i1,xw,yw,zw,dima_sca)
                       call tri_induct_3(ntri_w,ntri_w,i1,i,xw,yw,zw,dima_sca2)

                        dima_sum=dima_sca+dima_sca2
                        counter=counter+1
                  endif

                  temp  = .5*(dxw(i,k)*dxw(i1,k1)          &
                         +dyw(i,k)*dyw(i1,k1)              &
                         +dzw(i,k)*dzw(i1,k1))             &
                         *dima_sum

                  a_ww_loc(j+ncoil,j1+ncoil) = a_ww_loc(j+ncoil,j1+ncoil) + temp

              endif
           enddo
      endif
     enddo
enddo
```

Fig. 78 Matrix *dima* free kernel loop that builds the matrix *a_ww*.

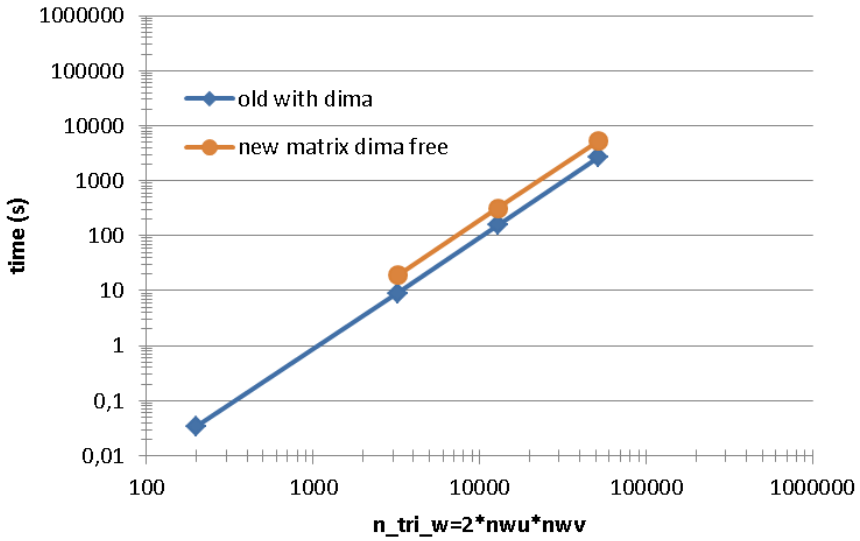


Fig. 79 Computational time of the kernel loop of the subroutine *matrix_ww* versus the problem size using the old code version (with *dima* matrix) – blue line and modified kernel loop (with *dima* free format) – orange line.

The next step was to check the parallelization efficiency of the kernel loop. This test is shown in Fig. 86. One can see that a speed-up factor of ~110 can be reached when 256 tasks are involved. Therefore, the computational time of the kernel loop without the *dima* matrix using 256 cores would be about one hour.

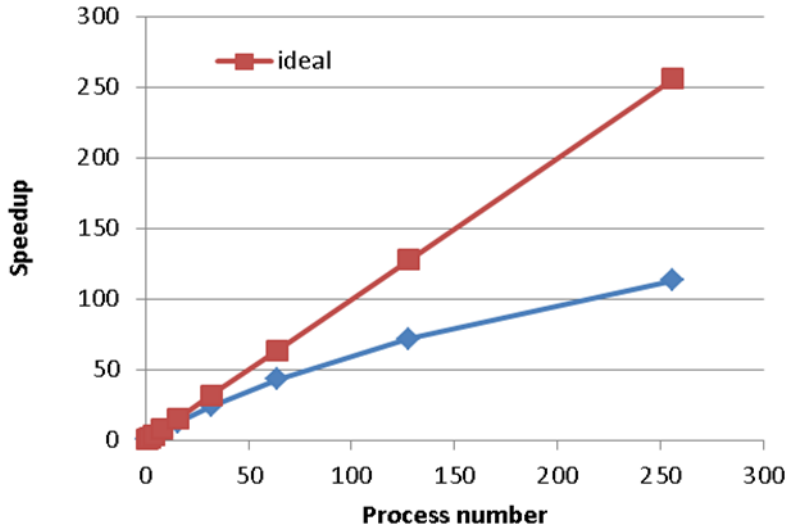


Fig. 80 Speed-up of the kernel loop versus number of MPI tasks.

9.2.2.2. Matrix free "dima" computation with ScaLAPACK indexing

In order to use the distributed matrices as input parameters for ScaLAPACK subroutines they must be transformed to a special format using the so-called Block-Cyclic distribution scheme, which should speed-up the calculation [3]. For example, if we consider the global matrix with a size of 9×9, which is mapped onto a 2×3 process grid (six tasks) and with a blocking factor of two, the decomposition which is shown in Fig. 87 has to be done. On can see that in this format different processes have different local matrix sizes, from 5×4 for process (0,0) to 4×2 for process (1,2). Moreover, the mapped indexes in the local distributed matrix are not sequential. For instance, in the process (0,0) the first row includes the following elements of the global matrix: a_{11}, a_{12}, a_{17}, a_{18}.

	0				1			2	
0	a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
	a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
	a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
	a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
	a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
1	a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
	a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
	a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
	a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

Fig. 81 Example of the Block-Cycling matrix distribution of size 9×9 into 2×2 blocks mapped onto a 2×3 process grid.

Hence, the Block-Cyclic distribution scheme described above has to be implemented in the subroutine *matrix_ww* in order to bring the local distributed matrix *a_ww* to a format compatible with the ScaLAPACK subroutines. Such index mapping was developed and implemented in two subroutines: *ScaLAPACK_mapping_i*, *ScaLAPACK_mapping_j* and then inserted in the kernel loop. Such index distribution causes bad scalability of the kernel loop when using the same structure shown in Fig. 88. Therefore, this kernel loop was rewritten one more time to ensure good scalability with the ScaLAPACK mapping scheme (Fig. 85). Using 512 cores with the new version a speed-up factor of 218 could be reached. The wall clock time was estimated for a large production run with *ntri_w*=500,000 to be about 4 hours.

```
do i =1,ntri_w
   do i1=1,ntri_w
        do k =1,3
           j = ipot_w(i,k) + 1
           call  ScaLAPACK_mapping_i(j,i_loc,inside_i)
           if (inside_i == .true.) then

               do k1=1,3
                  j1 = ipot_w(i1,k1) + 1
                  call ScaLAPACK_mapping_j(j1,j_loc,inside_j)
                  if (inside_j == .true.) then

                      dima_sca=0
                      dima_sca2=0
                      call tri_induct_3(ntri_w,ntri_w,i,i1,xw,yw,zw,dima_sca)
                      call tri_induct_3(ntri_w,ntri_w,i1,i,xw,yw,zw,dima_sca2)

                      dima_sum=dima_sca+dima_sca2

                      temp  = .5*(dxw(i,k)*dxw(i1,k1)          &
                              +dyw(i,k)*dyw(i1,k1)             &
                              +dzw(i,k)*dzw(i1,k1))            &
                              *dima_sum

                      a_ww_loc(i_loc,j_loc) = a_ww_loc(i_loc,j_loc) + temp

                  endif
               enddo
            endif
        enddo
     enddo
 enddo
```

Fig. 82 ScaLAPACK index mapping *dima* free kernel loop that builds the matrix *a_ww*.

9.2.3. **Parallelization of the *matrix_pp* subroutine**

The next subroutine chosen for parallelization was *matrix_pp*. It produces the intermediate matrix (*a_pp*) that will be used to calculate the input matrix for the eigenvalue solver. This subroutine is similar to the *matrix_ww* described above. The main difference lies in the construction of the *dima* matrix. It uses two additional matrices *dist1* and *dist2* in order to calculate its components. The size of the *dima* and the resulting matrix *a_pp* is also different from the previous subroutine, because it corresponds to the number of triangles within the plasma that should be discretized by *ntri_p*=200000 for a large production run. On one side, we got more complexity in the kernel loop, on the other side, the loop is smaller in comparison to the kernel *matrix_ww*.

The additional subroutine (*get_index_dima*) was developed in order to determine which indexes of the matrix *dima* are used for computing the matrix *a_pp* components. The kernel loop of this subroutine is shown in Fig. 89.

The scalability of this kernel loop, depicted in Fig. 89, is shown in Fig. 90. A speed-up factor of 220 can be achieved when 512 cores are involved in the computation. For a large production run the wall clock time (with 512 cores and *ntri_p*=200,000) reduces to about 2 hours.

```
do i =1,ntri_p
    do i1=1,ntri_p
        do k =1,3
        j = ipot_p(i,k) + 1
         call  ScaLAPACK_mapping_i(j,i_loc,inside_i)
         if (inside_i == .true.) then

            do k1=1,3
            j1 = ipot_p(i1,k1) + 1
            call ScaLAPACK_mapping_j(j1,j_loc,inside_j)
            if (inside_j == .true.) then

                call get_index_dima(i,i1,ku,ku2)

                dima_sca=0
                dima_sca2=0
                call tri_induct_3(ntri_p,ntri_p,ku,ku2,xp,yp,zp,dima_sca)
                call tri_induct_3(ntri_p,ntri_p,ku2,ku,xp,yp,zp,dima_sca2)
                dima_sca3=.5*(dima_sca+dima_sca2)

                temp  = (dxp(i,k)*dxp(i1,k1)                              &
                      +  dyp(i,k)*dyp(i1,k1)                              &
                      +  dzp(i,k)*dzp(i1,k1)) *dima_sca3

                a_pp_loc(i_loc,j_loc) = a_pp_loc(i_loc,j_loc) + temp

            endif
            enddo
        endif
        enddo
    enddo
enddo
```

Fig. 83 ScaLAPACK index mapping *dima* free kernel loop that builds the matrix *a_pp* in subroutine *matrix_pp*.

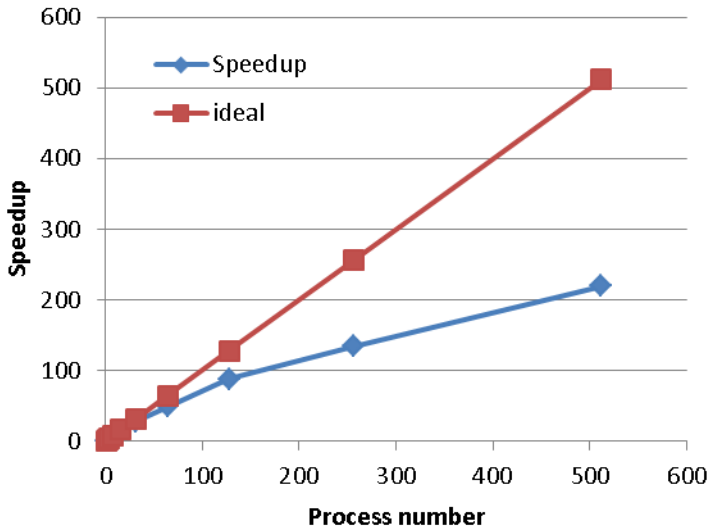


Fig. 84 Speed-up of the kernel loop in the *matrix_pp* subroutine versus number of MPI tasks.

9.2.4. **Parallelization of the *matrix_wp* subroutine**

The *matrix_wp* subroutine is similar to the previously parallelized subroutines *matrix_ww* and *matrix_pp* described above. The main difference lies in the presence of two large matrices, *dima* and *dimb,* that have to be eliminated from the code in order to save a significant amount of memory. Therefore, the components of these two matrices have to be calculated directly in place rather than stored in memory. Additionally, the *a_wp* matrix size (*npot_w*, *npot_p*) and the indexes of the kernel loop (*ntri_w*, *ntri_p*) are also different from the previous subroutines.

The subroutine was successfully parallelized providing identical results as the original version within an absolute difference of 10^{-10}. The scalability of the subroutine is shown in Fig. 91. A speed-up factor of 148 can be achieved when 256 cores are involved in the computation. The subroutine was tested for a large production run with *ntri_p*=2*10^5 and *ntri_w*=5*10^5. The execution time with 128 tasks was about 3.5 hours.

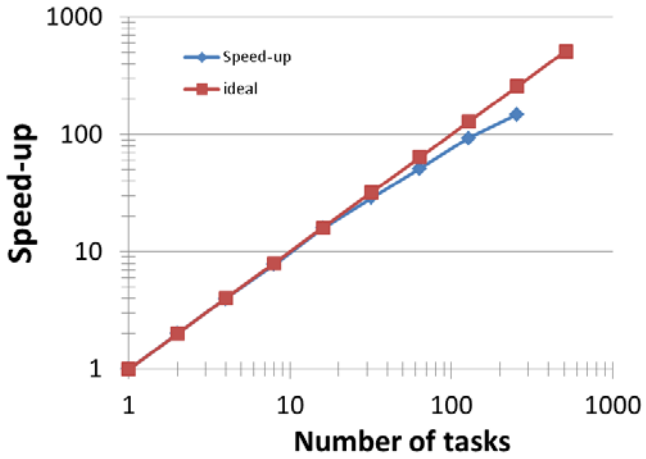


Fig. 85 Speed-up of the *matrix_wp* subroutine versus number of MPI tasks.

9.2.5. **Parallelization of the *matrix_rw* subroutine**

The parallelization of the *matrix_rw* subroutine was relatively straightforward in comparison to the previous *matrix_wp* subroutine since it does not involve the large matrices *dima* and *dimb.* The only problem was to bring the local matrix *a_rw* to the ScaLAPACK matrix structure described earlier. The subroutine was successfully parallelized providing accurate results within difference of ~10^{-10}. The subroutine was tested for a large production run with *ntri_p*=2*10^5 and *ntri_w*=5*10^5. The execution time using 256 tasks was in the range of a few minutes.

9.2.6. **Parallelization of the *matrix_pe* subroutine**

The *matrix_pe* subroutine has a different kernel loop structure compared to all previously parallelized subroutines. It is independent of the *dima* and *dimb* matrices and the indexes of the kernel loop run from 1 to the number of harmonics (*n_harm*) and to the number of boundary elements (*N_bnd*). The subroutine was parallelized with high accuracy (absolute difference of ~10^{-10}) and the output matrix (*a_pwe*) was re-ordered to be compatible with the ScaLAPACK matrix structure. Because of much smaller values of *n_harm* and *N_bnd* than *ntri_p* and *ntri_w* the execution time for this subroutine is small (few minutes) for a production run.

9.2.7. **Parallelization of the *matrix_ep* and *matrix_ew* subroutines**

The subroutines *matrix_ep* and *matrix_ew* have a similar structure with differences only in the size of the main arrays (*a_ep* and *a_ew*). *a_ep* has the size of the potential points for the plasma (*npot_p)* and *a_ew* of the potential points for the wall (*npot_w)*. All other loops and components are identical.

In the main body of these subroutines three additional supplying subroutines are called. They are *bfield_par, bfield_c* and *real_space2bezier.* Moreover, inside the subroutine *real_space2bezier* two LAPACK functions are executed (*dpotrf* and *dpotrs*). The former computes the lower-upper (LU) factorization of a tridiagonal matrix, while the latter solves a system of linear equations with a Cholesky factored symmetric positive definite matrix. Fortunately, these functions use as input parameters the matrices *aa* and *t* with dimensions (*n_dof_bnd*, *n_dof_bnd*). As the variable *n_dof_bnd* is about 400 for a production run, the double precision arrays (*aa*

and *t*) will not represent more than 1.5 MB. Therefore, we left these LAPACK functions untouched i.e. in the sequential version.

After the parallelization of the subroutines *matrix_ep* and *matrix_ew,* including the inner supplying subroutines, the total computational time was measured for a production run with *ntri_w*=500000. Using 256 tasks the wall clock time for the *matrix_ep* was 51 s, while 15 s was necessary for computing the *matrix_ew* subroutine.

9.2.8. **Parallel matrix transpose**

One part of the STARWALL solver recalculates the entries of the matrix *a_pwe* by using values from the transposed matrix *a_wp.* In order to improve the code performance this subroutine was replaced by the ScaLAPACK library function *PDTRAN* that can be adapted for a matrix transpose. The wallclock time does not exceed a few seconds for the production run.

9.2.9. **Parallel LU factorization with linear system solver**

Two LAPACK functions named *dpotrf* and *dpotrs* are executed after the *a_pwe* matrix transpose. The first function computes the lower-upper (LU) factorization of a tridiagonal matrix *a_pp*, while the second solves a system of linear equations with a Cholesky factored symmetric positive definite matrix. Both functions were replaced with their parallel counterpart from the ScaLAPACK library and grouped in the subroutine *cholesky_solver.* The subroutine provides the correct result within an absolute error of 10^{-10} in comparison with the sequential LAPACK version.

9.2.10. **Parallelization of building matrix *a_ee***

The sequential version of the code for building the matrix *a_ee* is shown in Fig. 92. As one can see this matrix is formed by the multiplication of the matrices *a_ep* and *a_pwe* using only a small part of the elements of the matrix *a_pwe*. This loop was replaced by the ScaLAPACK subroutine named *PDGEMM* that computes the matrix-matrix product. However, before the execution of this subroutine the distributed matrix *a_pwe* was rewritten to be used in the ScaLAPACK *PDGEMM* subroutine. Finally, the parallel version of the building matrix *a_ee* was tested and it provided correct results compared to the sequential version.

```
do i=1,nd_bez
  do k=1,nd_bez
    do j=1,npot_p
      a_ee(i,k) = a_ee(i,k)+a_ep(i,j)*a_pwe(j,k+nd_w)
    enddo
  enddo
enddo
```

Fig. 86 Sequential version of building the matrix *a_ee.*

9.2.11. **Parallelization of building matrices *a_ew* and *a_we***

Fig. 93 shows the sequential version of the building of the matrices *a_ew* and *a_we.* The structure of these loops is similar to the one described in the previous section with different sizes and indices. However, both loops can be replaced by the ScaLAPACK subroutine for the matrix-matrix product (*PDGEMM*) as it was done for building the matrix *a_ee*. Two new subroutines named *a_ew_computing* and *a_we_computing* were created, which include the parallel building of the distributed matrices *a_ew* and *a_we*, respectively.

```
do i=1,nd_bez
  do k=1,nd_w
    do j=1,npot_p
      a_ew(i,k) = a_ew(i,k) - a_ep(i,j)*a_pwe(j,k)
    enddo
  enddo
enddo

do i=1,nd_w
  do k= 1,nd_bez
    do j=1,npot_p
      a_we(i,k) = a_we(i,k) +a_wp(i,j)*a_pwe(j,k+nd_w)
    enddo
  enddo
enddo
```

Fig. 87 Sequential version of building the matrices *a_ew* and *a_we.*

9.2.12. Parallelization of the LAPACK *dgemm* subroutine

The last call of the STARWALL *solver* subroutine is the LAPACK *dgemm* subroutine for the multiplication of the matrices *a_wp* and *a_pwe.* This subroutine was replaced by its parallel counterpart from the ScaLAPACK library namely *PDGEMM*. The same subroutine was used to build the matrices *a_we*, *a_ew* and *a_ee.* Therefore, its implementation was relatively easy and required only a few additional ScaLAPACK descriptors. The whole computation was encapsulated in the subroutine named *matrix_multiplication.*

9.2.13. Parallelization of *resistive_wall_response* subroutine

The *resistive_wall_response* subroutine follows after the *solver* subroutine described above. There are three main parts of this subroutine: (i) eigenvalue solver, (ii) preparation of output matrices and (iii) printing of final results. The eigenvalue solver has been parallelized in the very beginning of this project described in section 9.2.1.

After solving for the eigenvalues the output matrices *a_ye*, *a_ey* and *d_ee* are computed. The sequential version of the calculation of these matrices is presented in Fig. 94. As we can see the matrices *a_ey* and *d_ee* are computed by the matrix-matrix multiplication scheme, while in order to calculate the matrix *a_ye* the transpose of the matrix *s_ww* is required. All loops were successfully parallelized and copied in three subroutines named *a_ey_computing*, *a_ye_computing* and *d_ee_computing.*

The last part of the *resistive_wall_response* subroutine is printing the computed matrices to the different output files. All matrices that were calculated in the parallel version of the STARWALL code are distributed over the number of MPI tasks using the ScaLAPACK block-cycling distribution scheme. Thus, the output subroutine should match with the reading subroutine in the JOREK code that is not implemented yet. Therefore, we did not modify the printing part of the code and postpone it until the reading part in JOREK will be implemented in order to know the necessary output format.

```
a_ye =0.
do i=1,n_w
   do k=1,nd_bez
    do j=1,n_w
      a_ye(i,k) = a_ye(i,k) + S_ww(j,i)*a_we(j,k)
    enddo
  enddo
enddo


do i=1,n_w
  do k=1,nd_bez
    a_ye(i,k) = a_ye(i,k)/gamma(i)
  enddo
enddo


a_ey = 0.
do i=1,nd_bez
  do k=1,n_w
    do j=1,n_w
      a_ey(i,k) = a_ey(i,k) + a_ew(i,j)*S_ww(j,k)
    enddo
  enddo
enddo


d_ee = 0.
do i=1,nd_bez
  do k=1,nd_bez
    do j=1,n_w
      d_ee(i,k)= d_ee(i,k) + a_ey(i,j)*a_ye(j,k)
    enddo
  enddo
enddo
```

Fig. 88 Sequential version of computing the final matrices *a_ye*, *a_ey* and *d_ee*.

9.2.14. Parallelization of matrix *s_ww* inversion

The last computing subroutine of the STARWALL code, before printing out the final results, performs the inversion of the eigenvectors matrix (*s_ww*). Two LAPACK subroutines are used for this purpose. They were replaced by their parallel counterpart from the ScaLAPACK library. First, the subroutine named *PDGETRF* calculates the LU factorization of a general matrix using partial pivoting. Second, *PDGETRI* computes the inverse of a matrix using LU factorization from the previous step. Both subroutines were grouped in the subroutine named *computing_s_ww_inverse.* The computational time of this subroutine was measured to be of ~2805 s for a production run (*ntri_p*=2*10^5 and *ntri_w*=5*10^5).

9.2.15. Parallelization of input subroutines

Three input subroutines were also parallelized: *control_boundary* that reads the JOREK control boundary data; *tri_contr_surf* that is used to generate the control surface triangles and *surface_wall* that performs the discretization of the wall. These subroutines were parallelized in such a way that only one master task reads the data from the input files and broadcasts it to the tasks involved in the computation. An additional subroutine named *control_array_distribution* was inserted after the reading part. This subroutine controls and checks the distribution of the matrices among the MPI tasks.

9.3. *Parallel performance test*

After the whole code was parallelized and tested for the correctness of the output results we did a comparison of the code performance with respect to the original version. The maximum possible problem size for the original code version which fits into memory is the following: *ntri_p*=48000, *ntri_w*=65000, *nharm*=11 (57 GB memory consumption). The wallclock time for such a simulation using 16 OpenMP processes is ~4 hours. We performed a simulation with identical parameters but with the new (MPI parallel) code version. In spite of the larger complexity of the solver due to the new version of the matrix building subroutines, which avoids the storing of the largest matrices in the code named *dima* and *dimb,* the total computational time (excluding the output) on one computing node and 16 MPI tasks is about the same as it is in the OpenMP version of ~4.2 hours consuming 41 GB of the memory. However, the computational time is reduced to about 40 minutes when using eight compute nodes and 128 MPI tasks. Nevertheless, for the small problem sizes which fit in the memory of one node, the OpenMP version is faster than the parallel one with 16 MPI tasks.

Next step was to test the code performance for a typical production run with the following parameters: *ntri_p*=202.240, *ntri_w*=500.000, *nharm*=11. Fig. 95 shows the execution time of some subroutines from the parallel version of the STARWALL code. For this test 2048 MPI tasks were used distributed among 128 compute nodes on HELIOS. The execution time from all subroutines shown in Fig. 95 represents 99% of the total computational time that is about 11 hours. One can see that four subroutines (*matrix_pp*, *matrix_wp*, *matrix_ww* and the eigenvalue solver – *simil_trafo*), described in details above, consume most of the computational time.

Name	Computation time (s)
matrix_pp	2774
matrix_wp	7591
matrix_ww	13206
matrix_rw	0,27
matrix_pe	0,007
matrix_ep	189
matrix_ew	146
cholesky_solver	345
a_pwe_s_computing	1591
a_ee_computing	3,9
a_ew_computing	38
a_we_computing	34
matrix_multiplication (dgemm)	382
simil_trafo (Eigenvalue solver)	11820
a_ye_computing	49
a_ey_computing	63
d_ee_computing	75

Fig. 89 The wall clock time of some subroutines from the parallel version of the STARWALL code for a production run with the following parameters: *ntri_p*=202.240, *ntri_w*=500.000, *nharm*=11. The subroutines are listed in their execution order.

We gradually increased the problem size and determined the maximum possible run within 128 nodes with the following parameters: *ntri_p*=202.240, *ntri_w*=551.250, *nharm*=11 and a wallclock time about 13 hours.

9.3.1. **Parametric scan of the ScaLAPACK blocking factor**

It was mentioned above that the ScaLAPACK library requires a special matrix distribution format (Block-Cyclic). The blocking size of such a format is defined by the

user, and it has a strong impact on the code performance. Fig. 96 shows the wall clock time for a production run (*ntri_p*=202.240, *ntri_w*=500.000) of five subroutines, that consume more than 95% of the total computational time, versus different sizes of the ScaLAPACK blocking factor (from *NB*=2 to *NB*=256). Among these subroutines are two from the ScaLAPACK library (matrix multiplication – *DGEMM* and the eigenvalue solver – *PDSYGVX*) and three for the building matrices (*matrix_pp*, *matrix_wp*, *matrix_ww*). One can see that the execution time of the ScaLAPACK subroutines decreases using a higher blocking factor. For small blocking factors (*NB*=2 or *NB*=4) the execution time of the eigenvalue solver is too large (>15 hours) for the program to finish within 24 hours. Therefore, these points are not depicted in Fig. 96. A significant reduction of the computational time is visible up to *NB*=64. After that the execution time decreases but only by a few percent when it reaches *NB*=128. With *NB*=256 the execution time of these subroutines begins to increase. The computational time of the matrix building subroutines fluctuates for all blocking factors. The total computational time (orange line) shows that the best performance for such a problem size is ~11 hours with *NB*=64. This is in agreement with the ScaLAPACK documentation where developers propose for the best performance to use the following blocking factors *NB*=32, 64 or 128 [3]. However, for a different problem size the best performance could be with a different blocking factor. Therefore, the STARWALL input file was extended including now the blocking factor as an input parameter.

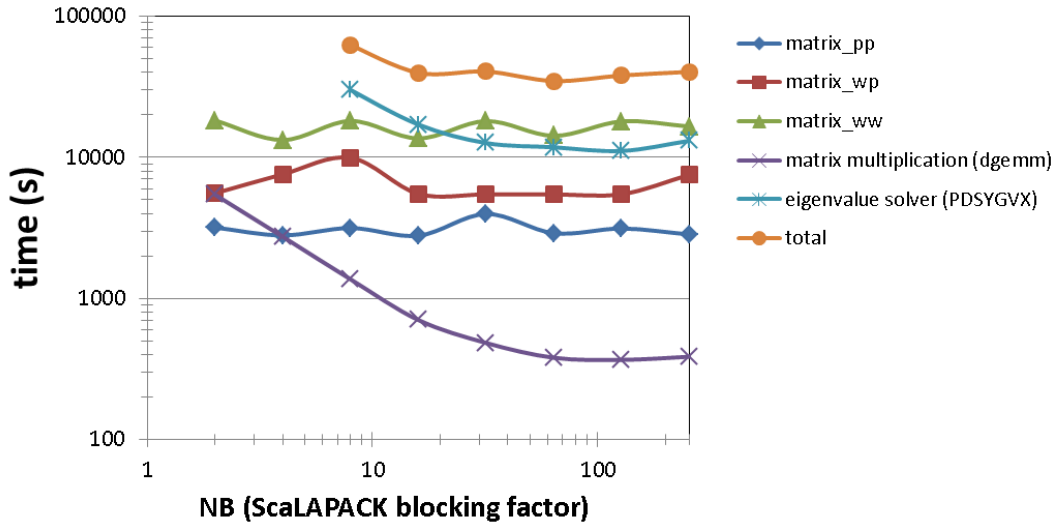


Fig. 90 The wall clock time versus the ScaLAPACK blocking factor for a production run with the following parameters: *ntri_p*=202.240, *ntri_w*=500.000, *nharm*=11.

9.3.2. **Scalability test**

We tested also how the total computational time scales according to the number of MPI tasks involved in the calculation. We decreased the problem size to be able to run it on a smaller number of compute nodes. Fig. 96 shows the wall clock time for a production run (*ntri_p*=202.240, *ntri_w*=460.800) of the four subroutines, that consume most of the total computational time, versus the number of MPI tasks. For such a problem size the whole code can be executed on 64 nodes (1024 MPI tasks). With a smaller number of nodes only a part of the code is performing due to the memory limit. One can see that the wall clock time decreases for all subroutines up to 128 nodes (2048 MPI tasks). Up to 4096 tasks the execution time of the ScaLAPACK eigenvalue solver continues to shrink. However, the computational time of the three matrix building subroutines starts to grow above 2048 tasks. We detected that the optimal code performance could be reached by using 128 compute nodes with a ScaLAPACK blocking factor of 64 for the production run described above. For a larger or smaller problem size the scaling could be different.

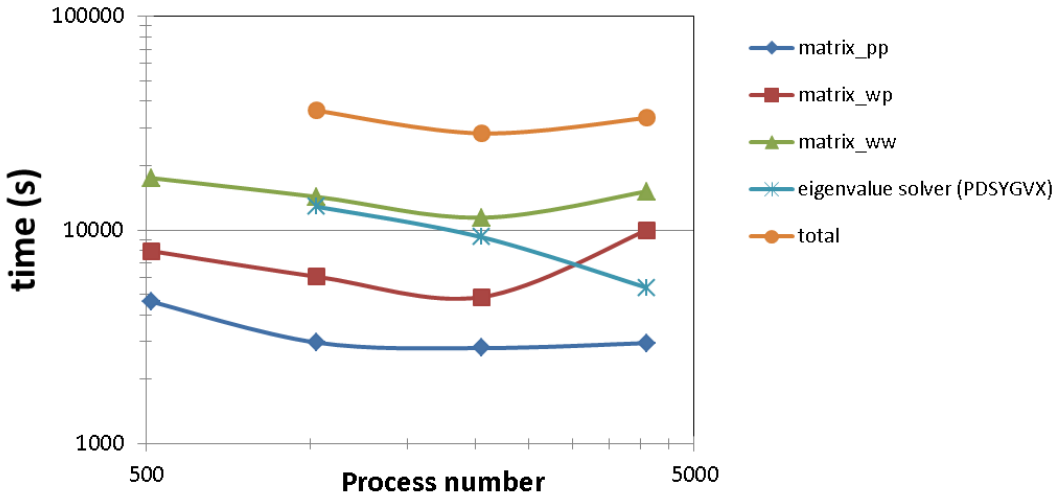


Fig. 91 Scaling of the most time consuming subroutines in the STARWALL code.

The scalability of the whole program execution including the output was tested also for a moderate problem size with *ntri_p*=48000, *ntri_w*=39200, *nharm*=11 (Fig. 98). A speed-up factor of nine was achieved with 256 MPI tasks in comparison to 16 MPI tasks. On one node, the original version is faster than the parallel version due to the much more complex algorithm used for the matrix building subroutines that avoids to store the largest matrix in the code named *dima* and *dimb*. However, with two nodes the total wall clock time becomes smaller than in the original version and the speed-up factor of six can be achieved with 256 MPI tasks in comparison to the original version.

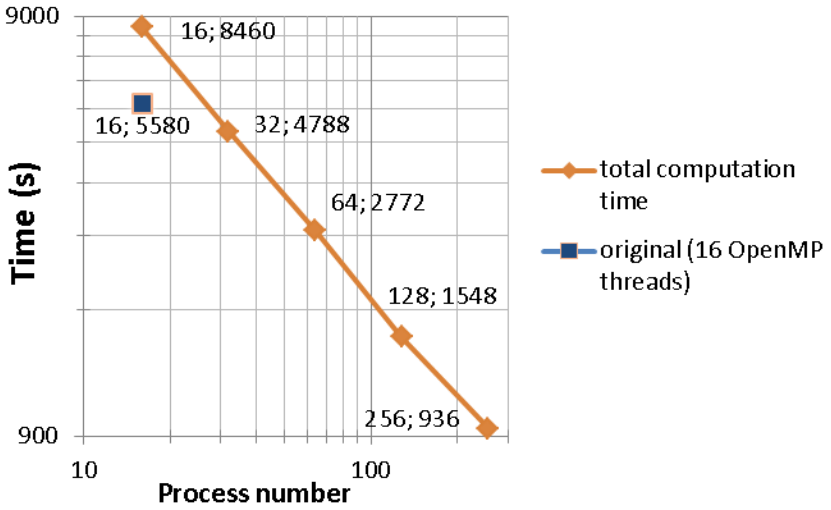


Fig. 92 Scaling of the total wallclock time in the STARWALL code for a small problem size with *ntri_p*=48000, *ntri_w*=39200, *nharm*=11. The numbers next to the points are the task number and computation time.

9.3.3. **Temporary output**

In the future a consistent format of the output of the STARWALL code and the input of the JOREK code has to be chosen. After that both subroutines must be parallelized. At the moment we use the same output format in the parallel version as in the sequential one. This gives a limitation for the problem size resulting from the output matrix size of no more than 3.5 GB.

9.4. ***Parallelization of the code version for magnetic coils***

The standard code version does not include a calculation over the external magnetic coils. However, in the future, this feature of the code must be usable for production

runs with a high number of finite element triangles. Therefore, it was decided to parallelize the subroutines that deal with the magnetic coils. Among them are one reading (*read_coil_data*) and five matrix building subroutines (*matrix_cc*, *matrix_cp, matrix_wc*, *matrix_rc, matrix_ec*). All these subroutines have been successfully parallelized providing identical results in comparison with the original code version. Due to the project time limit the performance of these subroutines was not measured. However, it is expected that including these additional subroutines will not increase the wallclock time for a production run by more than 10–20 %. This is due to the relative small matrix sizes being involved in the external coils calculation in comparison to the matrices that were parallelized before.

9.5. *Outlook*

The format of the distributed matrices has to be defined in the STARWALL and JOREK codes. Only then the according parallel I/O routine can be adjusted.

For some small problem sizes (not production runs) the parallel version of the STARWALL code works slower than the original OpenMP version due to the much more complex matrix building algorithms that avoid to store the two largest matrices in the code namely *dima* and *dimb*. In the future these matrix building subroutines can be rewritten in a way that if all matrices can be stored within the memory of one node the code will use the algorithms of the old version.

The advantage of the MPI 3.0 shared memory can be exploited for the code. It will improve the memory consumption in some subroutines. As a result simulations of a even larger problem size become feasible.

9.6. *Conclusions*

The STARWALL code has been analyzed for potential improvements and optimization by means of MPI parallel computation. It was found that for a large production run the whole code must be parallelized due to the lack of memory for saving the input/output matrices and due to the high demand of computational time.

All sequential LAPACK subroutines were analyzed and selected for replacement by their parallel analogues from the ScaLAPACK library. All these subroutines were replaced in the final code version because of the required large size of input matrices.

During the simulation tests a few bugs were found in the code that could have lead to unpredictable results. In addition, a bug with *stdin* input was detected on the *TOK-P* cluster of MPCDF.

The LAPACK subroutine for the eigenvector solver was replaced by the parallel subroutine counterpart from the ScaLAPACK library. A very good agreement was found in terms of the eigenvalues. In addition, the correctness of the results was proven by their consistency with the underlying physical model. The ScaLAPACK subroutine has shown better performance not only by using several processes in parallel but also in sequential mode due to the advantage of using IEEE arithmetics. Finally, good parallelization efficiency was obtained for this subroutine for large problem sizes.

The subroutines *matrix_ww*, *matrix_pp, matrix_wp* and *tri_induct* were re-written in order to avoid the storage of the largest matrix in the code named *dima*. This allows to save a significant fraction of the memory that will bring the opportunity to perform calculations for larger problem sizes. The subroutines were parallelized with MPI taking into account the specific output index format for matrices which is necessary for ScaLAPACK subroutines. A good scalability was achieved for all subroutines with a speed-up factor of more than 210 when 512 cores were involved in the computation.

Finally, the complete code was parallelized including all LAPACK and user written subroutines. The new parallel version of the code provides identical results in

comparison with the original one. This includes the part of the code handling the magnetic coils. The parallelized version allows production runs with much larger numbers of finite elements, i.e. to resolve the realistic wall structure. The simulation time in such a case is less then 12 hours using 128 computing nodes on HELIOS.

9.7. *References*

[1] M. Hoelzl, G.T.A. Huijsmans, P. Merkel, C. Atanasiu, K. Lackner, E. Nardon, K. Aleynikova, F. Liu, E. Strumberger, R. McAdams, I. Chapman, A. Fil, “Non-linear Simulations of MHD Instabilities in Tokamaks Including Eddy Current Effects and Perspectives for the Extension to Halo Currents”, arXiv:1408.6379 [physics.plasm-ph], 2014.
[2] https://software.intel.com/en-us/node/521158
[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users’ Guide, University of Tennessee and Oak Ride National Laboratory

10. Report on HLST project REFMUL3

10.1. *Introduction*

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

10.2. *Single-core optimization*

10.2.1. Code checking and profiling

`REFMUL3` is a full-wave code using a FDTD Yee scheme [1] with full polarization, able to cope simultaneously with o- and x-mode, which supports a general external magnetic field and a dynamic plasma. Its serial version constitutes the starting point for this project, and as such, the first step, after some basic checking and porting to the machines at hand (TOK cluster available at IPP and HELIOS available at CSC Japan), was to profile the code in terms of computational cost. To that end, measurements with `GPROF` were made on the IPP's TOK-I Linux cluster. The results are listed in Table 11. The rows depicted in magenta correspond to the three most expensive subroutines, which together are responsible for about 68% of the total cost. If we add up the remaining kernel routines of the code, depicted in green, plus the `addFLDCUDE` subroutine shown in black, we reach 98% of the total cost. This is the expected behaviour, since it is in these subroutines that most of the floating-point operations (FLOPs) are computed. This also indicates clearly where to start to increase REFMUL3's performance, namely, optimize the single-core FLOPs.

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

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

Table 11 GPROF profiling measurement of the original REFMUL3 serial code on a representative spatial grid-count of N_x=800, N_y=450 and N_z=450, for a fraction of the time steps of a production run.

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

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

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

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

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

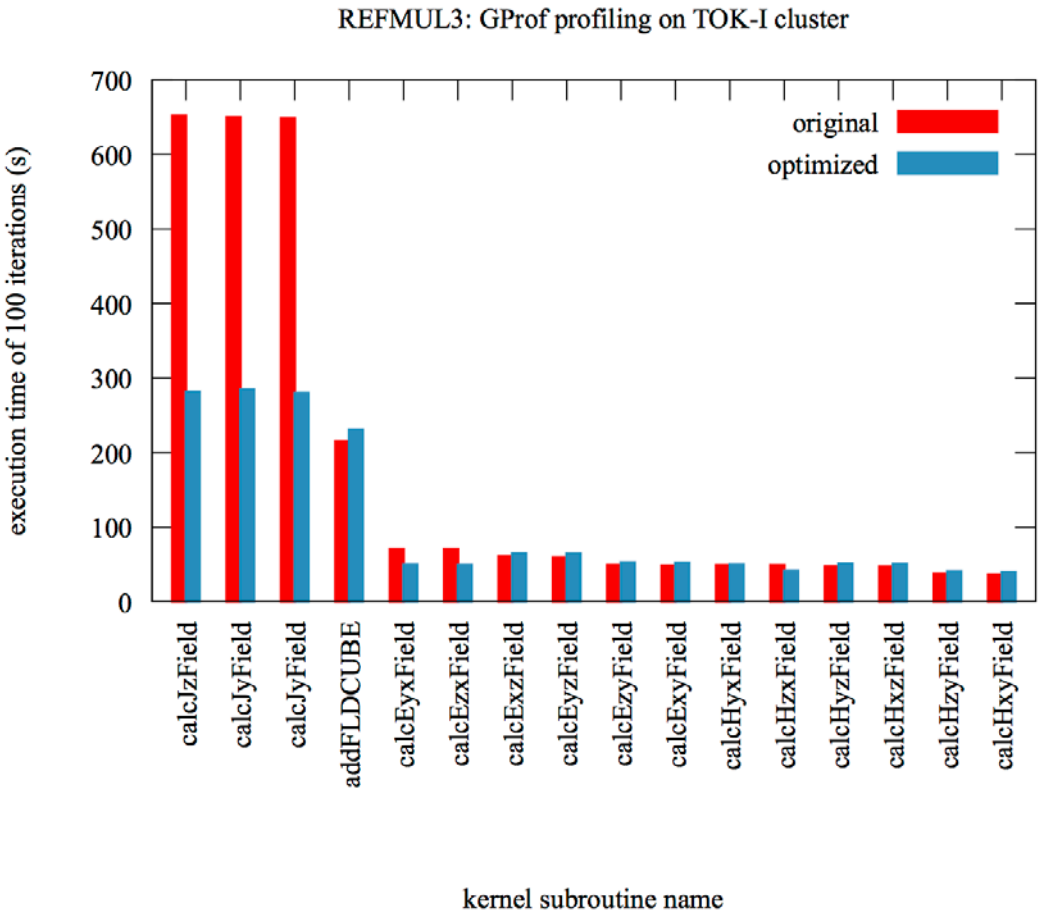


Fig. 93 Optimization of the hotspots (costly subroutines) in the serial code. The red bars represent the original cost in seconds and the blue bars correspond to the cost of the optimized versions. The same problem size of Table 11 was used here.

Minor modifications were made to comply with these rules in most of the kernel subroutines, except for the three represented in magenta in table Table 11, which according to the project coordinator (PC), had not yet been optimized in this sense. These required therefore a more extensive set of modifications. Their impact is shown in Fig. 99. As expected only these subroutines showed a significant improvement, a speedup factor on the order of 2.3x. For the others, the differences are not statistically relevant, as they are much smaller in relative terms and the measurements presented here correspond to a single run for each of the two versions of the code (original and optimized). For reference, the results shown correspond to the code compiled with the Intel C compiler. As a rule of thumb, one can conclude that it is always a good idea to apply the simple FLOP minimization rules outlined above, since they cost very little effort and can yield quite significant performance gains.

10.3. ***Parallelization strategy***

After the initial serial optimization of REFMUL3, the project proceeds towards its main goal, namely the implementation of a parallel version, followed by the assessment of the resulting performance gains. Since the idea is to scale the code beyond a single compute-node, the use of a domain decomposition invoking the MPI standard is expected. However, in order to gain experience with the code and, at the same time, obtain improved results within a smaller time investment, it was decided to start parallelizing the code's hotspot subroutines using OpenMP threads. The implementation of the MPI parallelization should follow, in order to have both parallel paradigms available depending on the user's choice, including a simultaneous combination of both, to provide a so-called hybrid OpenMP/MPI version. This has the

additional benefit of being in line with the present tendency of increasing the number of cores-per-processor/socket that share common memory regions.

10.4. *Thread parallelism*

The parallelization using OpenMP threads over one of the domain's dimensions in the optimized serial code is the next step. Since the numerical kernel is largely symmetric in its three spatial directions, both in terms of the numerical stencil, as well as the domain's grid-count, the most efficient way to proceed is to apply the thread-based parallelization `#pragma omp parallel for` to the slowest varying index (outer loop), which in `REFMUL3` corresponds to the *z*-direction. Doing so on all the subroutines listed in Table 11 (except for `setPEC`) and taking care to properly set the data scoping (shared/private variables) inside the parallel regions, as well as to do multi-thread initialization of the corresponding memory (`initFLDCUBE`), led to a speedup factor of 11.2x, when using 20 threads on a node of the TOK-S cluster at IPP, compared to the serial version of `REFMUL3` we started the project with. The measurements were made on the largest case provided by the project coordinator (6000 iterations on a 800x450x450 domain), corresponding to a representative set of code input parameters. Disabling I/O operations, the original serial code took 43h to finish, while the threaded (optimized) version took 3h50m (20 threads) to do the same task. The I/O cost, currently implemented sequentially in the code, represents roughly 1h40m, which for the threaded code corresponds already to 30% of the total runtime cost. The expectation that the I/O operations become a strong bottleneck for the parallel code is confirmed already at the node-level, even before moving to the MPI domain decomposition and distributing the problem over several nodes. This subject shall be addressed later during the course of the project, but for the time being it is removed from the analysis reported in the next few sections, which are only concerned with the scaling of the numerical algorithm.

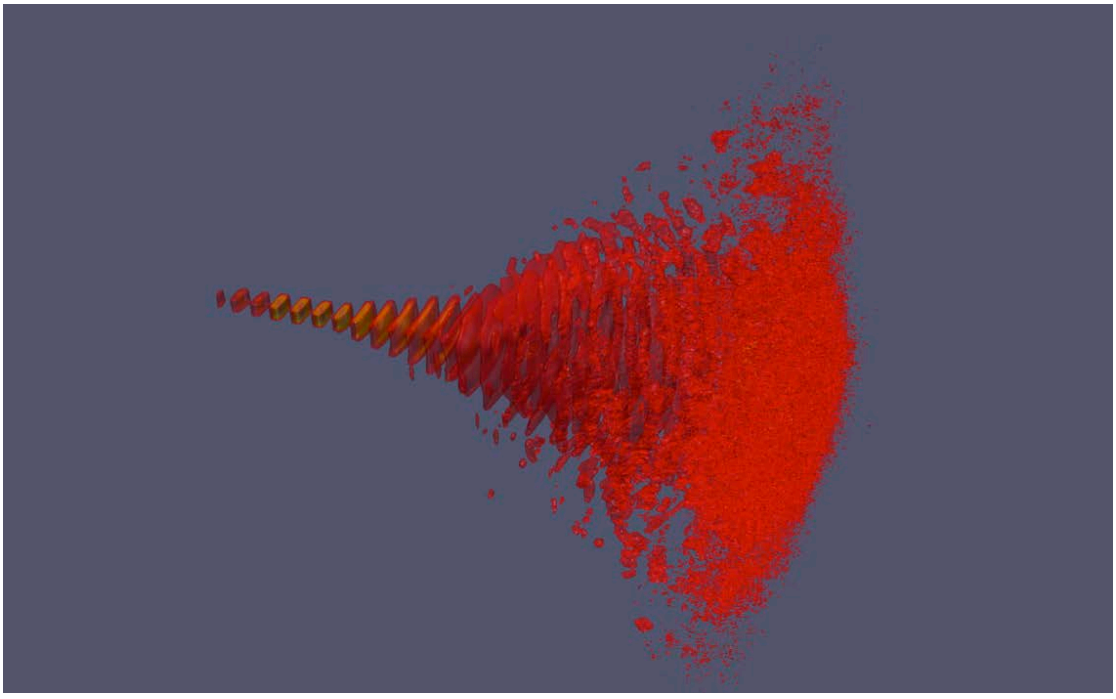


Fig. 94 Snapshot of the *z*-component of the electric field of a microwave wave propagating in the presence of a turbulent plasma. The domain was discretized on a 720x700x700 grid in the *x*, *y* and *z* directions, respectively. The simulation used 20 OpenMP threads and took about 16 h to finish 6000 iterations.

10.4.1. Domain decomposition

The thread parallelization together with the single-core optimization discussed before allowed to run REFMUL3 one order of magnitude faster than originally. In practice, that enabled already unprecedented results, like the ones illustrated in Fig. 100, from a simulation on a 720x700x700 grid in (*x,y,z*), which took 16 hours on 20 threads to complete 6000 iterations (on the TOK-S cluster). Although positive, this result is still not enough if one thinks of the time costs of a production three-dimensional FDTD simulation on bigger physics-relevant grid-counts demanding one or two order of magnitude higher number of iterations to converge. In practice, at least another order of magnitude in speedup must be achieved. For that to happen, one has to be able to distribute the problem over several computational nodes, or in other words, a domain decomposition with explicit inter-core data communication is implied.

In the course of projects HLST-REFMULXP (2013) and HLST-REFMUL2P (2014), the bi-dimensional domain of the code REFMULX was decomposed over one direction. Comparatively, REFMUL3's three-dimensional numerical kernel has a similar structure and simply involves more equations to be solved. In terms of problem sizes, this further translates into increasing the grid-count by two or three orders of magnitude, due to the higher dimensionality of the problem. Since the maximum number of resources that can be used in parallel is limited by the grid-count in the parallelized direction(s), extending the domain decomposition to at least another dimension is mandatory.

In practise it is easy to deduce that, to minimize the ratio between data communication, which is proportional to the surface of the boundaries between sub-domains, and the computation effort, measured by the volume of the sub-domains, having sub-domains with similar grid-counts in all directions is required. One can demonstrate that quantitatively using the following simple example.

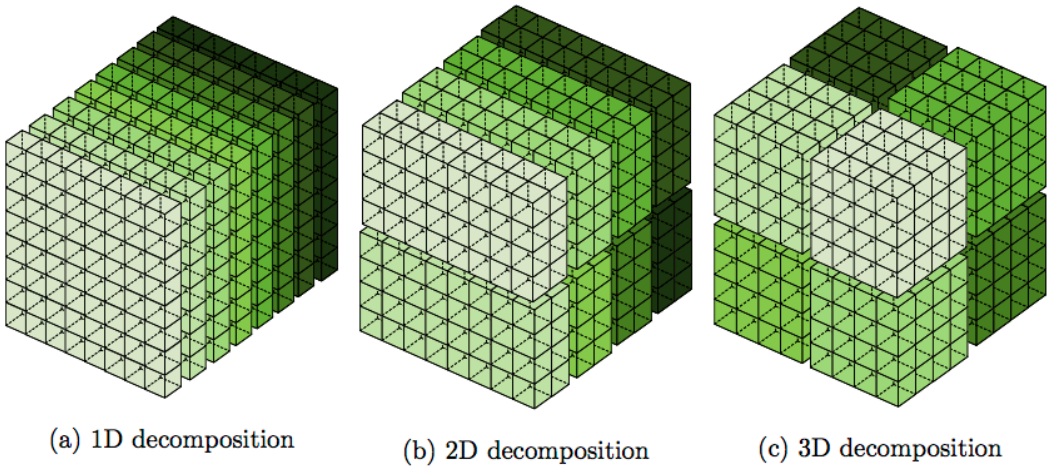


Fig. 95 Illustration of a N^3-sized three dimensional (*x*,*y*,*z*) data-set decomposed in the *z*-dimension (1D), in both (*y*,*z*)-dimensions (2D) and in all three (*x*,*y*,*z*)-dimensions (3D).

Assume you have a N^3 global domain in (*x*,*y*,*z*)-dimensions, a numerical method with two neighbours' stencil in each direction and a halo size *w* (number of ghost-cells on each boundary). Then, a one-dimensional domain decomposition over *p* processes in the *z*-direction, Fig. 101a, leads to an amount of communication per sub-domain proportional to

$$N^2 \cdot 2 \cdot w \cdot 1,$$

namely, one *xy*-plane of depth *w* per boundary in the *z*-dimension. On the other hand, if the decomposition is made instead over two dimensions, namely over *y* and *z*, as illustrated in Fig. 101b, then the amount of communication required per sub-domain becomes $\left(N^2/p_z + N^2/p_y\right) \cdot 2 \cdot w$, where p_z and p_y are the amount of

processes in the *y* and *z* dimensions, respectively, such that $p_z \cdot p_y = p$. Further assuming for simplicity that $p_z = p_y = \sqrt{p}$ leads to

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

Finally, applying the same reasoning to a decomposition over all three dimensions as schematised in Fig. 101c yields a value for the sub-domain communication proportional to

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

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

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

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

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

Such rules hold for `REFMUL3` as well, whose typical grid-count in all three spatial directions is comparable, and the numerical stencil depends isotropically on nearest-neighbors only. Additionally, it is such a domain decomposition that obviously maximizes the number of cores over which a given domain size can be distributed, with the upper limit being the usage as many cores as grid-nodes in the domain. In other words, we can push the strong scaling limits to the maximum in this case. Therefore, it was decided to proceed with its implementation, which moreover has the additional advantage that it can easily be reduced to the bi- or one-dimensional domain decomposition cases at compile time, simply by setting explicitly to unity the number of resources allocated to remaining domain directions. The disadvantage is that, from the technical point of view, this more general framework it is considerably more complicated than the lower dimensionality decomposition counterparts. For that reason, we decided to start by assessing and implementing the concept outside `REFMUL3`.

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

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

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

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

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

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

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

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

while specifying the solution at the domain boundaries, where the labels *m*+1 and *m* represent the current and the previous iteration, respectively.

This algorithm was implemented in C with the domain decomposed in all three dimensions. Each sub-domain contains one extra ghost-cell in all three dimensions, where the values on the boundary faces of the neighbouring sub-domains are stored. This is a necessary and sufficient condition to enforce continuity of the global solution across the sub-domains for the first-order algorithm under consideration. `MPI_Dims_create` is used to automatically find a good distribution of the resources over each dimension. The Cartesian virtual topology is created by invoking `MPI_Cart_create` and the process ranks (MPI tasks) are mapped to a three-dimensional coordinate system using `MPI_Cart_coords`. This considerably facilitates finding out the task ranks of the neighbouring sub-domains in all three directions with `MPI_Cart_shift`, which are needed for the ghost-cell exchanges at every iteration.

The ghost-cells exchange is executed using calls to asynchronous point-to-point communication directives `MPI_Isend` and `MPI_Irecv`. Each direction's communication is executed separately and in sequence (first *z*-, then *y*- and finally *x*-direction) to ensure that the edges and vertices of the boundary faces are also included in the exchanged data. Even though this constraint is not needed for the Poisson problem at hand, it was still implemented to have a more flexible parallel algorithm, and also because it is very easy to relax it, namely, one just needs to invoke all communication pairs at once, and then synchronize them with a single call to `MPI_Waitall`. In principle, neither does the `REFMUL3` stencil require exchanging edges and vertices of the boundary faces, so this feature can be switched off *a posteriori*, to increase performance.

Fig. 102 shows the strong scaling obtained when running our Poisson Jacobi solver on the TOK-P cluster, at IPP. A similar domain size to the test-case given by the PC for `REFMUL3` was used, namely, a grid-count of 800x450x450 in (*x*,*y*,*z*). The code was executed for 6000 iterations and the elapsed time per iteration is plotted. It is interesting to note that a similar intra-node strong scaling behaviour was obtained with the bi-dimensional `REFMULX` code (dashed yellow curve), here rescaled to be superimposed with the red curve, which points in the direction that the Jacobi iteration algorithm is a good choice to mimic the scaling behaviour of `REFMUL3`'s FDTD algorithm, as we suggested before. On the other hand, the inter-node scaling showed here is closer to the ideal than that of `REFMULX`, which is also not a surprise due to the considerable larger domain size of the three-dimensional solver, resulting in larger amount of work per core and consequently extends the saturation point of the strong scaling curve.

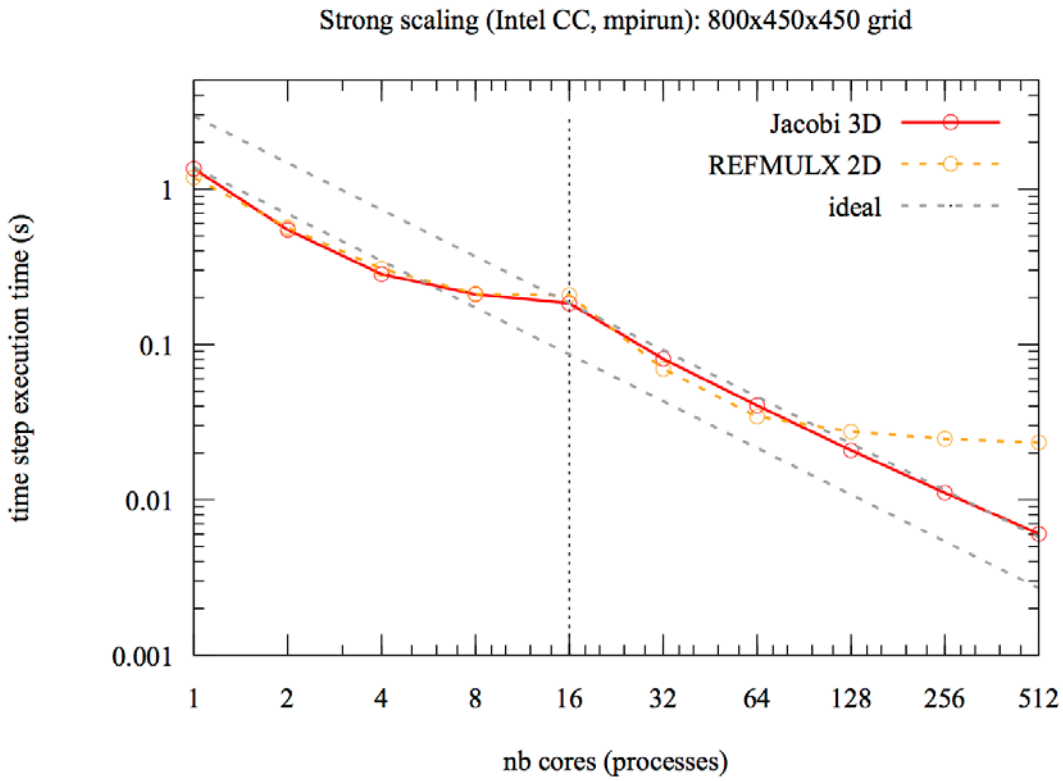


Fig. 96 Strong scaling of the three-dimensional Poisson solver using the Jacobi iteration method, employing a three-dimensional domain decomposition, measured in the TOK-P cluster (solid red). Re-scaled strong scaling of the parallel version of the code REFMULX, developed during the project HLST-REFMUL2P, measured in the HELIOS supercomputer (dashed yellow). The vertical dashed line represents a full node, with 16 cores.

Still, the question of whether the three-dimensional domain decomposition is in practice useful or not in parallel scaling terms for REFMUL3, depends on the details of its algorithm, as well as the problem sizes under consideration. Therefore, it can only be assessed with absolute certainty *a posteriori*, by means of scaling measurements of the parallelized code. However, the results obtained in Fig. 102 suggest that this is indeed the case, and therefore support the decision to implement the three-dimensional domain decomposition in REFMUL3, which is the next step to be done.

10.6. *Domain decomposition of REFMUL3*

The domain decomposition concept explained in the previous section is in the process of being implemented in the threaded version of the REFMUL3 code. The completed steps include the adaptation of the memory allocation and data access throughout the code to accommodate the partition of the global domain into several sub-domains. To do so in practice, the concept of first and last grid-node index variables was introduced. As the name suggests, these specify the global index values of the first and last grid-nodes of each sub-domain in each dimension (x,y,z). They are calculated using the specified global domain size together with the number and index values of the MPI ranks used. With this information at hand, the size of each sub-domain can be computed (which may differ between MPI tasks), and the corresponding memory local to each MPI task, including the extra ghost-cells, is allocated. Since the first and last grid-node indexes for each sub-domain are specified in terms of the global indexes, the bounds for the for-loops, which original spanned over the whole global domain, now simply need to be replaced to span instead over the former, i.e., from the first to the last sub-domain grid-node, which by definition take the correct global index values on each MPI task. Obviously, this modification proved to be more involved in the functions that contain spatially

localized quantities, like the sources or the shaped antennas and wave-guides, since they only exist in a subset of the sub-domains.

It was also decided that quantities with lower dimensionality than three, as well as three-dimensional quantities with very different sizes (typically much smaller) compared the main variables of the code (fields and currents), were not decomposed, being simply replicated in each task. Doing otherwise would impair the flexibility in the number of sub-domains that could be used for a given problem size, and moreover, the extra memory cost of having this global data replicated is hardly relevant, specially in comparison with the original serial code, where all the data was by definition global.

The new version of the code has been verified to work properly, and reproduces the results of the original version. Currently, the next step is underway, namely the implementation of the MPI initialisation and corresponding communication. This is being done in the exact same fashion it was done for the Jacobi iteration solver. We are confident that this can be finished before the end of the project, although ambitious in the sense that unforeseen bugs can arise, whose fixing, needless to say, can consume a significant time.

10.7. ***Parallel I/O***

As foreseen in the project proposal, the results obtained so far confirm that the task of writing the output results of a production run to disk will become a bottleneck. This implies that parallel I/O techniques must be invoked at some point. However, this can only be done once the full MPI domain decomposition of `REFMUL3` is in place. Therefore, due to lack of time in the current project, the task of parallelising the I/O operations, including a restart infrastructure shall be postponed for now. A strong possibility to consider is to address this point through a new dedicated HLST project, to be submitted in the future.

10.8. ***Visit from the Project Coordinator (Dr. Filipe Silva)***

Dr. Filipe Silva visited IPP for a week and a half (21–30 November 2016) in the framework of the HLST-REFMUL3 project to get acquainted with its current status. Not only could he get familiarized with the strategy chosen for the domain decomposition of his code, but he could also be directly involved in the process of implementing the necessary changes in the source code of several functions, as they were being adapted towards that goal. In that process, a couple of bugs were found and corrected in the original code, and a new smaller but still relevant reference test-case was also devised during this period. The latter was requested from the HLST side, and greatly facilitated all subsequent code changes, since the new test-case can be easily run interactively.

Finally, the visit was also very useful in that it allowed both parties to discuss and decide on important practical implementation details that were still open at the time, as well as to merge together the HLST and the project coordinator (PC) branches of the code, that had meanwhile evolved in parallel.

10.9. ***Summary and outlook***

The project is proceeding according to the milestones proposed. The single-core code profiling and optimization have been completed, resulting in an overall 1.6x speedup factor. OpenMP threads have also been implemented successfully to exploit the parallelism at the node-level, with a preliminary result of 11.2x speedup obtained on 20 threads in IPP's TOK cluster. The task of three-dimensional MPI domain decomposition is also well underway. This includes the successful implementation of a standalone test case outside `REFMUL3`, namely, a Jacobi iteration Poisson solver to test and prove the parallelisation concept. With similar numerical properties to `REFMUL3`, the very good scaling properties that the Jacobi iteration solver has shown, gave confidence on the parallelisation strategy chosen for `REFMUL3`. The

deployment of the same parallelisation techniques to REFMUL3 has started, with the infrastructure for the domain decomposition already finished. The remaining step comprises the MPI initialisation and the implementation of the corresponding communication steps, which shall be done during the remaining time of the project.

10.10. *References*

[1] K.S. Yee, *IEEE Trans. on Antennas and Propagation* **14** (1966) 302.

11. Final report on HLST project VIRIATO

11.1. *Introduction*

This project aims at improving the scalability of the code *VIRIATO*, which uses a unique framework to solve a reduced (4D, instead of the usual 5D) version of gyrokinetics applicable to strongly magnetized plasmas. VIRIATO is parallelized using domain decomposition with MPI over two directions in the configuration space domain. It is pseudo-spectral perpendicular to the magnetic field, and employs a high order upwind scheme for discretizing the equations along the magnetic field. The time integration is done via an iterative predictor-corrector scheme. A distinguishing feature of the code is its use of a spectral representation (Hermite polynomials) to handle the velocity space dependence. This converts a drift-kinetic equation for the electrons into a coupled set of fluid-like equations for each of the coefficients of the Hermite polynomials. In terms of numerical accuracy, this is rather advantageous: spectral representations are more powerful than grid-based ones, and this framework enables the deployment of the standard tools of CFD to deal with what would otherwise be a difficult kinetic equation. On the other hand, the scalability of the resulting algorithm, which uses extensively the bi-dimensional Fourier transforms, becomes a non-trivial problem, whose solution is the main goal of this proposal.

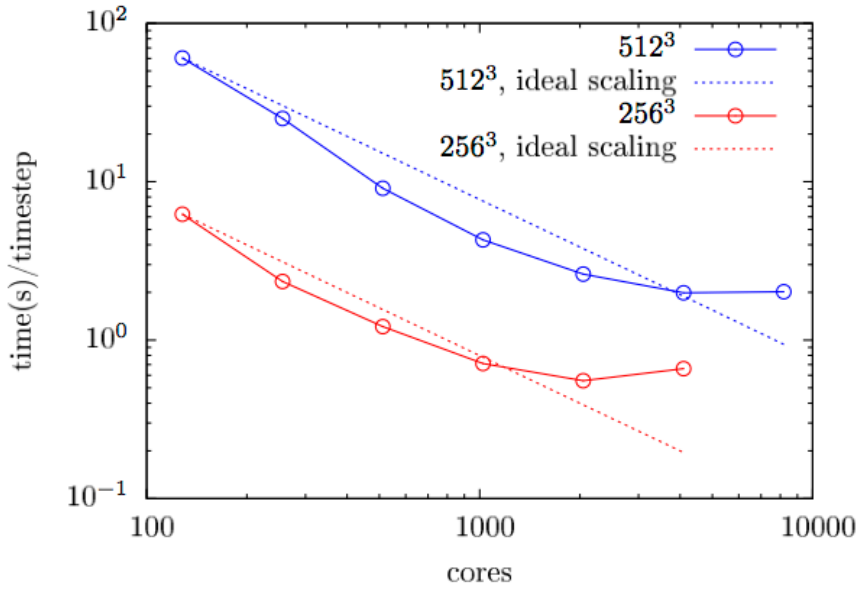


Fig. 97 Strong scaling of the original VIRIATO code with 16 velocity moments and a configuration space grid-count of 256^3 (red) and 512^3 (blue).

11.2. *Code checking*

Having received the code VIRIATO from the project coordinator (PC), the first steps involved getting acquainted with it at a basic level. The very first test was to verify that it could be compiled and ran on the HELIOS machine. Upon request, a representative test-case was also provided by the developer's team, together with a set of quantities calculated by the code, to be used as a reference whenever changes are made to the source code.

The next step involved using Forcheck to make static checks on the compliance of the source code to the FORTRAN standard. For that to work it was necessary to change VIRIATO to use the more modern MPI module header file, which allowed Forcheck to make a complete argument checking test. In doing so, an issue was found related to a mismatch between the double-precision integer variables used throughout VIRIATO (`-i8 compile-flag`) and the single-precision integer variables (MPI handles) expected within the framework of the MPI library. A complete fix for this is still to be made. Until then, even though strongly discouraged since the

introduction of the MPI-3.0 standard, the original implementation that includes the 'mpif.h' header file, has to be invoked at compile-time by means of a pre-processor flag, which was introduced for that purpose. Other smaller issues were found with Forcheck and fixed accordingly. Finally, the runtime diagnostic checks available on the Intel FORTRAN compiler were also invoked, with only some warnings being reported. Repeating the procedure with the Lahey/Fujitsu compiler is left for future work.

11.3. *Code profiling*

To assess the code performance and make a profile of its computational cost, the Allinea MAP software has been used. This provides an easy way to measure the time spent in each component of the code, including the measurement of CPU load and MPI communication costs. No changes to the source code are necessary, only a compilation with a debug flag (`-g`) is required. Subsequently, in order to have a set of simplified measurements optionally available at the end of every production run, the source code has also been manually instrumented using the performance library Perflib, developed by the Max-Planck Computing Data Facility (MPCDF). The results revealed unsurprisingly that the bi-dimensional fast Fourier transforms (bi-dimensional FFTs) dominate the costs, representing at least half of the total simulation time for relevant production cases. For instance, Table 12 shows the profiling for a typical medium-sized domain simulation running on 256 cores, where the bi-dimensional FFTs represent 70% of the total cost.

```
=== Context: global context ===
                                  Inclusive              Exclusive
Subroutine  #calls      Time(s)      %       Time(s)      %
-------------------------------------------------------------------
FFT2Ddir      84864      94.343     35.8       14.526     5.5
 F2f_x        84863       4.576      1.7        4.576     1.7
 F2f_gath     84863      68.297     25.9       68.297    25.9
 F2f_y        84863       6.944      2.6        6.944     2.6
FFT2Dinv      98040      92.349     35.0       10.165     3.9
 F2b_y        98039       4.767      1.8        4.767     1.8
 F2f_scat     98039      71.144     27.0       71.144    27.0
 F2b_x        98039       6.274      2.4        6.274     2.4
CodeInit          1      19.292      7.3        2.704     1.0
UpdateVa        101      36.946     14.0       31.843    12.1
Main              1     263.613    100.0        0.002     0.0
(....)
```

Table 12 Snippet of the VIRIATO's code profiling using MPCDF's Perflib on a medium-sized domain of 256^3 grid-nodes in the configuration-space and *ng*=16 velocity moments, on 256 cores on HELIOS.

This algorithm involves the combination of two sets of one-dimensional (1D) FFTs of tri- or four-dimensional (3D or 4D) data, interleaved with a data transposition, demanded by the non-locality of the floating point operations implied and the fact that one of the dimensions to be Fourier transformed is parallelized. This means that a major fraction of the project time has been dedicated to this part of the algorithm. On one hand, by experimenting with different methods to perform the data transposition, on the other hand, by attempting to use parallel threaded regions in this context, as explained in the next sections.

11.4. *Bi-dimensional Fourier transforms and data transposition*

VIRIATO is pseudo-spectral in the plane perpendicular to the magnetic field (*x*, *y*), and uses a spectral Hermite polynomial representation in the velocity space (*ng* coordinate). It employs the package FFTW v2.1.5 to perform the corresponding FFTs. The domain is parallelized in two directions of the configuration space, namely

the *y*- and *z*-directions. Therefore, the bi-dimensional FFTs must involve the transposition of data across a sub-set of the total MPI tasks (cores) in an all-to-all fashion. This operation poses a challenge when it comes to scalability on a large number of cores in the perpendicular plane (*npe*), the reason being that its complexity, or in other words, the number of MPI messages required, grows with $O(npe)^2$.

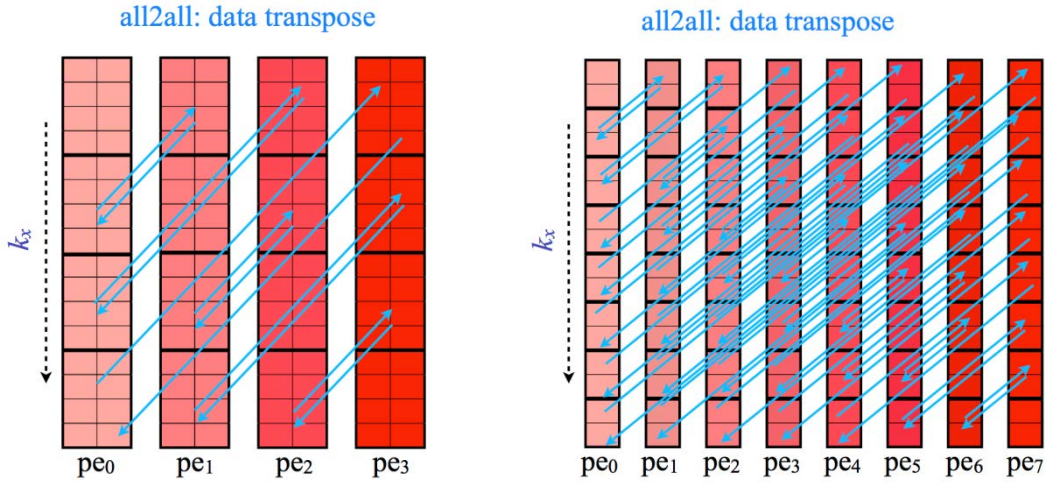


Fig. 98 Illustration of the all-to-all communication pattern for the parallel transposition of a bi-dimensional dataset parallelized over four (left) and eight (right) cores.

This is clearly shown in Fig. 104, where the all-to-all communication patterns involved in transposing the same bi-dimensional dataset distributed over four or eight cores are illustrated. The algorithm currently used for this operation was assessed in detail, and it was immediately concluded that it could be prone to suffer from network communication latency accumulation. The implementation of more efficient alternatives was made.

11.5. *Reduced test-case and the new transpose algorithms*

Since a significant part of the work was dedicated to the parallel bi-dimensional FFTs operations that are needed, both in the plane perpendicular to the magnetic field, as well as in the velocity space, a simplified test-case was devised to serve as a test-bed. It includes only the code parts responsible for the bi-dimensional FFT algorithm, therefore allowing to study their behaviour without the overhead of executing the whole VIRIATO code. It further facilitates the development and implementation of alternative algorithms, providing a direct way for the extensive testing needed to ensure correctness of the new tools before they are carried back into the main VIRIATO source code. Since both the original and the newly introduced methods are available in the test-case, comparing them in terms of parallel scalability becomes very easy.

The devised test-case calculates a sequence of bi-dimensional FFTs both in forward and backward directions on a 4D data-set, to allow comparing the final doubly transformed data-set with the original untransformed one. A sequence of convolutions is also implemented in order to mimic the method used in VIRIATO to calculate partial derivatives in the configuration-space, whereby a Fourier transformed quantity F is multiplied by $-ik_x$ and $-ik_y$ and then the inverse bi-dimensional FFT is calculated to yield $\partial f/\partial x$ and $\partial f/\partial y$, respectively. Below is the explicit formula for the former

$$\frac{\partial f}{\partial x} = \mathrm{FFT_{2D}}^{-1}\left(-iF_{k_x k_y} \cdot k_x^{\mathrm{global}}\right). \tag{1}$$

The original VIRIATO's bi-dimensional FFT algorithm has been implemented in the test-case code and then extended to minimize the accumulation of network latency for large numbers of cores, as explained below. Additionally, an alternative method was also implemented for the transpose step. It uses a sequence of MPI_Sendrecv point-to-point directives dictated by a bitwise exchange pattern computed using an exclusive-OR (XOR) logical operation between the MPI task ranks and an appropriate data-indexing within each sub-domain [1].

The original bi-dimensional FFT algorithm applies the transpose to each *xy*-plane sequentially for every grid-node in the remaining one (*z*-direction) or two directions (*z*- and *ng*-directions). While this maximizes flexibility in the sense that the same subroutine can be called for all Fourier transforms of the code inside do-loops covering the indexes in the remaining (one or two) dimensions, it is also prone to network latency accumulation. To understand this, we need to recall Fig. 104 and the corresponding discussion. We have seen that, as the number of cores used in the perpendicular plane (*npe*) is doubled, the size of each MPI message required to be communicated across cores is halved, but their total number increases by a factor of four. On the other hand, as the size of the messages decreases, the latency cost, a hardware limitation that can be conceptually defined as the (finite) time of exchanging a zero-sized message, starts to dominate the communication process. This means in practice that, for large enough *npe*, the transpose algorithm, independently of its quality, will start to be negatively affected by the network latency. This is essentially the reason for the degradation of the VIRIATO strong scaling, shown in Fig. 103. Since latency is a fixed cost that depends on the total number of messages exchanged, i.e., on the number of cores used (*npe*), and not on their size, repeating the perpendicular *xy*-plane transpose for each of the grid-nodes in the remaining dimensions just increases this cost linearly. To avoid this accumulation, the data in the remaining dimensions can be aggregated in order to be carried during one single transposition. The transpose algorithm was extended to do so.

The impact of these changes on the parallel performance was measured using the test-case described previously and the results are shown in Fig. 105 for a grid-count of 256^3 with 16 velocity moments. A substantial difference can be observed, especially for higher numbers of cores (*npe*). The red line represents the strong scaling of the original bi-dimensional FFT algorithm in VIRIATO. It fairly closely mimics the scaling of the full VIRIATO code (Fig. 103), as expected, since this algorithm represents the bulk of the computation costs therein. Up to 1024 cores, the number of MPI tasks involved in the *xy*-plane transpose (*npe*) is kept fixed, while the number of MPI tasks in the *z*-dimension (*npez*) is increased from 8 to 128. This explains why the scaling is so good in this region, namely, the amount of FFT computations-per-core decreases without increasing the complexity of the transpose's all-to-all communication (recall Fig. 104). That it is even super-linear seems to be related to the overhead associated with assignment of the buffers used for MPI communication, which for lower numbers of cores are relatively large. Because this overhead is accumulated for each grid-node in the *z*- and *ng*-directions, doubling *npez* translates into halving it, which is an effect that comes on top of having more resources available to calculate the same amount of FFT operations.

The light-blue line corresponds to the extended algorithm with data aggregation over the *z*-dimension. Even though the MPI-buffers in this case are even bigger than before, and so is the associated MPI-buffer overhead, there is no accumulation with the *z*-grid-count simply because the whole *z*-direction is carried in "one go" when calling the *xy*-transpose. Therefore, it costs much less communication time, and shows the expected perfect linear scaling up to 1024 cores. As *npez* approaches its maximum (128) for the used *z*-grid-count (256), both methods' performance tend to each other as expected, because the number of *z*-grid-nodes per sub-domain decreases to two, so that aggregation in this dimensions loses most of its role. The yellow line uses the same algorithm, except that it further aggregates the data in the velocity space dimension when the transpose is carried out. If on the one hand the

MPI-buffer overhead is higher because the messages are (*ng=) 16* times bigger compared to the solid light-blue case, on the other hand its accumulation is decreased by the same factor, so that both should approximately cancel. In reality, this is not exactly the case, at least for the domain size used here, as can be seen from the difference between both scaling curves (yellow and light-blue) for the lower *npe* numbers.

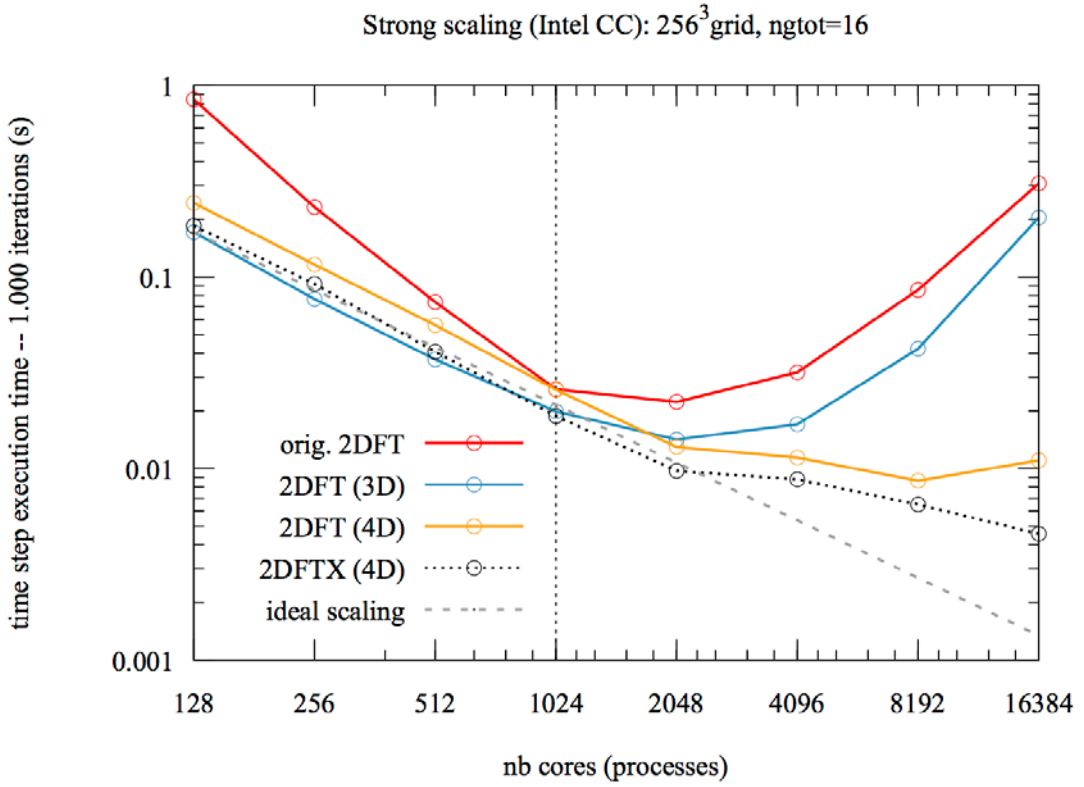


Fig. 99 Strong scaling of the parallel bi-dimensional FFT algorithm on the same medium-sized domain of Fig. 103. VIRIATO's original algorithm is shown in red. The extended algorithm with the data aggregated over the *z*-dimension and over the *z*- and *ng*-dimensions are shown in light-blue and yellow, respectively. The dotted-black curve shows the behaviour of the whole algorithm when the alternative XOR-transpose is employed together with data aggregation over the *z*- and *ng*-dimensions.

Beyond 1024 cores, indicated by the dashed vertical line, the increase in the number of MPI tasks corresponds to an increase in *npe*. This immediately degrades the scaling for all curves, as expected, because the complexity of the all-to-all communication pattern involved grows with $O(npe)^2$. That the red and light-blue curves show a qualitatively similar behaviour (shape) in this region is clear, since we concluded before that for *npez*=128 they converge to each other up to a factor of two. But why do their strong scalings completely break down? Or in other words, why using more resources on the same problem size results in higher computational costs? The reason is exactly the one we discussed in the beginning of this section, namely the accumulation of network latency. This limitation plays a role whenever the MPI message size is small enough that the cost of its transfer, dictated by the maximum network bandwidth, is comparable to or smaller than the latency. As *npe* increases, the size of the MPI messages decreases, so latency becomes the dominant communication cost, and worse, since the number of messages increases, so does the total latency cost. The difference between the red or light-blue curves and the remaining two, whose strong scaling still show the expected monotonic decreasing shape, comes from the data aggregation over the moments dimension performed in the latter. Hence, the latency price is paid only once in those cases, whereas it is multiplied by the number of moments kept in the velocity space for the

case of the light-blue curve (*ng*=16), and additionally by the *z*-grid-count divided by *npez* for the red curve (*ng*Nz*/*npez*=32). The difference between the yellow curve and the dotted-black curve arises solely from the different methods used to establish and execute the data exchange pattern of the transpose. The latter, based on a XOR exchange pattern seems to be slightly more efficient than the transpose used originally in VIRIATO. In terms of speedup figures between the latter (fastest) and the original method (red) the figures are approximately 2x, 4x, 15x and 55x for 2048, 4096, 8192 and 16384 cores, respectively.

11.6. ***Deployment to VIRIATO***

Since the test-case introduced in the previous section was made under a framework fully consistent with the full VIRIATO, deploying the new tools back to the main VIRIATO code could be made relatively straightforwardly. Basically, the same modules used originally in VIRIATO were extended with the new FFT bi-dimensional algorithms.

We have seen in Sec. 11.5 that the best result was obtained using the data-aggregation technique together with the XOR-transpose (dotted-black curve in Fig. 105). However, there is an important practical difference between this transpose algorithm and the original one used in VIRIATO. They differ in the way the Fourier transformed data in the *x*-direction is stored over the sub-domains. While the original transpose (independently of the data aggregation) stores the k_x modes in a compact manner over the sub-domains, the XOR-transpose distributes them evenly across sub-domains. Obviously this difference only comes about when the grid-count in the k_x direction is not a multiple of the number of cores *npe*, but this is more often the case than not, due to the Hermite redundancy of the k_x-spectrum. Since the untransformed dataset is purely real, the result after the first 1D Fourier transform (in the *x*-direction) keeps $N_x/2+1$ non-redundant complex Fourier modes out of the N_x real numbers upon which it acts. So for example, if we start with eight real numbers, the non-redundant spectrum has five k_x modes. If we have the *y*-dimension parallelized over *npe*=4 cores, then the dimension of the transposed dataset in the k_x-direction (*nkx_par*) is given by the smallest integer which is larger than $(N_x/2+1)/npe$=5/4, i.e *nkx_par*=2. This is the example illustrated in Fig. 106, for the case with N_y=4.

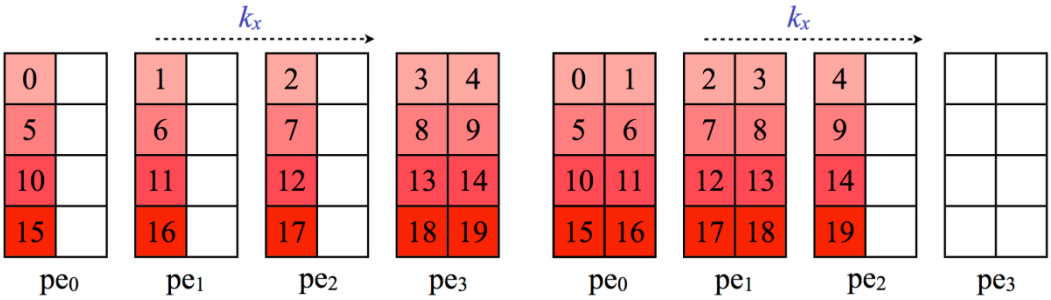


Fig. 100 Illustration of the different storage of the radial Fourier modes over the sub-domains yielded by the original transpose (a) and new XOR-transpose methods (b).

The figure on the left side (a) represents how the k_x-modes are stored over the available sub-domains when the original transpose method is used, and the figure on the right (b) represents its counterpart for the new XOR-transpose algorithm. The different storage schemes are equivalent from the performance point of view, but they do have practical consequences for VIRIATO, the most immediate one being that these methods can not be used interchangeably. Either one or the other has to be used everywhere in the code. The other issue to take into account is that, since the mapping from the k_x-modes to the position in the sub-domains changes, all numerical operations which involve it directly, must be adapted for the case of the new XOR-transpose method. One such example is the calculation of the partial

derivatives in the radial (*x*) direction, which is made using the convolution of the dataset whose derivative is being computed with the global k_x-mode number, as in Eq. (1). We can see for instance that in case (a) k_x=1 is stored in the sub-domain of the core rank p_e=0, whereas in case (b) it is stored in the sub-domain of the core rank p_e=1.

The new global k_x-mapping has been devised and implemented in the test-case. However, the impossibility of mixing both FFT bi-dimensional methods renders the deployment of the XOR-transpose to the main VIRIATO source code much more involved and time consuming. Only after the full implementation is finished can correctness-tests and debugging be started. Due to time constraints, it has been decided to leave this task for future work. Since the tools (test-case) are available, the full deployment of this method can be made by VIRIATO's developers later on, if they so decide.

Contrary to the XOR-transpose method, the extended version of the original bi-dimensional FFT method used in VIRIATO (red curve in Fig. 105) can be deployed progressively into the source code. This was done by changing, one by one, all the calls to the Fourier operations, replacing the original subroutines with overloaded versions which are selected based on the number of dimensions of the input arrays. These include `FFT2d_direct`, `FFT2d_inv`, `Bracket`, `Convol2` and `Funcgm`. Further, in the module transforms, two new subroutines were introduced, namely, `init_redistribute3` and `init_redistribute4`. They are responsible for the extension of the transpose using the aggregation of data in the 3rd and 3rd+4th dimensions technique to minimize the latency accumulation, as explained in Sec. 11.5. Because it is desirable to keep all the methods (original and the new alternatives) available, at least initially, it was decided together with the project coordinator to use pre-processing conditionals to decide on which method to use at compile time.

11.7. ***Strong scaling results of VIRIATO***

Before analyzing the full VIRIATO code scaling results, a few words are in order to guide their interpretation. First, we need to observe that the best results obtained with the simplified test-case shown in the strong scaling of Fig. 105 refer to a single four-dimensional data-set being transformed several times in sequence. They represent an idealized situation. In the full VIRIATO code, the scaling is necessarily worse for two main reasons. First, there are several three-dimensional quantities in the code, for which only the data aggregation in the z-direction is possible. In practice this implies that the strong scaling for the bi-dimensional FFTs of those quantities will degrade much earlier because smaller messages are involved. Second, even for the four-dimensional quantities involved in the code, there are regions where using the data aggregation in both the *z*- and *ng*-directions is disallowed by data dependencies (e.g. the calls to the `Funcgm` subroutine inside the `P_Loop`). Therefore, the light-blue (original bi-dimensional FFT method) and red (extended bi-dimensional FFT method with transpose invoking data aggregation in *z*- and *ng*-dimensions) curves in Fig. 105 provide an indication for the lower and upper bounds in performance expected for the full VIRIATO strong scaling. This is indeed what is observed in Fig. 107 and Fig. 108.

The red curve represents the scaling of the original VIRIATO code. It corresponds exactly to the red curve depicted in Fig. 103. The blue curve uses the data aggregation in the *z*-direction for the transposes. This can be done everywhere in VIRIATO since all FFT calculations involve quantities which are at least three-dimensional. The yellow curve corresponds to the same method except for the coefficients of the Hermite polynomials in velocity space. Being four-dimensional quantities, they allow in principle to further invoke the data aggregation in the *ng*-direction (on top of aggregation the *z*-direction) during the transpose step. Unfortunately, inside the `P_Loop` region, data dependencies prevent this. There, for each time step, the calculation of each moment coefficient *m* depends on the *m*-1 coefficient, which needs to be calculated before, so only data aggregation in the *z*-

direction is possible. As this operation represents a significant fraction of the total time-cost of VIRIATO, it necessarily contributes to the deviation of the yellow curve in Fig. 107 compared to the one in Fig. 105. Its behaviour is therefore, loosely speaking, in between the blue and the yellow curves of Fig. 105.

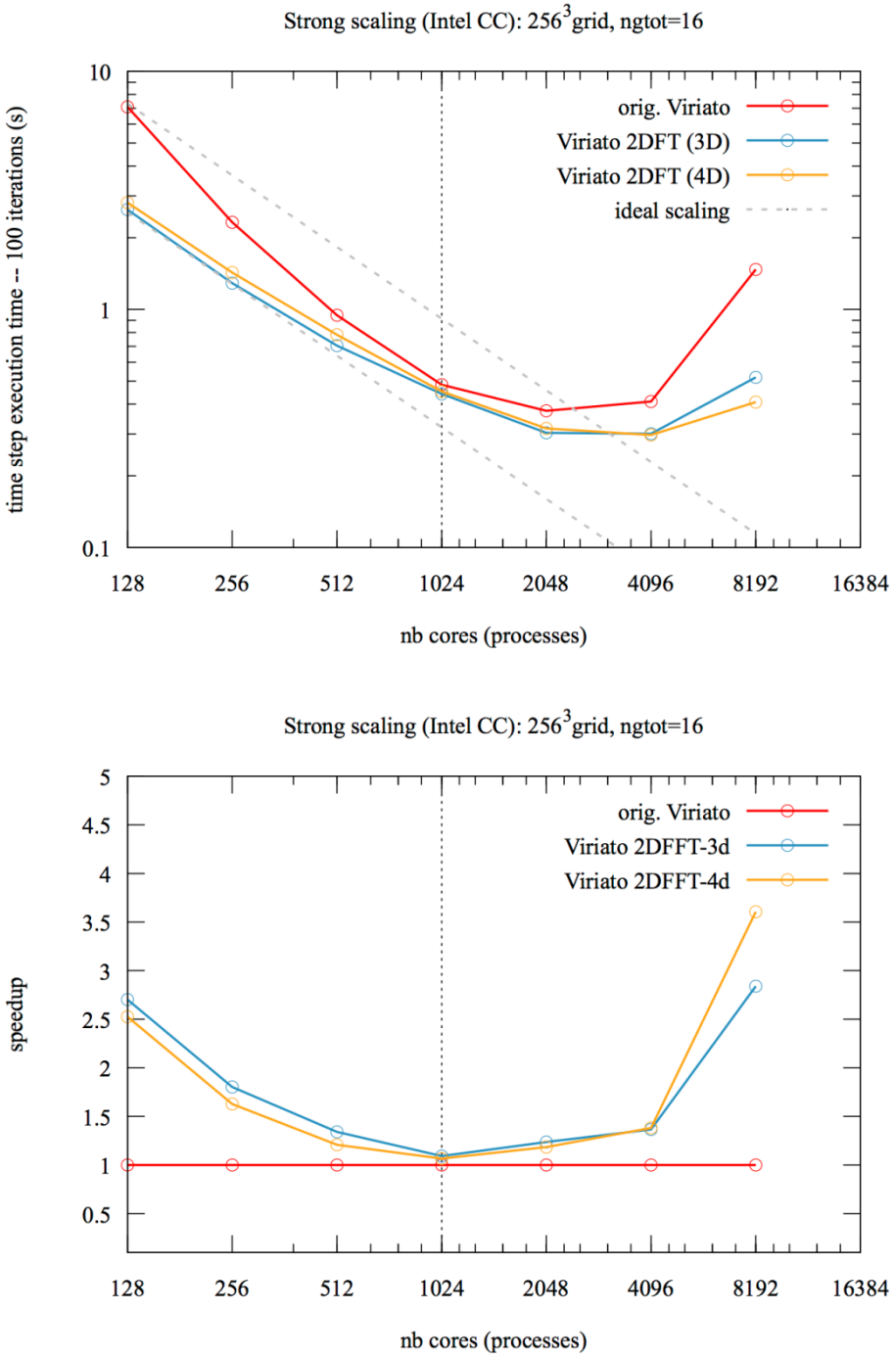


Fig. 101 Strong scaling (top) and corresponding speedup for the full VIRIATO code on the same medium-sized domain (red curve) of Fig. 103. The same colour coding of Fig. 105 is used here.

The same qualitative behaviour is observed in Fig. 108, where the same configuration-space domain was used, but with a much higher number of moment equations (256 instead of 16). Now, because more moments are available to do the data aggregation in the *ng*-direction, the difference between the blue and yellow curves is more pronounced.

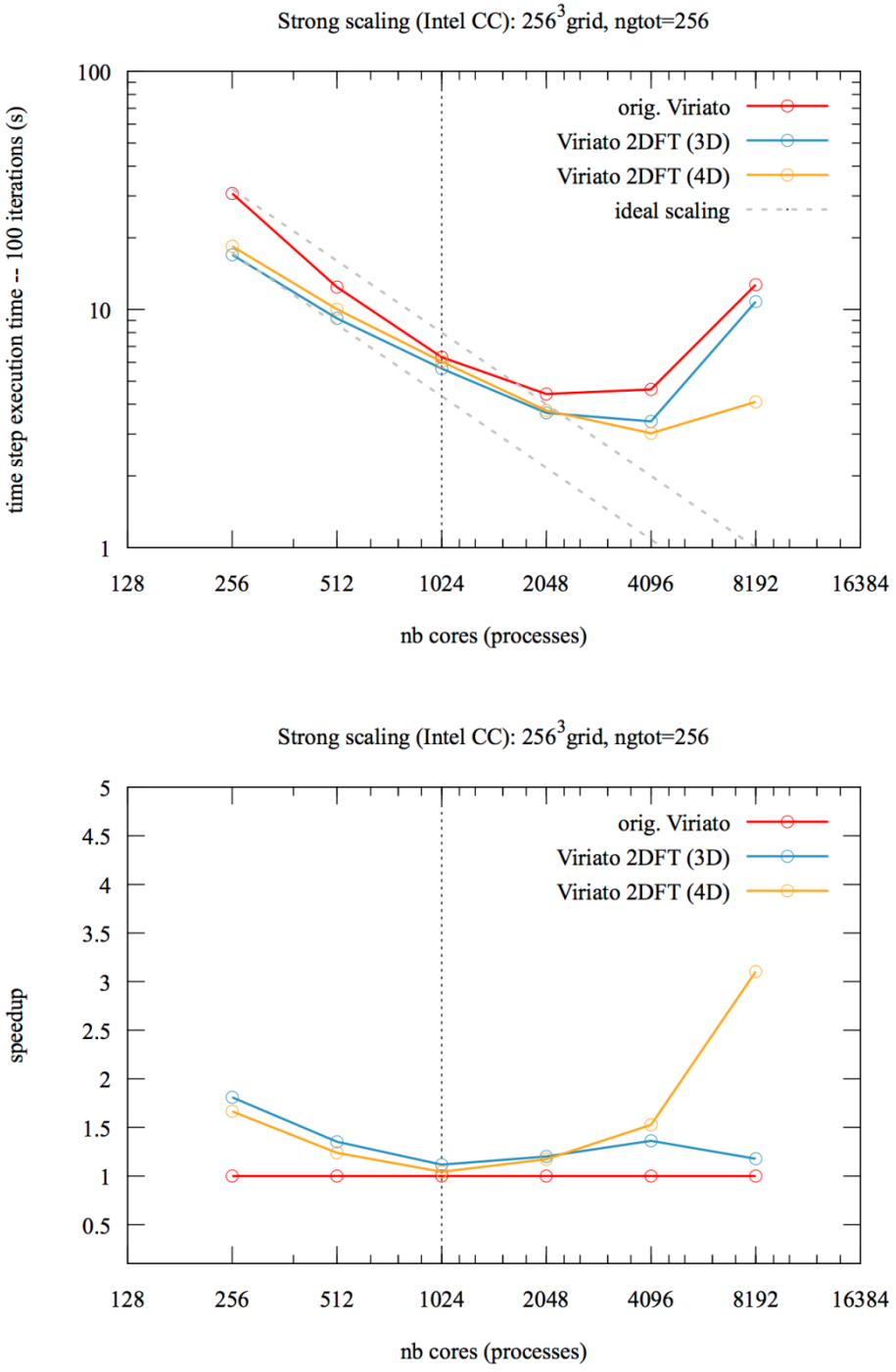


Fig. 102 Strong scaling (top) and corresponding speedup for the full VIRIATO code on the same configuration space domain of Fig. 107, but with 256 moments considered. This makes the difference between the blue and yellow curves more pronounced. The same colour coding of Fig. 105 is used here.

In practice, even though more modest than the isolated test case results, they are still very positive. On one hand, for smaller core-counts, which could be useful for smaller development runs, speedup factors of two or three can be achieved, as seen in the bottom plots of Fig. 107 and Fig. 108. On the other hand, being able to improve the strong scaling of VIRIATO beyond the point where the degradation used to start (vertical dashed line) is something with a significant practical impact on the cost of production runs, especially considering that these can cost several thousands of node-hours.

11.8. *Strategy for the hybrid parallelization*

The main idea behind the concept of a hybrid parallelization scheme stems from the fact that the topology of modern HPC facilities is not flat, in the sense that the physical cores are grouped into compute-nodes. The main consequence of this is a non-uniform memory bandwidth across the whole machine. As such, treating all the MPI tasks (cores) on equal footing is not necessarily the most efficient solution, and this is where the hybrid parallelization concept comes in, to attempt to better exploit the hardware resources. There are two main possibilities to achieve a hybrid parallelization, and these are covered in the following sub-sections.

11.8.1. OpenMP threads

VIRIATO employs a Hermite representation of the velocity space, meaning that instead of one kinetic equation the code solves a hierarchy of coupled fluid-like moment equations for the coefficients of the Hermite polynomials. The original proposal for the project was to split these amongst different cores using OpenMP parallel regions. With say, *N* OpenMP threads, each of them would solve a subset *M*/*N* of the *M* moment equations. Such a parallelization scheme needed to be implemented on top of the spatial MPI parallelization already employed in the code, leading to a hybrid MPI/OpenMP parallel code. However, there is a significant issue with such a model. Namely, since each thread would need to spawn MPI tasks, the level of MPI thread-safety required would have to be set to the highest available, namely, `MPI_THREAD_MULTIPLE`. This is the worst scenario in terms of complexity for an OpenMP/MPI hybrid implementation, and, at the time of writing of this report, is not yet supported by all MPI libraries, in particular by the BullxMPI available on HELIOS.

An alternative that avoids the issues described before, creates instead the parallel OpenMP forks in the *y*-direction of the domain, which is already parallelized with MPI. As each MPI task in this direction can use threads (up to 16 on HELIOS) to share the work within its local sub-domain, only the master thread needs to invoke MPI communication. Hence, only a level of thread-safety of `MPI_THREAD_FUNNELED` is required from the MPI library. But perhaps more importantly is that this scheme also affects the data transposition step, which is a bottleneck communication-wise. Having threads sharing the memory at the node level over this dimension can greatly reduce the complexity of the all-to-all communication involved in the bi-dimensional FFTs. Since only the MPI tasks, not the OpenMP threads, need to be involved in the MPI transpose communication, the number of messages can be significantly reduced, from $O(npe)^2$ to $O(npe/nthreads)^2$ instead [1]. For example, with four network channels available for inter-node communication and 16 cores per node on HELIOS, one can devise an ideal situation of having four MPI tasks per node each with four OpenMP threads. This configuration would substantially reduce the complexity of the all-to-all communication, by reducing *npe* by a factor of four, while still having access to the full inter-node network bandwidth for the MPI communication.

To explore the possibility of implementing thread-based parallelism in the bi-dimensional FFT algorithm, another simplified test-case consistent with VIRIATO's framework was initiated to study its behaviour in a controlled manner. The very first step was to calculate the forward 1D FFT part of the algorithm using OpenMP threads. The transformation results obtained were correct but showed some degree of irreproducibility in terms of their performance scaling with the number of threads, for reasons not yet clear. Although more effort was clearly required to clarify the issue, probably using more sophisticated performance analyzing tools, like the Intel VTune, it was decided to postpone such task, as well as the subsequent thread-based hybridization steps, mainly due to time constraints. This decision was further justified by the fact that the approach under consideration here would imply code modifications in the remaining parts of VIRIATO, which are only MPI-tasks aware. Otherwise, the extra resources (threads) allocated to the FFT algorithm would idle elsewhere in the code during runtime. The problem would become even more

pressing as the time spent in the FFT computation is expected to decrease as a result of the optimizations proposed here, yielding the remaining parts of the VIRIATO code necessarily more costly in relative terms. Noteworthy is that there is no technical impediment in doing this other than the man-power needed to implement such changes in the available time frame.

11.8.2. MPI-3 shared memory windows

Alternatively to the concepts discussed in the previous sub-section, there is another hybrid scheme that can be devised for VIRIATO relying solely on the MPI library, without the need to invoke OpenMP threads. The general idea is to split the existing communicator(s) in the pure-MPI VIRIATO code into sub-communicators in order to yield different levels of communication that would better map to the hardware topology of a modern HPC machine. In particular for the FFT algorithm therein, this would mean splitting the *npe* communicator responsible for the decomposition in the *y*-direction into two kinds of sub-communicators. One to group tasks bellow a compute-node level according to a given criterion (e.g. cores-per-node, cores-per-socket, etc.) into a set of sub-communicators (one per group), while the other kind groups the set of all master-rank tasks within each of the former groups. For clarity in the remaining discussion, let's call the former *intra-group communicators* and the latter the *inter-group communicator*.

In this scenario, only a subset of the *npe* tasks would be involved in the "all-to-all" communication pattern, namely the tasks belonging to the inter-group communicator, while the remaining resources would be responsible for a local transposition of the data, using resources contained inside a compute-node. The complexity of the all-to-all communication is reduced in the same fashion as before (Sec. 11.8.1), by reducing the amount of involved MPI messages from $O(npe)^2$ to $O(npe/npintra\text{-}group)^2$. A clear benefit of this hybridization model with respect to the MPI/OpenMP paradigm is however that, since the total number of MPI tasks remains unchanged compared to the original flat-MPI code, the same amount of resources (cores) would still be available for the floating point operations (FLOPs) everywhere in the VIRIATO code at runtime, even if only parts of it, like the FFT algorithm, were hybridized.

There are two main possibilities of implementing the pure-MPI hybrid concept into the bi-dimensional FFT algorithm of VIRIATO, and the difference between them refers to the choices made inside each of the intra-group communicators. Either one invokes explicit data communication inside each intra-group communicator via calls to `MPI_Gather/Scatter`, as was already studied elsewhere [1], or one allocates so-called shared memory windows within each intra-group communicator. The latter scheme, which is available since the introduction of the MPI-3 standard, completely suppresses the need for explicitly intra-group communication since the data stored in each shared memory window is accessible by all MPI tasks in that intra-group communicator. Since this is a very interesting new concept with very good portability characteristics, as it is support by the newest MPI standards, we decided to try it out during the remaining time of the project.

After an initial assessment phase, which included a learning period that involved participating in a training event on the subject [2], it was decided to extend the existing stand-alone bi-dimensional FFT test-case to accommodate the shared memory windows. Due to time constraints, the extension was applied only to the XOR-transpose algorithm using three-dimensional data aggregation (recall Sec. 11.5). In practice the changes implied allocating shared memory windows that serve as communication buffers for the inter-group MPI communication. All the cores belonging to the same intra-group communicator share these buffers and can therefore perform the local transpose step inside the group without explicit communication. The performance results obtained are very positive, as can be seen from Fig. 109, where the cases of sharing memory between all the cores inside a node (dotted green) and inside a socket (solid green) are shown. Namely, the latter yields the best scaling results obtained during the project.

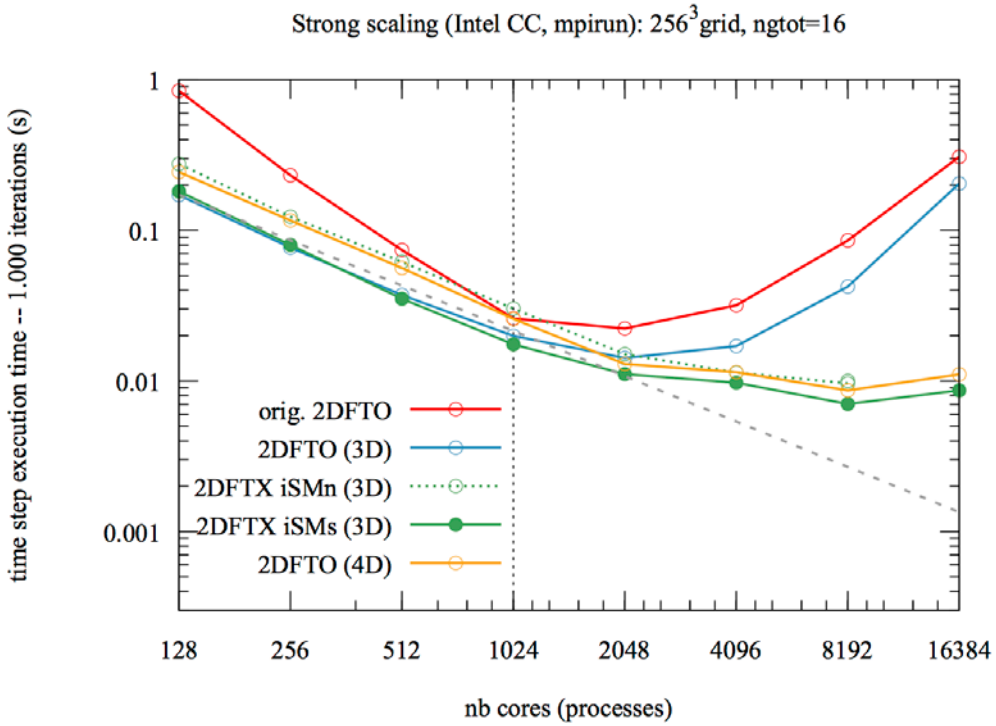


Fig. 103 Same as Fig. 105, but including the curve for the XOR-transpose with the data aggregated over the *z*-dimension using shared memory windows over the cores belonging to the same compute-node (dashed green) and compute-socket (solid green).

It is clear from the different behaviour of both green curves that there is room for optimization depending on the choice made for the intra-group communicators. Grouping the cores in order to share memory resources within a socket, provided that the appropriate task binding is enforced, yields considerable better results (within a factor of two) than doing the same within the full node. This is consistent with the fact that intra-socket memory bandwidth is higher than the memory bandwidth yield between both sockets within a node. Clearly, NUMA effects are relevant performance-wise for this algorithm, as was expected.

But the most important point to make here is that, not only is this the best performing FFT algorithm, it is also applicable everywhere in the full VIRIATO code. Since it only invokes the *z*-direction data aggregation in the transpose, the algorithm does not suffer from the moment data-dependency that prevented the four-dimensional data aggregation transpose to be applicable inside the `P_Loop` subroutine, which substantially degraded the scaling behaviour of the full VIRIATO, as explained in Sec. 11.7. Hence, it is expected that the scaling behaviour of the solid green curve in Fig. 109 directly translates to the VIRIATO code once the new shared memory windows' based algorithm is ported there. This is a very promising prospect, but as the project time was finally exhausted, the corresponding work must be left for the future.

11.9. *Visit to IPFN-IST Lisbon*

A trip to the IPFN–IST association in Lisbon, where the PC and its team are based, was made between 05–25.07.2015. This was very fruitful, as it came during a period where important technical decisions needed to be made regarding the following steps in terms of the project roadmap. For instance, it was then decided to put the main effort on improving the transpose algorithms at hand, which as reported here, led to significant gains in the overall parallel bi-dimensional FFT algorithm of VIRIATO. It further allowed to explain to the VIRIATO's developers team the extent of the changes needed and to discuss with them in real time as the first results were being achieved.

Later on, it also greatly facilitated the task of handing back the optimized VIRIATO code to them at the end of the project, since they were, to some extent, already familiar with the changes introduced.

11.10. *Summary and outlook*

The main goal of this proposal was to improve the parallel scalability of VIRIATO, and the work has been carried out successfully. Having initially profiled the code under a production sized domain, it was confirmed that the bi-dimensional FFT algorithm constituted a hot-spot of VIRIATO in terms of computational cost. The performance of VIRIATO's original version of this algorithm was assessed in detail and more efficient alternatives have been implemented in a stand-alone test-case, mostly by modifying the parallel transpose algorithm implied in these operations. The main idea was to aggregate data before carrying out the all-to-all communication patterns to avoid a penalisation due to network latency accumulation. Significant gains in performance have been achieved, in some situations reaching figures of the order of more than 50x on 16 thousand cores on HELIOS.

The deployment of the new developed tools back into VIRIATO's source code was made. Even though the absolute gains obtained within the full code were, as expected, much more modest, they allowed to have VIRIATO run the same problem with double the number of cores while still maintaining a good strong scaling behaviour. The main reason for the scaling degradation in VIRIATO compared to the stand-alone FFT algorithm is that the most performing version, which uses data aggregation in the *z*- and *ng*-directions, could not be invoked everywhere in the code due to algorithmic data-dependencies in the numerical scheme used in VIRIATO.

The remaining time of the project was dedicated to the assessment of a possible hybridization of VIRIATO. Having a hybrid parallel code nowadays is desirable as it allows exploiting the current trend in computer architectures, where many cores have access to shared memory banks, reducing, or even completely dropping, the need for explicit communications at that level. Indeed, the upcoming EUROfusion HPC machine based in CINECA (Italy) will use Intel Broadwell processor chips with 18 cores each, which more than doubles the corresponding figure available currently on HELIOS. Two paradigms were considered, namely, one that uses a mixed MPI/OpenMP hybridization and another one that does a pure MPI hybridization. Some tests were made using the former, although most of the effort was in the latter. The main reason for this decision being that the latter allows for a partial hybridization of a pure flat MPI code without having idling resources at runtime. To better use the limited remaining time available in the project, we restricted ourselves to the hybridization of the bi-dimensional FFT algorithm. The existing stand-alone test-case was then extended to accommodate the so-called shared memory windows, available since the introduction of the MPI-3 standard (and already implemented in the Intel MPI). For practical reasons, this was applied to the XOR-transpose method using three-dimensional data aggregation. The complexity of the underlying all-to-all communication required was greatly reduced [1] by making use of the shared memory resources available inside a compute-node, and as a consequence the best scaling results were obtained with this technique. Further, unlike the algorithm which before yielded the best performance, namely the four-dimensional data aggregation transpose, the new hybrid algorithm is not affected by the moment data-dependency that degraded the overall VIRIATO scaling behaviour compared to the one of the stand-alone FFT algorithm. Even though there is still some work required to deploy the new method to VIRIATO, much better scaling properties are to be expected, especially for larger numbers of cores. This very promising prospect serves as a motivation for a possible future HLST project on VIRIATO.

11.11. *References*

[1] T.T. Ribeiro and M. Haefele, *Parallel Computing: Accelerating Computational Science and Engineering* (*ParCo Conf. 2013 proceedings)* **25** (2014) 415.

[2] *Introduction to hybrid programming in HPC @ LRZ*, 14 January 2016, Garching (Germany)

12. Report on process pinning options on MARCONI-Fusion

12.1. *Introduction*

The MARCONI supercomputer uses the popular cache-coherent Non-Uniform-Memory-Access (ccNUMA) architecture in its nodes. This means in particular that not all the cores are equivalent in terms of how fast they can access data from the node's main memory. Specifically, a node in MARCONI has two sockets with 18 cores each, yielding 36 cores sharing a total of 128 GB of main memory. However, this memory is made out of two banks (of 64 GB), each directly connected to one of the two sockets, as schematized in Fig. 115. Using NUMA terminology, we say that the cores on each socket together with the attached memory bank form a *NUMA node*. Thus, a compute node on MARCONI is comprised of two *NUMA nodes*. For the sake of simplicity, we shall denominate *NUMA node* by *socket* in the remainder text. Therefore, even though any core on the compute node can access the whole main memory of the node, the access is faster within the same socket than across the sockets. As a result, it is generally very important performance-wise to specify appropriate binding affinities of the computational processes to the hardware cores within a node at run-time, and to ensure that the processes remain pinned that way throughout the simulation. This holds for both OpenMP threads, as well as MPI tasks, including obviously their combination in hybrid codes.

In the remaining text we present examples of MARCONI PBS batch submission scripts that allow to obtain several relevant process binding affinities, when invoked together with the Intel MPI library. Indeed, OpenMP thread and MPI task pinning are discussed in Secs. 12.2 and 12.3, respectively, while Sec. 12.4 describes the hybrid case of using both. The first two provide the simplest batch submission script examples for codes which are either purely OpenMP or purely MPI. The latter provides a more general case, which although slightly more involved, works also for pure OpenMP and MPI cases, so it can be regarded as an interesting alternative if one does not want to have different scripts for different parallelization paradigms. Finally, Sec. 12.5 provides some intra-node strong scaling results of a production code (REFMULXp [dasilva2015]), which serve to emphasize the practical relevance of the process binding concepts explained before.

The material presented here is not intended to be, in any fashion, exhaustive. It merely attempts to provide a sub-set of rules that are of practical use for the typical cases encountered within codes in the framework of the EUROfusion consortium. For more detailed information we suggest consulting the Intel manual pages on *Process Pinning* [intel0] and respective subsections, *Environment Variables for Process Pinning* [intel1] and *Interoperability with OpenMP API* [intel2].

12.2. *OpenMP thread affinity*

Let’s begin with the pure OpenMP threaded case, for which the script provided in Table 13 is our baseline example. The lines that control the number of threads to be spawned and their binding affinity to the hardware cores are highlighted in blue. The first assumption that we make here, and that will be clear later on, is that one needs to ensure that pinning of the threads throughout the run. This is clearly reasonable, even before looking at any results, since we know already that the main memory bandwidth is not uniform within the node. This can be achieved via the `KMP_AFFINITY` environment variable. Therefore, it is recommended to always set it explicitly in the batch script.

The example script sets `KMP_AFFINITY=compact`, whereby the processes are mapped to neighboring cores as closely as possible, filling first the first socket, and only if there are more threads than cores per socket, do the threads begin to fill the second socket. This also means that there is no need to turn on the option

OMP_PROC_BIND=true, which is superseded by the former. We further added the option verbose so that the explicit binding of the threads to the cores is written to the standard error file, which makes it easy to confirm the actual affinity used in the corresponding simulation afterwards.

```
#!/bin/bash
#$ -cwd
#PBS -N omp_12
#PBS -l select=1:ncpus=36:mpiprocs=1:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export OMP_NUM_THREADS=12
export KMP_AFFINITY=compact,verbose
BIN=a.out
CMDLOPT=''
./$BIN $CMDLOPT
```

Table 13 Basic script used to run a.out with 12 threads distributed according to the default affinity on a MARCONI node, which at the time of writing of this document was *compact*.

The compact affinity of the 12 threads (processes) yielded by this script is depicted on the left side of Fig. 110. However, depending on the algorithm at hand, this might not necessarily be the best choice of binding pattern, as we shall see later, in Sec. 12.5. Changing it is quite simple, namely, if we replace the blue line setting the KMP_AFFINITY in Table 13 with the following one

```
export KMP_AFFINITY=scatter,verbose
```

a *scatter* affinity is obtained, whereby the processes are mapped as remotely as possible so as not to share common resources, in this case, the L3 caches and the memory banks in each socket (see Fig. 115). The particular example yielded by the modified script is also illustrated in Fig. 110, but now on its right side.

If no value is explicitly set for the number of OpenMP threads, then the PBS system will fix this automatically to OMP_NUM_THREADS=ncpus, where ncpus is the number of cores requested in the select PBS directive, in our example 36. Note that, alternatively to using the variable OMP_NUM_THREADS, the number of threads can also be set by adding ompthreads to the select PBS directive above. For instance, for 12 threads, this would be

```
#PBS -l select=1:ncpus=36:mpiprocs=1:ompthreads=12:mem=118GB
```

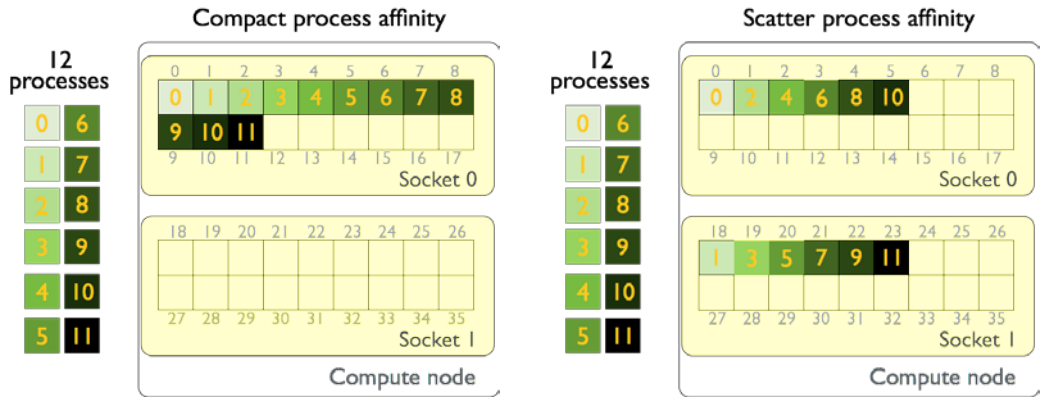


Fig. 104 Schematics of 12 processes (threads or tasks) distributed with *compact* (left) and *scatter* (right) affinity on a MARCONI node. Note how consecutive processes alternate between sockets on the latter.

It is also noteworthy to comment on the value used for the PBS batch resource list option `ncpus=36`. This tells the batch scheduler to reserve the full node, even if less resources are requested, like in the example provided (12 threads, instead of 36). On one hand, for the *EUROfusion* dedicated part of MARCONI (MARCONI-Fusion), this makes no difference in terms of budget accounting because we can only ask for full nodes, unlike the remaining part of the machine where nodes can be shared by different jobs. On the other hand, `ncpus=36` is really needed for pinning affinities other than *compact*, because resources (cores) not included in the PBS select directive will not be available at run-time. For example, if we were to set it instead to the number of threads in the script above, namely `ncpus=12`, then the setting `KMP_AFFINITY=scatter` would, in practice, yield a *compact* affinity. An even more extreme situation would be to set `ncpus=1`, in which case all the 12 threads would be pinned to core 0 of the node, regardless of the pinning affinity specified.

12.3. *MPI task affinity*

For a domain decomposed code, much of the same considerations about process pinning to the hardware resources (cores) apply. In this case, the processes are MPI tasks, so a different set of environment variables is responsible for specifying the affinities. For the simplest case of a pure MPI code, our baseline script is listed in Table 14. As before, the lines highlighted in blue are the ones used to tune binding affinities, which now refer to the MPI tasks. The first sets `I_MPI_DEBUG=5` and its purpose is similar to the `verbose` keyword used before for the `KMP_AFFINITY` variable, namely, to write the explicit binding of the processes (MPI task ranks) to the cores to the standard error. The second line turns on the MPI task pinning `I_MPI_PIN=1`, so that the chosen affinity is enforced throughout the simulation, preventing the task to move between cores. Note that the process pinning features presented in the following are only valid if this option is turned on, which according to Intel reference pages [intel1] is the default value.

The last line in blue is the one that actually sets the affinity, and it does so by explicitly providing a <*proclist*>, which in the case of MARCONI, where hyper-threading is disabled, corresponds to a list of cores, specified in terms of their logical numbering. Note that it is typical for Intel NUMA nodes to have different logical and physical numberings for the processors, as can be seen from the schematics of a MARCONI production node in Fig. 115. By specifying the <*proclist*> to cover all cores within the node in their increasing sequential order, this tells the Intel MPI library to place the MPI tasks such that their rank matches the logical core number. In other words, place the MPI tasks requested using a compact affinity, as depicted on the left side of Fig. 110, except that now the 12 processes are 12 MPI tasks.

```
#!/bin/bash
#$ -cwd
#PBS -N mpi_12
#PBS -l select=1:ncpus=36:mpiprocs=12:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export I_MPI_DEBUG=5
export I_MPI_PIN=1
export I_MPI_PIN_PROCESSOR_LIST=0-35
BIN=a.out
CMDLOPT=''
mpirun -prepend-rank ./$BIN $CMDLOPT
```

Table 14 Basic script used to run `a.out` with 12 MPI tasks distributed *compactly* on a MARCONI node. The `-prepend-rank` option simply adds the MPI rank number in front of each output line, which can be a useful information.

To obtain the scatter affinity shown on the right side of the same picture, all that is needed is to change the <*proclist*> accordingly. Namely, by replacing the corresponding line in the script of Table 14 with the following one.

```
export I_MPI_PIN_PROCESSOR_LIST=0,18,1,19,2,20,3,21,4,22,5,23
```

This ensures that both sockets of the node are used if more than one MPI task is requested, whereas in the compact affinity case, this only happens if more that 18 MPI tasks are used. An alternative way to enforce the property of using both sockets, while still keeping consecutive task ranks as close to each other as possible, is to use instead the following.

```
export I_MPI_PIN_PROCESSOR_LIST=0-5,18-23
```

We shall call this *bunch* affinity, following the nomenclature of the Intel MPI library, whereby the processes (*domains*) are mapped consecutively as closely as possible but at the same time distributed over both sockets in a balanced way. This kind of affinity is illustrated in Fig. 111. However, it must be pointed out that this particular configuration only works as intended for 12 MPI tasks. With less than 12 tasks, more of them will be placed on the first socket than on the second, creating an asymmetric distribution; with more than 12, the extra tasks will start to be placed in the already occupied cores following the same pattern specified in the <*proclist*=0-5,18-23> in a round-robin fashion, using the resources in a sub-optimal manner, and further potentially creating load-balance issues. Therefore, for the *bunch* affinity to work appropriately with different numbers of tasks, the <*proclist*> must be adapted accordingly.

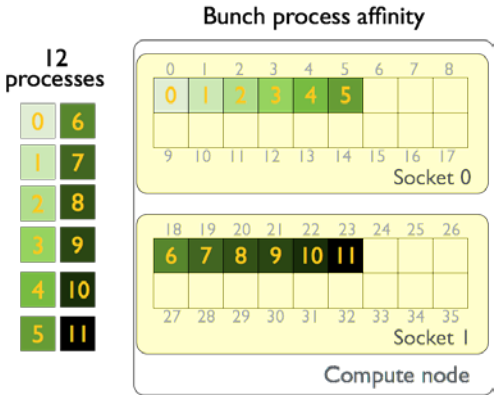


Fig. 105 Schematics of 12 processes distributed with bunch affinity on a MARCONI node. Note that now consecutive processes stay close to each other within each socket.

Finally, it is worth mentioning that there is an alternative way to easily obtain the *scatter* and *bunch* affinities invoking the same Intel MPI library environmental variable, without having to specify explicitly the <*proclist*>, as was done before. Instead, a predefined <*procset*>:<*map*> pair can be invoked. So, for a scatter affinity one can use

```
export I_MPI_PIN_PROCESSOR_LIST=allcores:map=scatter
```

whereas for a bunch affinity one would use

```
export I_MPI_PIN_PROCESSOR_LIST=allcores:map=bunch
```

corresponding to Fig. 110 (right) and Fig. 111, respectively, when 12 MPI tasks are requested.

It should be noted that there is no `I_MPI_PIN_PROCESSOR_LIST` configuration implicitly exported by the PBS scheduler, nor is `I_MPI_PIN` set, so the recommended practice to be on the safe side is to set them explicitly in the batch script, as done in the examples provided.

12.3.1. Reference

The available options for `I_MPI_PIN_PROCESSOR_LIST=`<*procset*>:<*map*> are

- <*procset*>=`all`, `allcores`, `allsocks`
- <*map*>=`bunch`, `scatter`, `spread`.

For more details consult the reference guide [intel1].

12.4. *Hybrid task/thread affinity*

So far we have been dealing with pure OpenMP or pure MPI codes, whose basic pinning strategies were discussed in Secs. 12.2 and 12.3, respectively. Now we move to the hybrid case of mixing both parallelization paradigms. The Intel MPI library provides alternative environment variables that allow the interoperability with the OpenMP API. Namely, `I_MPI_PIN_DOMAIN` defines the concept of *domains*, which are non-overlapping subsets of logical processors within a node, with each one mapped to a single MPI task that can have OpenMP threads below. Such domains can then be pinned with a given affinity. The threads spawned inside them can also be pinned using `KMP_AFFINITY`. It is noteworthy that, if the `I_MPI_PIN_DOMAIN` environment variable is defined, then the `I_MPI_PIN_PROCESSOR_LIST` setting is ignored. In other words, the Intel MPI process affinity environment variables that are interoperable with OpenMP supersede the ones described in the previous section. Actually, they can even be used as an alternative for pure MPI code, provided that, either the domain environment variable is set to `I_MPI_PIN_DOMAIN=core`, or that it is set to `I_MPI_PIN_DOMAIN=omp` with the number of OpenMP threads set to

`OMP_NUM_THREADS=1`, even if the code being ran does not use OpenMP threads whatsoever. Both alternatives essentially say that each MPI *domain* maps to one core only. The reason we dedicated the previous section to the case of pure MPI codes is that the concepts and subtleties just described can simply be ignored by their users. However, for a user/developer of a hybrid code, even if running it just in pure MPI or OpenMP mode, it makes sense to have a single script that works for all situations (pure OpenMP, pure MPI or hybrid), rather than having three different scripts to contemplate each separate case. The purpose of this section is to lay down the basic rules to devise such a baseline "hybrid" script.

Let's start with the case of a pure MPI code and mimic the last two affinity configurations of Sec. 12.3. By replacing the line `I_MPI_PIN_PROCESSOR_LIST=`<*procset*>:<map>, in the script of Table 14 with

```
export I_MPI_PIN_DOMAIN=omp        |    export I_MPI_PIN_DOMAIN=core
export I_MPI_PIN_ORDER=scatter     |    export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=1           |    <leave OMP_NUM_THREADS unset>
```

or with

```
export I_MPI_PIN_DOMAIN=omp        |    export I_MPI_PIN_DOMAIN=core
export I_MPI_PIN_ORDER=bunch       |    export I_MPI_PIN_ORDER=bunch
export OMP_NUM_THREADS=1           |    <leave OMP_NUM_THREADS unset>
```

we obtain the *scatter* affinity of Fig. 110 (right), or the *bunch* affinity of Fig. 111, respectively. The default value for `I_MPI_PIN_ORDER` is *compact*. Note that in the above script lines on the right side, not explicitly setting the number of threads is equivalent to setting them to `OMP_NUM_THREADS=ncpus`, which is the default value. However, this does not affect the size of the domain, which in this case is set independently of the number of threads. Note also that besides the *bunch* ordering, there is also the *spread* ordering available, but on MARCONI's dual-socket node architecture (see Fig. 115), they are in practice almost equivalent.

To generalize this to an hybrid OpenMP/MPI code is pretty straight forward and the easiest way to show this is to provide an example of the corresponding MARCONI batch script, just like we did in previous sections. Let's assume that we want to run four MPI tasks, each spawning three OpenMP threads, for a total of 12 processes. The script in Table 15 does so by creating four *domains*, each with the size of the three OpenMP threads, and then pinning them with a *compact* (or *bunch*) affinity. Inside each *domain*, the threads are pinned with *compact* affinity, for the resulting hybrid process pinning affinity illustrated on the left (or right) side of Fig. 112.

It is worth noting the reason why we used always `KMP_AFFINITY=compact` with the hybrid script examples. On MARCONI, since the first level of non-shared resources is the L3 cache, which is local to each socket, the only way to obtain a scattered affinity for the threads is if only a single *domain* with more than 18 threads is used. So, in practice, as soon as two or more *domains* (MPI tasks) per node are requested, `KMP_AFFINITY=scatter` yields in practice a *compact* affinity, because each *domain* fits inside a single socket, i.e., there are no non-shared resource at the *domain* level.

```
#!/bin/bash
#$ -cwd
#PBS -N hyb_4_3
#PBS -l select=1:ncpus=36:mpiprocs=4:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export I_MPI_DEBUG=5
export I_MPI_PIN_DOMAIN=omp
export I_MPI_PIN_ORDER=compact    <OR>    export I_MPI_PIN_ORDER=bunch
export OMP_NUM_THREADS=3
export KMP_AFFINITY=verbose,compact
BIN=a.out
CMDLOPT=''
mpirun -prepend-rank ./$BIN $CMDLOPT
```

Table 15 Basic script used to run the hybrid code `a.out` with four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with *compact* <OR> *bunch* affinities on a MARCONI node.

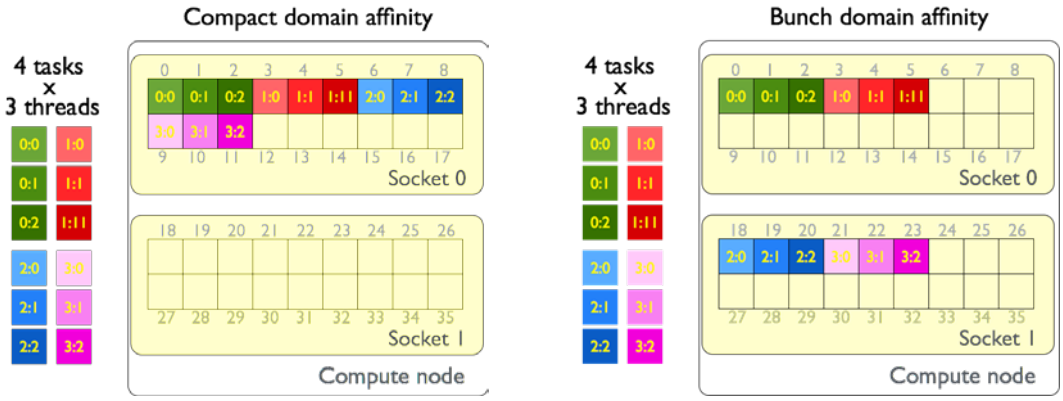


Fig. 106 Schematics of four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with *compact* (left) and *bunch* (right) affinities on a MARCONI node. The numbers inside the coloured squares represent <*task rank*>:<*thread index*>.

To finalize this section, we refer to the most flexible way to specify the affinity of the MPI domains. Rather than relying on the `I_MPI_PIN_ORDER` presets used before, one can directly specify a mask domain, pretty much like we did before in Sec.12.3 for the pure MPI code using the environment variable `I_MPI_PIN_PROCESSOR_LIST`. In the hybrid case this is however slightly more complicated because, instead of providing a list of cores, we need to provide instead an hexadecimal mask describing each domain requested. So if, like before, four MPI tasks are used each with three OpenMP threads, this means we need four *domains* and so, we need to provide four masks, with the syntax `I_MPI_PIN_DOMAIN=[mask0,mask1,mask2,mask3]`. In binary format, each core

in a MARCONI node is represented by a sequence of 35 zeros and a one in the position corresponding to that core's logical number. For instance, core 0 has the binary mask "`00000000000000000000000000000001`", core 1 has the binary mask "`00000000000000000000000000000010`", all the way to core 35, whose binary mask is "`10000000000000000000000000000000000`". If we want to specify the four domains represented in Fig. 113 in terms of their binary masks, these would be:

- domain rank 0 (green): cores {0,1,2}
  binary mask: "`00000000000000000000000000000111`"

- domain rank 1 (red): cores {6,7,8}
  binary mask: "`00000000000000000000000111000000`"

- domain rank 1 (blue): cores {12,13,14}
  binary mask: "`00000000000000000001110000000000`"

- domain rank 1 (magenta): cores {18,19,20}
  binary mask: "`00000000000000011100000000000000`".

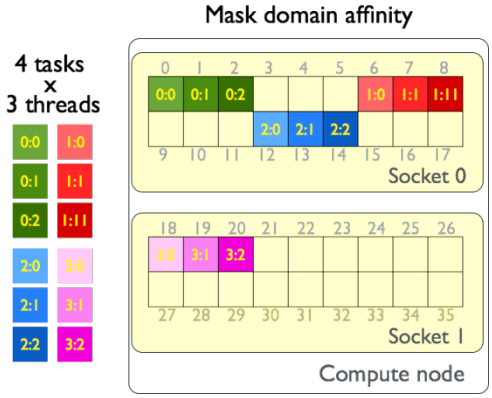


Fig. 107 Schematics of four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with a tailored affinity on a MARCONI node.

These four masks in binary basis are quite long. It is much more convenient to represent them in a hexadecimal basis, in which case they take the more condensed representation `[7,1C0,7000,1C000]`. This is precisely the format required in the script. Hence the batch script corresponding to the process affinity of Fig. 113 is obtained by replacing the blue coloured lines in Table 15 with the following ones

```
export I_MPI_DEBUG=5
export I_MPI_PIN_DOMAIN=[7,1c0,7000,1c0000]
export OMP_NUM_THREADS=3
export KMP_AFFINITY=verbose,compact
```

12.4.1. Reference

The available options for `I_MPI_DOMAIN=`<*size*> are

<*size*>=`omp`, `auto`, <n>

and for `I_MPI_PIN_ORDER=`<*order*> are

<*order*>=`compact`, `bunch`, `scatter`, `spread`, `range`.

For more details consult the reference guide [intel2].

12.5. ***Single-node scaling results in MARCONI***

We already noted in the introduction that, under NUMA architectures, a processor (core) can access its own local memory (within the socket) faster than non-local memory (across sockets). Therefore, as also emphasized there, pinning affinity strategies can be of importance performance-wise. However, the choice of the best process affinity obviously depends on the code being used. If an algorithm implies strong local data dependencies and high arithmetic intensities (CPU-bound), then using a compact affinity to have the neighbouring processes mapped as closely as possible in terms of the hardware cores is probably advantageous. On the other hand, if access to main memory is the bottleneck of the algorithm (memory-bound), then having the processes distributed over both sockets with a *scatter* or *bunch* affinity is most probably a better solution, since within one socket, only half the memory bandwidth of the full node can be achieved.

Here we shall test these concepts in practice by using the hybrid MPI/OpenMP REFMULXP code [dasilva2015], whose underlying memory-bound algorithm makes it well suited to illustrate the effect of the different process affinities on the overall performance. At the same time, it provides a real-world example with an actual finite-differences-time domain (FDTD) wave-propagation production code, which is used for physics studies. Examples of similar analysis using the standard STREAM benchmark [stream] can be found elsewhere.

Being that REFMULXp can be compiled as a pure threaded code using OpenMP, as a pure MPI code or using both paradigms for a hybrid parallel configuration, it allows us to test all three cases separately and then compare them. This is what is shown in Fig. 114. A detailed discussion of the results, for which better statistics than the “single run per data point” used here would be required, goes beyond the scope of this writeup. Instead, what is intended is to highlight the influence of the pinning strategies on the performance of the code. It is clear from the plot that, as expected, when the processes are distributed compactly on the hardware cores, the time to run an REFMULXp simulation is higher than when they are evenly distributed across sockets. The two main reasons which justify this difference are that, to access the full main memory bandwidth both sockets need to be used, and that the main memory bandwidth increases nonlinearly with the number of cores used, with roughly 8 cores per socket being enough to saturate it within each socket for the algorithm at hand. In light of this, it is easy to understand that a scatter or bunch process affinity, which always distributes the work load evenly among both sockets, provides a higher bandwidth per core value than a compact affinity, for which the first socket necessarily has a heavier load, except when the full node is used. The maximum difference is therefore yielded when exactly half a node is used (18 cores).

Another concept whose effect, although less relevant, still seems to be measurable on this algorithm, is the data locality. Comparing the compact and scatter affinities for the pure threaded case when the whole node is used (rightmost point in the red curves), it seems to be advantageous to have consecutive threads placed in consecutive cores, rather than scattering them across sockets (recall the difference between Fig. 110 and Fig. 111).

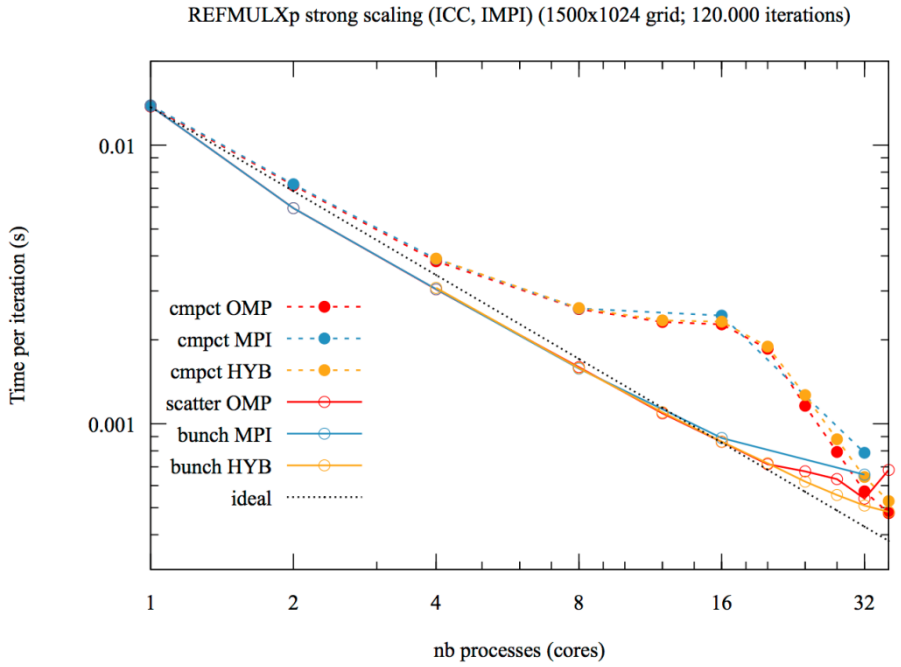


Fig. 108 MARCONI intra-node strong scaling REFMULXp results using different parallelization paradigms (pure OpenMP, pure MPI and hybrid MPI/OpenMP) with several process binding affinities. The hybrid configuration has four MPI tasks and varies the number of threads, which corresponds to the examples given in Fig. 112 and Fig. 113 when three threads are used.

12.6. ***Summary and outlook***

The investigation on process pinning was made to provide practical guidelines on how to use the process pinning options within the Intel MPI library together with the PBS batch queueing system available on the MARCONI supercomputer. Examples of scripts for pure OpenMP, pure MPI and hybrid OpenMP/MPI cases were given for process affinities which are typically relevant for the codes used within the community. Their impact on code performance was then illustrated with strong scaling results from an FDTD production code on MARCONI. For more detailed information on the Intel MPI library and OpenMP environment variables, the reader is referred to the corresponding technical reference guides listed in the bibliography.

12.7. ***References***

[dasilva2015] F. da Silva, M. Campos Pinto, B. Després and S. Heuraux, *J. Comp. Physics*, **295**: 24-45 (2015).

[stream] John D. McCalpin “STREAM: Sustainable Memory Bandwidth in High Performance Computers" *University of Virginia* (1991-2007)

[intel0] https://software.intel.com/en-us/node/528816

[intel1] https://software.intel.com/en-us/node/528818

[intel2] https://software.intel.com/en-us/node/528819

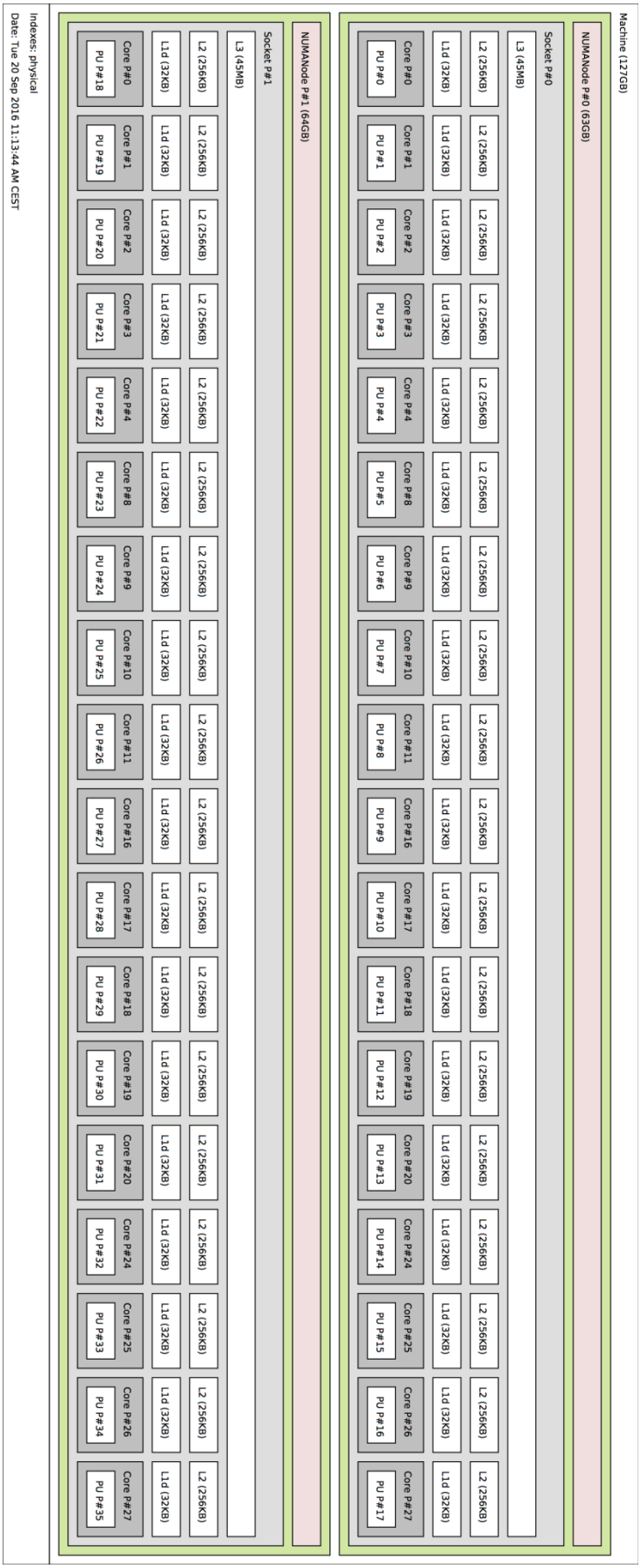


Fig. 109 Schematics of a MARCONI node obtained using the hwloc software package, by invoking the command: “lstopo --no-io --no-icaches lstopo_MARCONI". Note the difference between the (somewhat odd) physical numbering of the cores (Core P#), compared to the logical numbering of the processing units (PU P#) below.