EUROfusion

EUROFUSION WPISA-REP(16) 16235

R Hatzky et al.

HLST Core Team Report 2015

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 2015

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
3DPSOLV	Kan Seok Kang	finished
BEUIFERC	Serhiy Mochalskyy	finished
COCHLEA	Michele Martone	finished
FWTOR-15	Michele Martone	finished
JORSTAR	Serhiy Mochalskyy	running
MGBOUT	Kab Seok Kang	finished
SOLPSOPT	Tamás Fehér	running
TOPOX2	Tiago Ribeiro	finished
VIRIATO	Tiago Ribeiro	finished

Table 1 Projects distributed to the HLST core team members.

Roman Hatzky has been involved in the support of the European users on the IFERC-CSC computer. 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 SOLPSOPT project.

The SOLPS code package is widely used to simulate Scrape-Off Layer (SOL) plasmas. Two main components of this package are the B2 and the EIRENE codes. B2 is a plasma fluid code to simulate edge plasmas and EIRENE is a kinetic Monte-Carlo code for describing neutral particles. EIRENE is parallelized with MPI, and the B2 code is parallelized using OpenMP. The SOLPSOPT project is a continuation of the PARSOLPS project, aiming to further improve the parallelization and decrease the execution time of B2 and the coupled B2-EIRENE system.
More than 25 subroutines have been parallelized, and 90% of parallelism has been reached in the whole B2 code. With these changes a factor of six speedup could be achieved for the ITER test case when executed on a single compute node. Several of the subroutines were optimized to reach a speedup which is close to the bandwidth limit. As a positive side effect, these optimizations led to a speedup of the sequential version of the code, it is now 20% faster than originally.
The individual subroutines have been tested separately with unit tests to ensure the correctness of the modified B2 code. The whole code was tested for both shorter and longer simulations, and the results agree with the original code.
Unfortunately, the MPI version of EIRENE in the SOLPS 5.0 code package was not functioning, so it was decided to switch to the latest version of EIRENE from SOLPS-ITER. The parallel performance of the stand-alone version of EIRENE was investigated, and we have found that the currently implemented parallelization strategies do not scale due to load imbalance. A simple and balanced parallelization strategy was implemented instead. Tests with different particle numbers show that it provides reasonable speedup.
During the tests it was found that the parallel version of EIRENE gives incorrect results, even if we use the original code version. A testing framework is under construction to find the underlying errors.

Serhiy Mochalskyy worked on the BEUIFERC and JORSTAR projects.

In the BEUIFERC project we analyzed the performance of the MIC partition of the HELIOS supercomputer by means of different micro-benchmark tests. The network performance was tested first by means of the Intel MPI Benchmark suite. It was found that the two available Infiniband ports per node were not working properly since they were providing a too low memory bandwidth. A software solution could be provided resulting in a three times higher memory bandwidth. The different OpenMP pragma overheads were measured on both the Sandy Bridge node and the MIC card using the EPCC micro-benchmark suite. The overhead time is more than a factor of ten higher on the MIC than on the Sandy Bridge processor due to the large number of threads.
New features of the MPI 3.0 standard (the collective communication, the remote memory access and the shared memory segments) were also tested on the HELIOS computer. For collective communication an efficient communication-calculation overlap (85–99 percent) could be achieved only with large message sizes (>10 KB), while for small messages this overlap was between 20 and 55 percent. The passive target communication in the remote memory access of the MPI 3.0 standard works properly only when the asynchronous progress support is switched on allowing to achieve more than 95 percent overlap between calculation and synchronization. Finally, the implementation of the shared memory segments by means of the MPI standard 3.0 was also tested on HELIOS. The tests provided correct results showing that all MPI tasks can have direct access to a shared memory region being created initially by a master task.

In the JORSTAR project the STARWALL code has been analyzed for potential improvements and optimization by means of MPI parallelization. For large production runs the entire code must be parallelized due to local memory restriction for saving the input/output matrices and due to the computational time for different subroutines. The LAPACK subroutine for the eigenvector solver was replaced by its parallel counterpart from the ScaLAPACK library. Interestingly the ScaLAPACK subroutine has also shown better performance in sequential mode due to the advantage of using IEEE arithmetics. Finally, good parallelization efficiency was obtained for the solver subroutine for large problem sizes.
Several subroutines were re-written in order to avoid the building up of too large matrices. In combination with the MPI parallelization concept this should finally bring the possibility to perform simulations of ITER sized problems. A good parallel scalability was achieved in all modified subroutines with a speed-up factor of more than 210 when using 512 cores.

Kab Seok Kang worked on the 3DPSOLV and MGBOUT projects.

Kab Seok Kang has visited the project coordinator to discuss the current status of the BIT2 code and the future plans. The project coordinator is currently testing the BIT2 code with the multigrid solver which was developed in the last project KinSOL2D-3. It was especially designed to suit the current BIT2 code structure. Kab Seok Kang will further support the project coordinator to bring BIT2 in its multigrid version to production stage.
The implementation of the 3-dimensional multigrid solver with the finite difference and finite element method is at the moment at the stage of building up the according matrix equation and intergrid transfer operators. However, the project duration comes to an end and the work has to stop here. A succeeding project will be needed to finish the implementation and testing of the 3D multigrid solver in BIT3. This is not critical as the project coordinator is still busy with testing BIT2 with its 2D multigrid solver. In parallel, he will further develop the BIT3 version. Therefore, he did not apply for a new HLST project in the support period of 2016 to continue the work on the 3D multigrid solver. Fortunately, the MGBOUT3D project in 2016 is also about an implementation of a 3D multigrid solver. Thus, the work will continue in a more global

context with clear synergetic effects for a future BIT3 multigrid project to be probably submitted in 2017.

Kab Seok Kang visited the CCFE and gave lectures on basic numerical methods and multigrid methods to the BOUT++ team. He implemented a 1D parallel multigrid solver according to the BOUT++ structure in C++ and tested it on a problem which was provided by the BOUT++ team. The results of the 1D parallel multigrid solver showed that there was a need for a more enhanced parallel multigrid solver. So the structure of the multigrid solver was adapted in the following way. Starting with the finest multigrid level a 1D parallelization is used. From a certain level on the problem becomes too small for a 1D parallelization. Therefore, the data are redistributed and the parallelization concept is switched to a 2D parallelization of the domain. Finally, for the coarsest grid level a serial solver is being used. The numerical results showed that the new hybrid solver has a very good weak scaling property which enables the user to solve larger problems on a large number of MPI tasks in reasonable time scales. The implementation was sent to the project coordinator for testing within BOUT++.

Michele Martone worked on the COCHLEA and FWTOR-15 projects.

COCHLEA is a numerical C++11 code calculating the magnetic and electric fields in a complex waveguide of a cylindrical symmetry with an arbitrary excitation. Most of its execution time is spent in the FDTD algorithm loops repeatedly updating a 3D grid.
We have identified and applied the most relevant changes for achieving parallelism and increased efficiency. Our changes do not alter the code structure (apart from the core loops), are minimally invasive, and allow optional decision of their inclusion/exclusion. This was done in an approach of planning and experimenting 'offline', deploying everything together with the PI during a one week visit, finally culminating in code acceptance on the PI's side.
With representative options, in experiments ranging from toy cases up to a full HELIOS node case, an efficiency improvement (up to ≈ 4x) can be obtained serially, to be combined with a ≈ 10x OpenMP speedup obtained with the 16 threads we had available. So combined, our intervention can yield a ≈ 40x speedup on the largest cases. This allows a large simulation like that of e.g. the last part of the ITER gyrotron beam tunnel to complete in six days instead of a few months. Larger cases and a finer physics model (under active development by the PI) will benefit from further optimizations and distributed memory parallelism (e.g. MPI) in terms of both memory and time constraints.

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.
Our objective was to obtain an MPI parallelization atop of an OpenMP one. We opted for a 2D domain distribution without padding or symmetry constraints. Parallelizing required the introduction of ghost cells around each process subdomain. The new MPI code was placed in a separate file. It is still possible to obtain an MPI-free build by switching off the preprocessor-controlled wrappers to the communication primitives. To keep continuity with the original code we used the original's variables naming convention combined with a global array indexing.
Our largest HELIOS node fitting case (≈ 5e+7 cells) can now be distributed over 32 x 32 = 1024 processes, while OpenMP partitions the workload along one dimension. Such a test case achieved a parallel MPI speedup of ≈ 200x on 512 nodes. Smaller cases have a lower parallelization efficiency, e.g. below 50% already for 32 nodes. The code is now in a state for testing production cases much larger than before.
However, further improvements are possible e.g.: trading off computation for communication, grouping data for fewer messages, or avoiding certain extra

exchanges. The current serialized I/O is a limit for large runs. A parallel I/O is thus necessary.

Tiago Ribeiro worked on the TOPOX2 and VIRIATO projects.

Modern tokamaks have strongly shaped diverted magnetic structures in which the last closed flux surface is a separatrix with an X-point. This leads to high magnetic shear near the X-point region, whose effect on drift-wave turbulence is believed to be severe. This constitutes the framework within which project TOPOX was devised. In particular, the objective is the extension of a Poisson solver based on a method developed by Sadourny et al. This method is built on a triangular grid in *RZ*-space, with the points arranged along flux surfaces, which are topologically treated as hexagons.
First an appropriate stand-alone test-case was implemented on closed flux surfaces using both Sadourny's method together with WSMP and PETSc libraries and an alternative finite-differences pseudo-spectral solver. Their comparison demonstrated the better discretization efficiency of the former. A scheme for the extension of the Sadourny's method beyond the X-point, into the SOL was then devised. It consists in setting an artificial boundary between the X-point and a grid node in the outermost SOL flux surface that is topologically treated as a hexagon. This ensures the hexagonal topology of every node and its six nearest neighbors on the grid (including the SOL), allowing to carry the method with only minor modifications. For the code implementation, several simplified analytical poloidal flux functions were devised to yield a set of nested flux surfaces mimicking a diverted tokamak equilibrium. A hexagonal mesh was constructed using an equidistant arc-length between grid-nodes along the field lines to minimize the local deformation of the elemental hexagons.
A potential issue related to local grid deformation was identified near the X-point. There was no time to fully test this problem, but it is clear that in the case where it reveals itself unsurmountable, new solutions have to be devised. Either by using different poloidal distribution of nodes along the mesh's main hexagons, or by locally relaxing the constraint that each grid node has six nearest neighbors.

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. It comprised devising more efficient alternatives for the 2D Fourier transform algorithm, mostly by modifying the parallel data transpose implied in these operations. The main idea was to aggregate data before carrying the all-to-all communication patterns to avoid penalizations due to network latency accumulation. The deployment of the new algorithms back into VIRIATO's source code was made successfully. Compared to the original version, VIRIATO now runs the same problem size using double the number of cores while still maintaining perfect linear strong scaling.
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. These approaches are promising, but due to time constraints, their implementation is left as a recommendation for future work.

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

1.2.1. **Dissemination**

Mochalskyy, S. and Hatzky, R: The experience of the High Level Support Team (HLST), *3rd IFERC-CSC Review Meeting*, 9th March 2015, Naka, Japan.

Mochalskyy, S. and Hatzky, R.: Simulation using MIC co-processor on Helios, *IFERC-CSC HELIOS MIC Workshop,* 17th–18th March 2015, Maison de la Simulation, Gif-sur-Yvette, France.

1.2.2. **Training**

Tiago Ribeiro has visited:

- The project coordinator Nuno Loureiro to work for the VIRIATO project, 5th–25th July 2015, Lisbon, Portugal.

Kab Seok Kang has visited:

- The project coordinator David Tskhakaya to work for the 3DPSOLV project, 12th–14th January 2015, University of Innsbruck, Innsbruck, Austria.
- The project coordinator Fulvio Militello to work for the MGBOUT project, 16th–27th March 2015, CCFE, Culham, UK.

Michele Martone has visited:

- The project coordinator Christos Tsironis to work for the FWTOR-15 project, 31st August – 4th September 2015, National Technical University of Athens, Greece.
- The project coordinator Ioannis Tigelis to work for the COCHLEA project, 23rd–27th November 2015, National and Kapodistrian University of Athens, Greece.

1.2.3. **Internal training**

The HLST core team has attended:

- IFERC-CSC HELIOS Training, 27th–28th January 2015, IPP, Garching, Germany.
- HLST meeting at IPP, 15th April 2015, Garching, Germany.
- HLST meeting at IPP, 14th October 2015, Garching, Germany.

T. Fehér, Intel HPC Code Modernization Workshop, 19th–20th November 2015, Garching, IPP, Germany.

Kab Seok Kang has attended:

- HPC Programming-Workshop, 6th–7th May 2015, LRZ, Garching, Germany.
- MPK Symposium on Frontiers in Materials Science, Optics, Materials and Functionality, 29th–30th June 2015, POSCO International Center, Pohang, Korea.
- Joint Seminar of Max Planck POSTECH/Korea Research Initiative and Pohang Mathematics Institute on “High Performance supercomputer: theory and application”,1st July 2015, POSTECH, Pohang, Korea.
- 23rd International Conference on Domain Decomposition Methods, 6th–10th July 2015, International Convention Center Jeju (ICC-Jeju), Jeju Island, Korea.
- C++ for Beginners Course, 19th–22th October 2015, LRZ, Garching, Germany.
- Advanced C++ with Focus on Software Engineering, 2nd–4th November 2015, LRZ, Garching, Germany.

Michele Martone has attended:

- Advanced OpenMP@EPCC, 2nd–3rd July 2015, University of Manchester, Manchester, UK.
- Advanced C++ with Focus on Software Engineering, 2nd–4th November 2015, LRZ, Garching, Germany.

Serhiy Mochalskyy has attended:

- 17th VI-HPS Tuning Workshop, 23rd–27th February, 2015, HLRS, Stuttgart, Germany.
- Intel MIC&GPU Programming Workshop, 27th–29th April 2015, LRZ, Garching, Germany.
- HPC Programming-Workshop, 6th–7th May 2015, LRZ, Garching, Germany.
- PRACE PATC Course: Node-Level Performance Engineering, 10th–11th December 2015, LRZ, Garching, Germany.

Roman Hatzky and Kab Seok Kang have attended:
Numerical Methods for the Kinetic Equations of Plasma Physics (NumKin2015), 26th–30th October 2015, IPP, Garching, Germany.

1.2.4. **Workshops & conferences**

Roman Hatzky has attended:
IPP Theory Meeting, 23rd–27th November 2015, Plau am See, Germany.

Fehér, T., SOLPS Parallel Optimization, *SOLPS Optimization Working Session*, 30th November 2015, IPP, Garching, Germany.

Fehér, T., Parallel Optimization, *NMPP Seminar*, 15th December 2015, IPP, Garching, Germany.

Kang, K.S., Scalable implementation of the parallel multigrid method on massively parallel computers, *SCENT HPC Summer School @GIST*, 15th–17th July 2015, GIST, Gwangju, Korea.

Kang, K.S., Scalable implementation of the parallel multigrid method on massively parallel computers, *8th Euro-Korean Conference on Science and Technology*, 22nd–24th July 2015, Faculte de Medecine, Strasbourg, France.

Martone, M.: Auto-tuning shared memory parallel Sparse BLAS operations in librsb-1.2, *Int. workshop on Sparse Solvers for Exascale: From Building Blocks to Applications*, 23th–25th March 2015, Greifswald, Germany.

F. da Silva, S. Heuraux, T. T. Ribeiro and D. Aguiam: REFMULF: 2D Fullwave FDTD Full Polarization Maxwell Code, *Proc. 42nd EPS Conference on Plasma Physics*, 22nd–26th June 2015, Lisbon, Portugal.

F. da Silva, T. Ribeiro, S. Heuraux and B. Scott: REFMULF: 2D Fullwave FDTD Full Polarization Maxwell Code for a synthetic reflectometry diagnostic, *16th European Fusion Theory Conference*, 15th–18th October 2015, Lisbon, Portugal.

Ribeiro, T., Improving the scalability of the VIRIATO code*, IPP Theory Meeting*, 23rd–27th November 2015, Plau am See, Germany.

1.2.5. **Publications**

Heuraux, S., da Silva, F., Ribeiro, T., Despres, B., Campos Pinto, M., Jacquot, J., Faudot, E., Wengerowsky, S., Colas, L., and Lu, L.: Simulation as a tool to improve wave heating in fusion plasmas, J. Plasma Phys. 81, 2015, Art. No. 435810503.

Kang, K. S.: On the finite volume multigrid method: comparison of intergrid transfer operators, Computational Methods in Applied Mathematics **15**, p. 189–202, 2015.

Kang, K. S.: Scalable implementation of the parallel multigrid method on massively parallel computer, Computers and Mathematics with Applications **70**, p. 2701–2708, 2015.

Sonnendrücker, E., Wacher, A., Hatzky, R., Kleiber, R.: A split control variate scheme for PIC simulations with collisions, Journal of Computational Physics **295,** p. 402–419, 2015

1.2.6. **Meetings**

Roman Hatzky attended on a regular basis:

- CSC management meeting
- CSC European ticket meeting

2. **Report on HLST project SOLPSOPT**

2.1. *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.

SOLPS is a collection of several codes; two main components are the B2 and the EIRENE codes. B2 is a plasma fluid code to simulate edge plasmas and the EIRENE code is a kinetic Monte-Carlo code for describing neutral particles. In sequential mode it can take 1–12 weeks to converge for ITER or DEMO simulations. While EIRENE is parallelized with MPI, the B2 code was previously only partially parallelized using OpenMP, and because of this parallel simulations were not practical. In 2014, the parallelization of the B2 code was considerably improved in the framework of the PARSOLPS project, and a factor of six speedup has been reached for the ITER test case (Fehér, 2015).

The SOLPSOPT project is a continuation of the PARSOLPS project, aiming to further improve the parallelization and decrease the execution time of B2 and the coupled B2-EIRENE system. Additional subroutines were parallelized in B2 and the sequential execution time of B2 was improved. The first part of the report discusses the changes in the B2 code, and gives an overview of the total speedup. The modified code was extensively tested, and the reproducibility of the results is discussed in detail.

According to the original work plan, the work should have continued by coupling the modified B2 code with the MPI version of EIRENE code. Afterwards, further optimization of B2 was planned with more drastic code changes. Unfortunately the MPI version of the EIRENE code was not working as expected, therefore we had to focus on fixing the MPI scaling of EIRENE. The second part of the report discusses the work related to the EIRENE code.

The work presented here was carried out in collaboration with Lorenz Hüdepohl from the Max Planck Computing and Data Facility (MPCDF).

2.2. *OpenMP parallelization of the B2 code*

The OpenMP parallelization of B2 was improved by increasing the parallel fraction of the code. The difficulty was that the execution time is distributed over many small subroutines. We have measured how long different subroutines take to execute in sequential mode[1], this is shown in the upper part of Fig. 1. The subroutines parallelized previously by F. Reid (Reid, 2010) are represented by the blue boxes. We can see several other serial subroutines (red boxes) where the code spends 1–7% of the execution time, and most of them need to be parallelized to achieve a significant speedup. This work was shared among T. Fehér and L. Hüdepohl. During the PARSOLPS and SOLPSOPT project we have parallelized more than 25 additional subroutines. The lower part of Fig. 1 shows how the execution time in the parallelized code is divided between different subroutines. The green boxes show the additional parallel subroutines. Note that some of the parallelized subroutines still have a sequential part. Due to single core optimization, the sequential execution time became significantly shorter for B2STBC_PHYS and B2SIFR (see Section 2.2.1). In

[1] The exact number depends on the optimization flags, the test case and the number of time steps taken in the simulation (among other factors). For the measurements shown here, the code was compiled with *–O3 –xavx* flags and ten time steps were taken in the ITER test case. The FTIMINGS tool from L. Hüdepohl was used to make the measurements.

the modified code, around 90% of the execution is spent in parallel regions (measured on one thread).

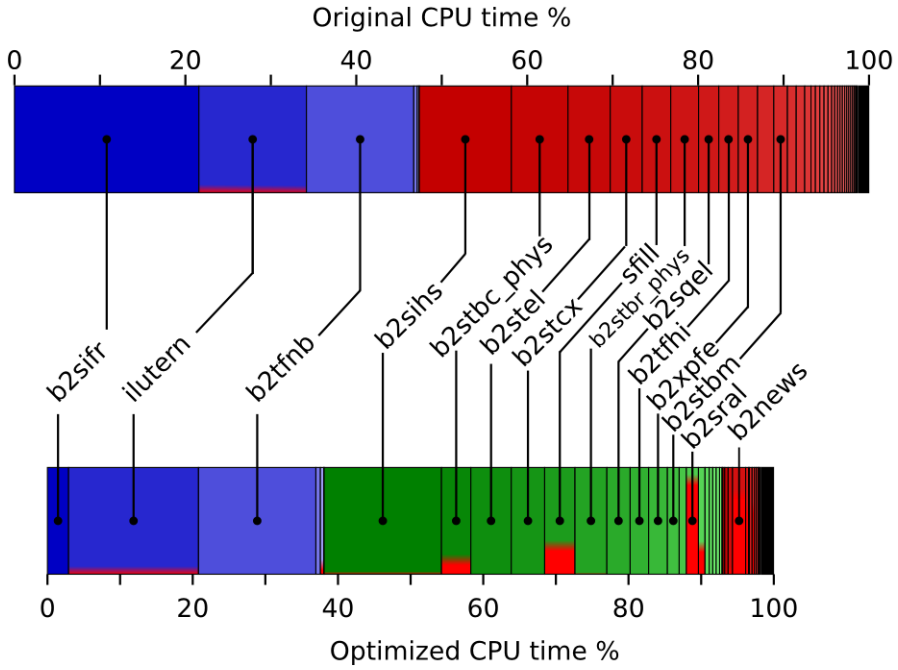

Fig. 1 Relative execution time of subroutines in the B2 code. The upper part of the figure shows the status at the beginning of the PARSOLPS project, the lower part shows the final status. The colored boxes represent different subroutines, and the width of the boxes indicates the share from the execution time. The blue boxes were parallel when the project started; the red boxes denote the sequential subroutines, and the green boxes show the newly parallelized subroutines. The boxes that contain both red and blue or green colors are parallelized subroutines which still have a sequential part. Due to sequential optimization, the execution time of certain subroutines became shorter.

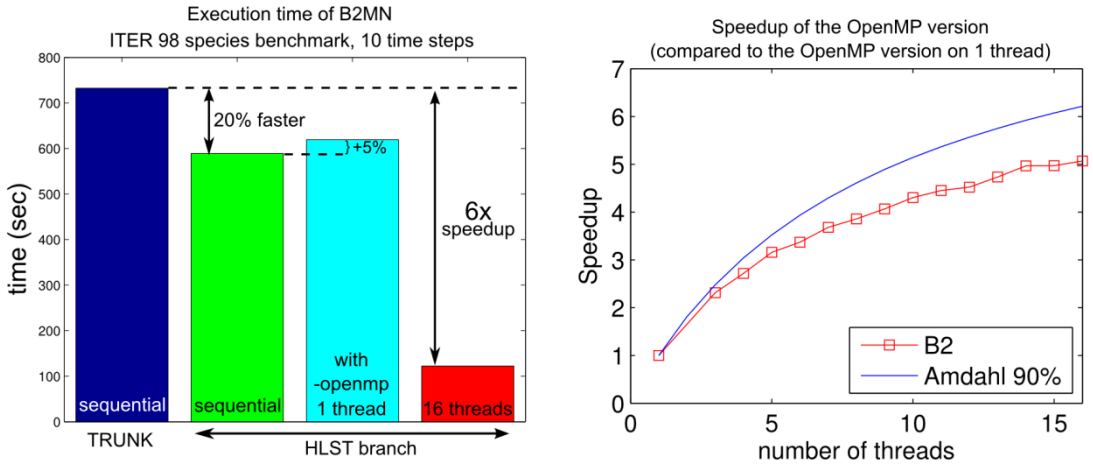

Fig. 2 Execution time and speedup of the B2 code for the ITER test case. Left: the original (trunk) version of the code is compared to the OpenMP parallelized (HLST) version. Right: strong scaling of the OpenMP code. The blue curve shows the theoretical estimate for ideal scaling (Amdahl's law for 90% parallel code).

Fig. 2 shows the speedup of the OpenMP version of the B2 code. In the left subfigure, the speedup of the optimized OpenMP code (HLST branch) is compared to the original code (TRUNK). If we compile the code in sequential mode (without the `-openmp` flag) then the optimized code is 20% faster. Adding the `–openmp` flag

introduces 5% overhead. The maximum speedup on 16 threads is a factor of six compared to the TRUNK version.

The right side of Fig. 2 shows the speedup of the OpenMP code, as a function of the number of threads. The blue curve shows the theoretical maximum speedup according to Amdahl's law.

2.2.1. **Execution time on one core**

We have two options to run the code on one core: compile it without the –openmp flag (referred as sequential version in the following) or compile with –openmp flag and set the number of threads to one. The OpenMP version is expected to take slightly longer than the sequential version due to the overhead of creating the OpenMP regions. Contrary to the expectations a significant difference was found during the development: the OpenMP version took 20% longer than the sequential version. Table 2 lists the subroutines where the difference is larger than 0.5 second. Most of the overhead came from the subroutines B2SIFR and B2TFNB. These subroutines were parallelized earlier by F. Reid (Reid, 2010), and since they scaled well they were not investigated before.

sequential time (s)	omp time (s)	subroutine name	difference (s)
77.2	187.8	B2SIFR	110.7
96.8	115.5	B2TFNB	18.7
108.1	110.3	ILUTERN	2.1
3.1	5.1	B2USMO	2.0
95.6	96.9	B2SIHS	1.3
1.5	2.4	B2USCO	0.9
25.3	26.1	B2STCX	0.7

Table 2 Execution time of different subroutines in the sequential and OpenMP version of the B2 code (ITER 98 species benchmark, 10 time steps)

The execution time of the B2SIFR subroutine has 110.7 seconds of difference. This function is used to calculate the linearized representation of the friction force between all particle species. There are no OpenMP regions inside B2SIFR. Instead, it is called from an OpenMP region as illustrated in Fig. 3.

```
!$OMP PARALLEL DO
do is=0, ns-1
! …
  call b2sifr (…,is,…)
! …
enddo
```

Fig. 3 The B2SIFR subroutine is called from an OpenMP region. Inside B2SIFR there are no OpenMP directives.

Each call to B2SIFR calculates the friction force exerted on particle species is. It turned out that the compiler uses different optimization when the subroutine is called from an OpenMP region, and this causes difference in the execution time. In case of B2SIFR most of the time was spent in calculating the Coulomb logarithm. A closer look revealed that it is not even necessary to recalculate the Coulomb logarithm in each iteration since it is only dependent on the background plasma temperature and density. The code was modified accordingly, and the execution time is shown in Fig. 4.

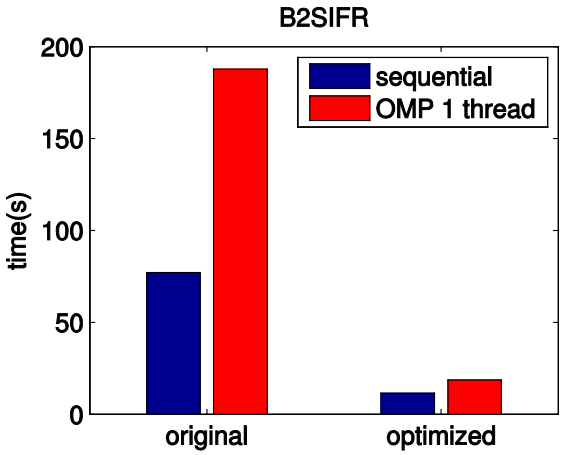

Fig. 4 Execution time of B2SIFR in its original form (left) and after optimization (right). ITER 98 species test case, 10 time steps.

The overall execution time is reduced, but the problem with the OpenMP overhead remains: the code called from the OpenMP region takes longer to execute. Nevertheless, the severity of the problem decreased: compared to the execution time of the whole code the overhead from the B2SIFR subroutine is now only 1%.

With fine tuning this overhead can be probably further decreased. For example many of the subroutines called from B2SRAL were optimized and for those subroutines the execution time of the OpenMP version is slightly faster (see Fig. 5) than the execution time of the sequential version. This optimization work is very time consuming and only expected to improve the speedup by a few percent; therefore it was decided not to continue it.

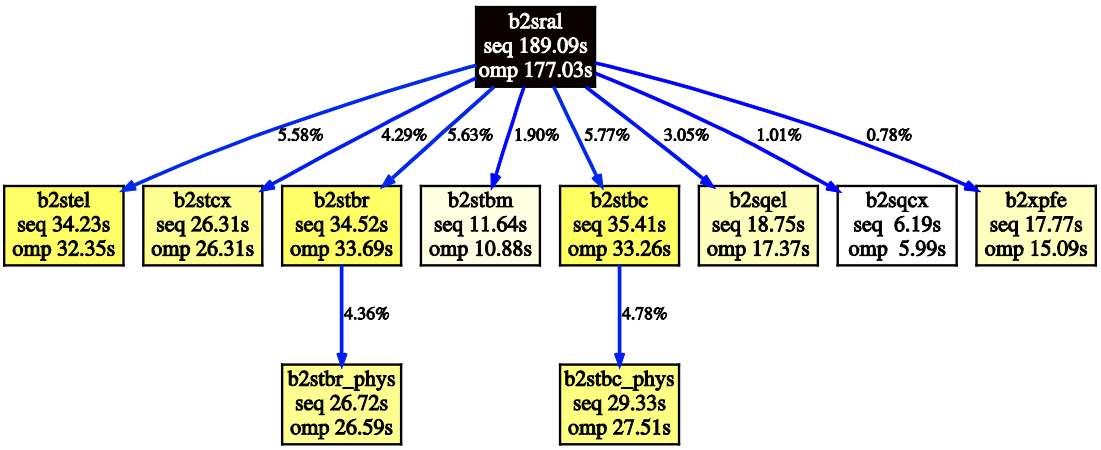

Fig. 5 Execution time of subroutines called from B2SRAL. The sequential code (compiled without –openmp) is compared to the OpenMP code using one thread. Each box represents a subroutine, the first line in the box is the name of the subroutine, the second line shows the execution time of the subroutine in sequential mode, and the third line is the execution time in OpenMP mode. The times are total times including the time spent in the called subroutines. The code was executed four times and the results from the fastest run are shown in the figure.

After improving B2SIFR the overhead of the OpenMP version of the B2 code is 5%. This is shown in Fig. 2 as the difference between the green and the light blue bars. The improvements in B2SIFR did not improve the overall speedup. The previous version of B2SIFR had super linear scaling and the overhead disappeared when the code was executed on a large number of threads. So the overall speedup of the HLST version of the code using 16 threads remains a factor six compared to the trunk version.

2.3. *B2 in the HLST branch of the SVN repository*

The development of the B2 code was done using the GIT repository provided by MPCDF. There were changes in the original code base since we started our work. These changes (up to revision 4794) were merged with the development version of the code. Afterwards the code with the OpenMP modification was uploaded step by step into the HLST branch of the SVN repository. Here we describe some of the steps separately.

2.3.1. **Modules introduced**

Most of the subroutines in the B2 code had originally Fortran 77 style procedure calls: the functions and subroutines were defined as external symbols, no interface information was specified. This was changed by L. Hüdepohl: every file was encapsulated into a module and the *EXTERNAL* statements were replaced by USE statements. This way the compiler gets access to interface information about the procedures and can check whether the argument list corresponds to the procedure definition. The source files of B2 are organized in different subdirectories. The new module names reflect this directory structure; the file *dirname/filename.F* was enclosed into the module called *DIRNAME_FILENAME*.

The use statements explicitly define the dependencies between the source files. This dependency information was used to compile only the modified code parts during the development work. The project coordinator has indicated that having module interfaces would be useful in the official code base, so these changes were committed to the HLST branch in the SVN repository.

The B2 code package contains several executables. During development, we focused on b2mn.exe, where the main calculation takes place. The other code parts had problems during compilation because of the new interface information: some circular dependencies were not resolved and the type kind parameter for certain function calls was not correct. This has been fixed on the revision 4703 of the HLST branch, which contains the code with the module interfaces.

2.3.2. **Problems with the build system**

The development version of the B2 code in the GIT repository was equipped with the FBUILD build system by L. Hüdepohl. It provides faster compilation (up to 20% faster compared to the original build system) and several convenient features that helped testing and debugging.

In the HLST branch of the SVN repository, the original build system was retained. After introducing the new modules the dependency list for the makefile had to be updated. Creating a Fortran dependency list is a nontrivial problem, and there is no standardized way to do it. The Perl scripts of the FBUILD make-system do a decent job in listing the dependencies. In contrast, the original build system uses *grep* commands to list the dependencies, and has the following shortcomings:

- fails if the *USE* statements are indented by more than six spaces,
- assumes that modules are in b2mod_* files,
- system modules should be flagged with `!IGNORE` flag,
- does not understand if some modules are used only conditionally.

Most of these issues could be easily fixed by modifying the *grep* commands.

2.3.3. **R8 parameters included**

The B2 code uses 8 byte reals for the calculation. The precision is explicitly defined by setting the type kind parameter for the variables and constants. These declarations were missing in some parts of the code, which could lead to the problem that is illustrated in Fig. 6. If the type kind is not specified for a constant, then according to the implicit conversion rules of Fortran, it is not specified what will be

assigned to the least significant digits. This happens with variable *a* in Fig. 6. It is a well known feature of Fortran, and it can lead to problems when we want to compare the output of the code: we cannot expect bit identical results. In revision 4712 the missing type kind specifiers were added to the code. This way, the output of the code is the same as if it was compiled with the `-r8` flag.

```
program kindness_test
  INTEGER, PARAMETER :: R8 = SELECTED_REAL_KIND (14)
  real(kind=R8) :: a, b
  a = 42.137
  b = 42.137_R8
  write(*,*) 'Without type kind specifier', a
  write(*,*) '   With type kind specifier', b
end program

Output:
Without type kind specifier   42.1370010375977
   With type kind specifier   42.1370000000000
```

Fig. 6 Usage of type kind specification. The R8 parameter is used to define double precision reals. The constant in the assignment for variable *a* has no type kind specifier, so the value assigned to *a* and *b* are different.

2.3.4. **OpenMP modifications**

The subroutines that have OpenMP modifications are uploaded into the SVN repository in separate commits. The optimized version of B2XPFE, B2STCX, and B2STEL is considerably different from the non optimized versions. For these subroutines a simpler OpenMP version, which contains only small modifications compared to the sequential version, is also available in the repository.

2.3.5. **Different compilers**

The B2 code was compiled with the Intel Fortran compiler (ifort) version 14 and 15 during the development. The project coordinator requested that the code should also be compatible with ifort v12 which is used on the Eurofusion Gateway computer.

To compile with ifort v12, one has to remove the `-align array32byte` flag from the compiler options. The same flag can be specified for ifort v13 or newer.

Apart from the alignment flag, in earlier versions of the optimized code, three of the subroutines (*B2XPFE*, *B2STCX*, *B2STEL*) had `assume_aligned` compiler directives. This is an Intel specific directive that is used to specify the alignment of dummy arguments of the functions. The problem with this directive is that without the `-align array32byte` flag the compiled code would be incorrect. So if the users switch back and forth between ifort v12 and newer versions of ifort, then there is the danger to miss out the `-align array32byte` flag and generate invalid code. A runtime check would catch such an error, but it was considered safer to remove the alignment directives from the source code. This way the compiled code should be always correct. The execution time of the affected subroutines can increase, but the change compared to the overall execution time is negligible.

To compile with the `-openmp` flag using ifort v12, the *OMP PARALLEL DO SIMD* directive in *sources/b2stbm.f* was replaced with *OMP PARALLEL DO*, because the SIMD directive is a new addition in the OpenMP 4.0 standard.

After these considerations the B2 code was compiled and tested with ifort v12, v13.1, v14, v15, and v16b. In sequential mode all versions produce correct results, and the execution time is about the same. All versions work using one thread in OpenMP mode, but if more threads are used, then the code fails if it was compiled with ifort v12 or v13. The code stops execution with the message:

b2xvfx--faulty argument x-edge fne,

which is an internal consistency check in one of the subroutines. The *fne* array is calculated in the *B2XPFE* subroutine. The error occurs with the optimized version of the subroutine (revision 4745). It is believed to be a compiler bug, since the same code works with ifort v14 or newer. The problem was fixed by replacing *B2XPFE* with the non optimized OpenMP version (revision 4744). By switching back to the non-optimized version, theoretically we loose a factor of two speedup for *B2XPFE*. Overall it is a negligible difference, since only 2.5% of the total execution time is spent in *B2XPFE*.

To summarize, the OpenMP version of the B2 code can be compiled with ifort v12 or newer. All of these compilers can generate code for the AVX instruction set which is used in the Sandy Bridge and Ivy Bridge processors. It is however strongly recommended to use the newer compiler versions, since they contain important bug fixes.

2.4. *B2 correctness checks*

During the parallelization work, all the modified subroutines were tested with separate unit tests. The results agree well with the original version of the subroutines up to machine precision. These tests were discussed in the final report of the PARSOLPS project. Even though the separate components of the code were tested with unit tests, we had to ensure that the whole code is correct when compiled with all the modifications. In the following we describe additional tests.

2.4.1. **Benchmarks**

Three test cases were used to test the correctness of the modified code as a whole. They correspond to the test cases in the benchmark folder of the SVN repository, but different numbers of time steps are used:

- AUG_16151_D: ten time steps
- AUG_16151_D+C+He: one time step
- ITER_D+T+He+Be+Ne+W: one time step

To compare the results of the simulation, the output files were investigated. It was not possible to do a simple binary comparison, because the date and time of the simulation were saved into the files. Instead, a small program was created which reads the content of the files, and compares the results from different simulations.

The output of B2 is done both in formatted and unformatted files. The name, size and value of different arrays are saved into the files. The test program is saved under *src/Braams/b2/src_xpb/test/check_b2_output.F90* in the SVN repository, and it was used to compare the following output files:

- b2fstate – provides the final state,
- b2fmovie – provides data for movie,
- b2ftrace – provides data on the numerical evolution,
- b2fplasma – plasma state at the end of the simulation.

During the testing the code was compiled with the following options: `-fp-model source –no-vec -02`. The first option specifies that only value safe optimizations can be used.

After introducing the new modules and the additional type kind specifiers (revision 4712) the code in the HLST branch was compared to the SVN trunk. When we compiled in sequential mode (without `–openmp` flag), the results were binary identical to the results from the trunk version, if the latter was compiled with `–r8` flags. When the `–openmp` flag was included, the results were not identical anymore, but the results were still correct (see section 2.5 for details).

2.4.2. **Long test**

The project coordinator pointed out that previously they have encountered errors which appeared only after a longer simulation, therefore it is recommended to perform tests over 1000 time steps. Such long tests were done for the AUG_16151_D+C+He test case. The results of the HLST branch (containing the OpenMP changes) have been compared to the trunk (original) version of the code. As described in section 2.3.3, some of the type kind specifiers for constants are missing in the trunk version code. This problem was fixed in the HLST version. To be consistent, the trunk version of the code was compiled with the `-r8` flag, this way all constants are interpreted as double precision constants in both code versions. Because of the correction in the precision it takes longer until the residuals are converged. We have performed simulations up to 5000 time steps.

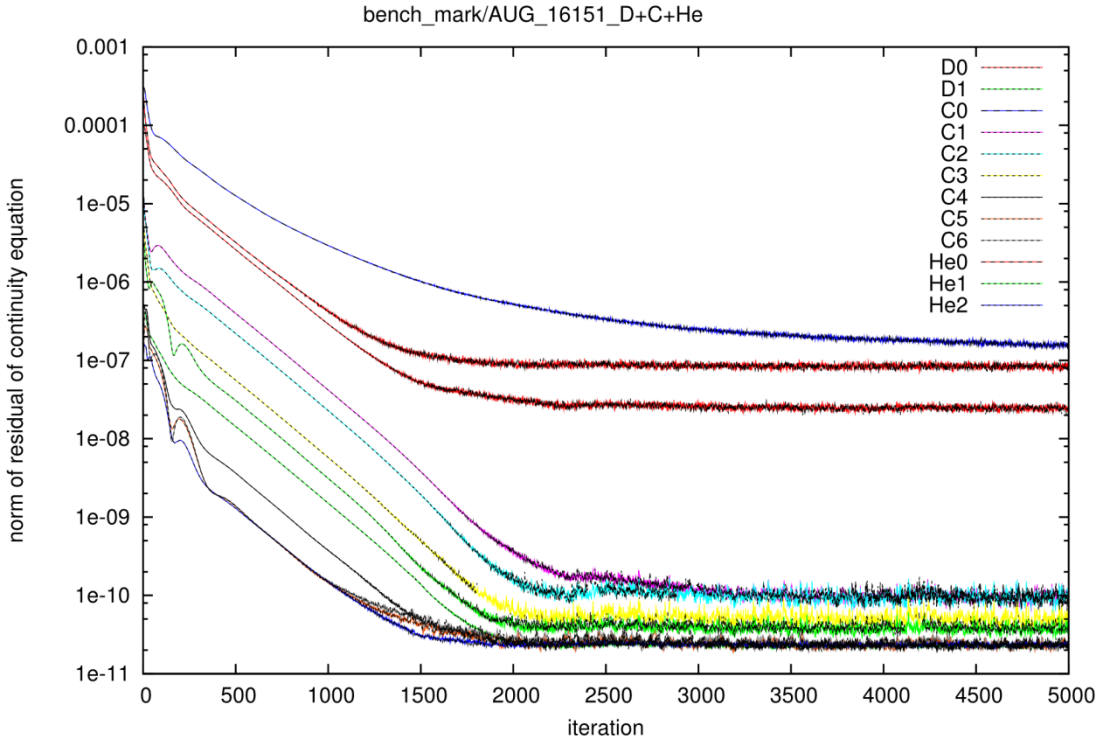

Fig. 7 Residuals of the continuity equation. The results from the trunk version of the code are shown in color; the results from the HLST branch are displayed with black dashed curves on top of the colored curves.

The residuals of the continuity equations are shown in Fig. 7, where the results from the trunk are displayed with colored curves, and the results from the HLST branch are shown with dashed lines. The HLST version of the code was executed on 20 threads. The results agree well. The final values of the variables describing the plasma state (*na, ne, te, ti, po*) agree up to machine precision. The variable *ua* has larger difference, but this is understandable because *ua* changes sign over the simulation domain and the element by element comparison of the *ua* arrays lead to large relative difference where the value is close to zero. Two other variables (*smfr* and *b2stbc_smo*) show also large discrepancy, but they are dependent on *ua*, which explains the differences.

2.5. ***B2 reproducibility***

2.5.1. **Reproducibility and OpenMP reductions**

Some of the OpenMP regions include reductions, an example is shown in Fig. 8. In this example, contributions to the final value of the *ue* array are calculated by different threads, and the results are summed up at the end. The OpenMP standard

does not define in which order the results from different threads will be added together, so round-off errors can appear. Therefore, there is no guarantee that the results will be bit identical.

In fact, the results are not bit identical compared to the results of the sequential code, even if only one OpenMP thread is used. The maximum relative difference that is introduced by the reduction clause in Fig. 8 is around 10^{-14} in the array *ue.*

```
!$OMP PARALLEL DO DEFAULT(none) SHARED(rza,ua,na,ns)
REDUCTION(+:ue)
        do is=0,ns-1
          ue(:,:)=ue(:,:)+rza(:,:,is)*ua(:,:,is)*na(:,:,is)
        enddo
!$OMP END PARALLEL DO
```

Fig. 8 OpenMP parallel region with reduction over the *ue* array.

2.5.2. **Differences due to reductions**

The error caused by the reduction spreads during the calculation, and certain variables in the output files can accumulate large errors. We investigated how large the difference was by comparing the results from a sequential simulation with an OpenMP simulation using only one thread. For these tests, a version of the B2 code was used which contained only a single reduction operation: the same loop that is illustrated in Fig. 8. All other parallel regions with reductions were removed.

For the test case AUG_16151_D, the values in the output files are bit identical. For tests with higher particle number (AUG_16151_D+C+He, ITER 98 species), the output files can contain large differences. Table 3 lists the relative and average differences for some of the output variables. Most of the variables are two or three dimensional arrays (2D spatial grid + particle species). For the ITER case, the maximal error is very high for the particle flux (*fna*), but the average error is 10^{-7}, which is considered as tolerable. The values describing the final plasma state *(te, ti, na, po*) have an average relative error of 10^{-14}, except for the parallel velocity *ua,* which is more sensitive to the round off errors from the reduction and has a relative error of 10^{-7}. For several variables, there is no error or the error is close to machine precision.

To ensure that the differences are indeed caused by round-off errors and not by any other mistake, two additional tests were performed. First, to emulate the round off errors a small random noise was added to the *ue* variable in the sequential code. This resulted in similar differences as described in Table 3. Second, if the REDUCTION clauses were replaced by SHARED and one thread was used for the OpenMP code, then identical results (compared to the sequential code) could be achieved.

Filename	Variable	AUG 16151 D+C+He		ITER 98 species	
		Max diff	Avg diff	Max diff	Avg diff
B2FSTATE	kinrgy	0	0	5.2e-2	6.5e-7
	ua	3.3e-8	2.3e-9	4.0e-2	4.6e-7
	fna	0	0	1.5e-2	1.9e-7
	fhe	0	0	7.8e-5	1.4e-7
	na	0	0	1.6e-13	2.9e-14
	te	0	0	9.5e-14	6.5e-14
B2FPLASMA	fna_52	0	0	4.2	4.6e-4
	rrahi	0	0	4.9e-1	2.8e-6
	smq	0	0	1.1e-1	8.1e-7
B2FMOVIE	delua	1.2e-4	7.3e-7	4.0e-2	4.6e-7
	delna	0	0	1.1e-9	5e9e-13
	delpo	3.1e-6	1.3e-6	1.4e-12	1.3e-13
	delte	8.3e-6	8.2e-7	1.3e-12	3.0e-13
B2FTRACE	data	1.8e-8	1.3e-8	1.15e-9	1.9e-10

Table 3 Relative differences in the output files. The sequential code (compiled without the `-openmp` flag) was compared with the OpenMP parallel code (using one thread). The table shows the maximum relative difference and the average relative difference for two different test cases: AUG 16151 D+C+He and ITER. The maximal relative difference is large for certain array elements (denoted by red) but the average difference over the whole array is moderate even for these arrays. The difference in basic physical quantities (denoted by green) is close to machine precision.

2.5.3. **Deterministic reduction**

Intel's OpenMP implementation has the possibility to set an environment variable `KMP_DETERMINISTIC_REDUCTION=true`, which according to the documentation "has the effect that [...] an OpenMP reduction clause has a consistent floating point result from run to run, since round-off errors are identical." It promises to give the same result if the same number of threads is used. In practice this flag does not work for the array reductions used in B2. This problem was reported to Intel, and we are waiting for their feedback. Until this problem is resolved it is not possible to get deterministic results with the OpenMP version of the code.

2.5.4. **B2 Differences due to optimization**

Changes in the optimization flags or in the floating point model flag instruct the compiler to use different optimizations, and can lead to differences in the output that are similar to those in Table 3. Round off errors in certain variables can cause surprisingly large differences in some of the output values. For example, even if we change a single array element in the *B2XPFE* subroutine:

fne(42,16,0) = fne(42,16,0) * (1.0D0 + 1.0D-14),

the maximal error for the AUG_16151_D test case becomes 10^{-2} for the *fmo* variable (the other two test cases had a maximal error of 10^{-9}).

It was necessary to change the order of certain arithmetic operations during the optimization process. Therefore, the results from the optimized code are different compared to the results from the original version of B2. During testing it was ensured that the final physical parameters have small errors. The error in *te, ti, ne, na and po* was always smaller than 10^{-13} and the error in the variable *ua* was around 10^{-7}.

2.6. ***EIRENE***

After the work on the B2 code, the next step in the SOLPS optimization project is to ensure that the coupled B2-EIRENE system is working together in parallel. The SOLPSOPT project started with the assumption that the MPI parallelized EIRENE code was working properly and scaling reasonably well. Unfortunately, the MPI version of the EIRENE code in SOLPS 5.0 was not maintained, and the changes

introduced during the previous years (dating back to 2011) were only tested in the sequential version of EIRENE. As a result, the MPI version of EIRENE did not even compile.

The EIRENE code exists in several versions hosted by different institutes. The EIRENE code in SOLPS 5.0 is an older version of EIRENE. The recently released SOLPS-ITER code includes the latest version of EIRENE. After discussion with the project coordinator, it was decided that we should switch to the latest version, instead of fixing the version included in SOLPS 5.0. We evaluated the parallel performance of the stand-alone version of EIRENE from the ITER repository, and found that it does not scale for the test cases that we have. In the following sections we will discuss this problem, devise a scalable parallelization strategy, and test it.

2.6.1. **Code version and test cases**

We use a version of EIRENE from the ITER repository (feature/HLST branch from *ssh://git.ITER.org/bnd/Eirene*). We use two stand-alone (EIRENE without B2) test cases**:** an ASDEX Upgrade (AUG) test case that has 630k particles, 5 bulk ion species and 59 reactions, and an ITER test case with 73k particles and 25 bulk ion species and 26 reactions.

2.6.2. **MPI bugfixes**

The feature/HLST branch is based on the main ITER-develop branch of the EIRENE code. A few problems were fixed by the project coordinator and L. Hüdepohl, such that the MPI version of the code could work for the AUG test case, at least with certain MPI libraries. Unfortunately, the MPI version of the code was still very fragile, and several other problems had to be corrected before the MPI scaling could be analyzed. In the following we list the problems that were encountered.

A problem was discovered with *MPI_GATHERV* calls in the *COLLECT_CENSUS* subroutine. Two of these calls had identical source and destination buffers, which is an error that was caught by the runtime checks when the code is executed on the Hydra computer using the IBM MPI library. The problem was fixed using the `MPI_IN_PLACE` option that has to be specified on the root process.

The *EIRENE_BROADCAST* subroutine is responsible of distributing the input data among the MPI processes. Some of the input data is stored in allocatable arrays. *Process 0* allocates its own arrays, fills it with input data, and broadcasts them. The rest of the MPI tasks check the allocation status of these arrays, and if they are not allocated, or the allocated size is not in agreement with the input data, then reallocate them before receiving the data from *process 0*. The arrays are handled using pointers, and the *ASSOCIATED* intrinsic is used to check the pointer association status. This check failed because some of the pointers were not initialized, and it led to segmentation faults. According to the Fortran standard, the pointer association status is not defined initially; it has to be initialized either by the *NULLIFY* statement or by using the *NULL()* intrinsic function in an initialization expression during type declaration. The problem was fixed by initializing the pointer that caused the problem.

Unfortunately, not all the input data is handled centrally by the *EIRENE_BROADCAST* function. The subroutine *EIRENE_INIT_REFL_HLM* opens its own input file and reads data. This file was opened by each MPI process separately, which is not the recommended way of reading input files. In practice, it led to errors when a larger number of MPI processes were used. The problem was fixed by reading the data only on *process 0*, and then distributing it with *MPI_BROADCAST*.

The *MPI_BARRIER* function is seldom needed in a production code, but it can be useful during debugging or during profiling. The EIRENE code has 27 calls to *MPI_BARRIER*. Some of them are for the mentioned timing and debugging purposes, but most of them are unnecessary, because they are directly preceding or following collective MPI routines, which anyway synchronize all the tasks. Inside the

IF3COP subroutine, an *MPI_BARRIER* is used to synchronize the processes. Unfortunately, it is not guaranteed that all processes call *IF3COP*, and therefore the *MPI_BARRIER* can lead to a deadlock, which became apparent using the ITER test case. The deadlock only occurs for specific combinations of strata numbers and processor numbers. The problem was fixed by removing the call to the barrier.

The fix for these problems was sent to the SOLPS-ITER developers, who have incorporated them into the main development line.

The Forcheck static source code analyzer was used to check the source files of EIRENE. A couple of uninitialized variables were found, but these are not used in our test case. For one subroutine there was an inconsistency of actual and dummy argument type. These problems were reported to the EIRENE developers, who have corrected them. The runtime checks from the Intel compiler did not find any issues.

The MUST tool (RWTH Aachen University, 2015) was used to check whether the Message Passing Interface is used correctly. This is a successor of the MARMOT tool, and performs runtime checks on the usage of the MPI library. The test succeeded without any errors, only one warning was received because of a broadcast of a zero sized message.

After the bugs had been fixed, the stand-alone version of EIRENE was working for the two test cases, and the parallel scaling could be analyzed.

2.6.3. **Parallel work distribution in EIRENE**

A simplified outline of EIRENE is presented in Fig. 9. In the AUG test case, 97% of the serial execution time is spent in the particle loop. The particles are grouped into different strata. When the code is executed in parallel, then the loop over the strata and/or the loop over the particles can be executed concurrently. After following the particles, the results are summed up over the MPI processes.

```
1. initialize
2. for each stratum
3.     init stratum
4.     for every particle in stratum
5.         follow particle
6.     sum contributions in stratum
7. sum over strata
```

Fig. 9 Outline of EIRENE algorithm. Loops in the second and fourth lines can be parallel. The steps in the first, sixth, and seventh lines involve MPI communication.

The work is distributed in parallel by filling out a table that describes which MPI process calculates which strata (see Fig. 10 for illustration). A round robin distribution of processors is used if there are more strata than MPI processes. In this case one process can calculate several strata. If there are more MPI processes than strata, then each process calculates only one stratum, but a stratum can be associated to more than one processor. The number of particles inside a stratum is distributed evenly between the MPI processes that calculate that stratum.

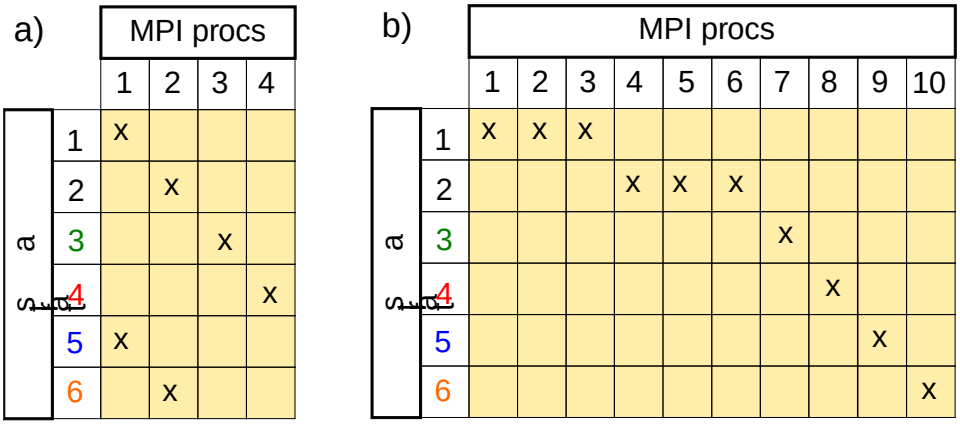

Fig. 10 Work distribution: the MPI processes are assigned to the strata by filling out the *PROCFORSTRA* matrix. (a) Round robin distribution if $N_{procs} < N_{strata}$. (b) Strata with higher work load get additional processors if $N_{procs} > N_{strata}$.

In the AUG test case, we have six strata as shown in Fig. 11. The number of particles per stratum varies strongly as shown with the blue bars. The computational time to follow the particles depends not only on the number of particles, but also on the physical parameters in the stratum. The CPU time to calculate all particle trajectories in different strata are shown with red bars in Fig. 11. If the number of available CPUs is larger than the number of strata, then the strata with larger workload get more CPUs. Using a large number of CPUs this could lead to a balanced work distribution. With a small number of CPUs it is difficult to achieve load balance. The problem of load imbalance during stratified sampling is also discussed in the manual of EIRENE (Reiter D. , 2009): “B2-EIRENE is heavily resorting to ‘stratified sampling’. This means the primary sources of neutral particles are split by their physical or computational origin into ‘sub-strata’, and results are then linearly superimposed finally. This […] leads to load balancing issues”.

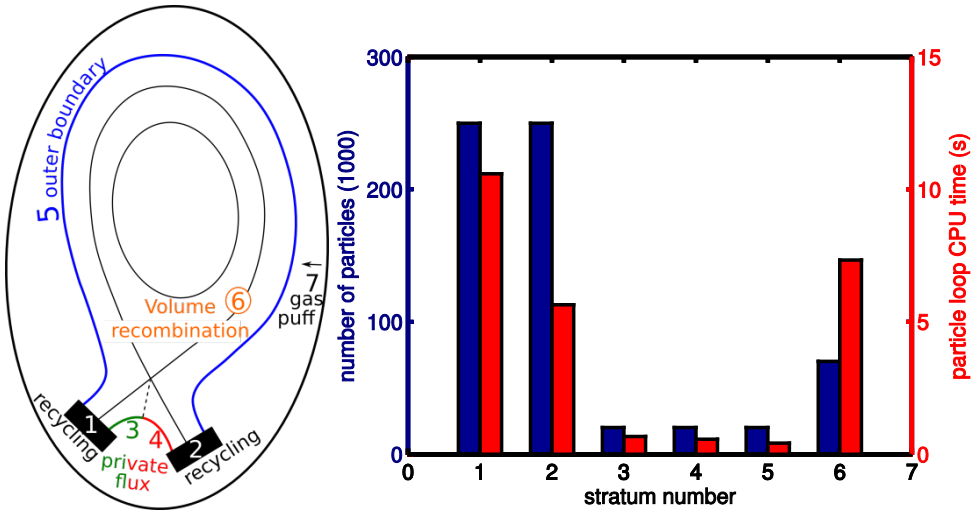

Fig. 11 Stratified sampling for the AUG test case. Left: the sketch of the strata, right: particle number (blue bars) and computational time (red bars) in different strata.

Using the default scheduling (as illustrated in Fig. 10) the execution time of the code was measured. The speedup is shown as a function of MPI processes in the left part of Fig. 12. The curve saturates quickly because of load imbalance. This imbalance is illustrated on the right hand side of the figure for the case of eight MPI processes. The execution time of the different MPI processes are represented with bars. A large part of the execution time is spent as MPI waiting time (illustrated by red color).

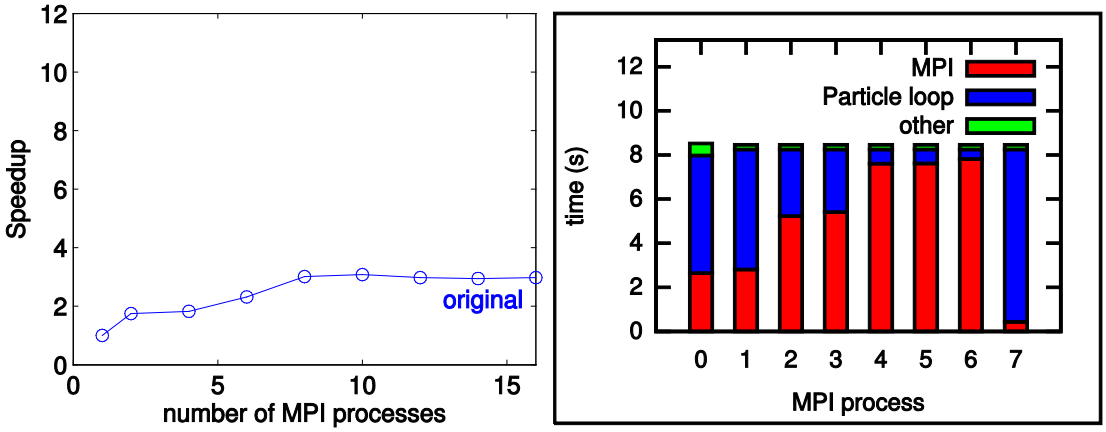

Fig. 12 Left: speedup of the EIRENE code for a fixed size ASDEX Upgrade test case (strong scaling), measured on an Intel Sandy Bridge node. Right: distribution of work among eight MPI processes. The time spent in the particle loop is shown with blue color, red denotes MPI waiting time, and green shows time spent in sequential parts of the code.

2.7. *More balanced workload distribution*

Our aim is to achieve reasonable scaling within a computer node (up to 16 cores). Since the number of strata is comparable to the number of MPI processes, we do not except to achieve good scaling just by assigning processes to strata. Instead we propose to let all MPI processes calculate all strata and distribute the particles evenly between the processes, as illustrated in Fig. 13. Let us call this APCAS (All Process Calculates All Strata) workload distribution.

		MPI procs									
		1	2	3	4	5	6	7	8	9	10
strata	1	x	x	x	x	x	x	x	x	x	x
	2	x	x	x	x	x	x	x	x	x	x
	3	x	x	x	x	x	x	x	x	x	x
	4	x	x	x	x	x	x	x	x	x	x
	5	x	x	x	x	x	x	x	x	x	x
	6	x	x	x	x	x	x	x	x	x	x

Fig. 13 Work distribution table when all MPI processes calculate all strata.

The advantage of this approach is that the load imbalance between different strata would not affect the load balance of different MPI processes. Modifying the work distribution is also relatively easy using the existing framework. The disadvantage is that now only the particle loop is truly parallel, other steps that are done outside the particle loop are now effectively performed sequentially.

The work distribution matrix (*PROCFORSTRA*) was modified to test the APCAS workload distribution. The green curve in Fig. 14 shows the achieved speedup. We compare the speedup to theoretical estimates based on Amdahl's law. Originally 97% of the execution time is spent in the particle loop, so the black curve in Fig. 14 would give the maximum achievable speedup for the test case. But we have an overhead due to MPI communications which decrease the parallel fraction to 94% (mangenta curve). We can see that the measured speedup is close to this theoretical estimate.

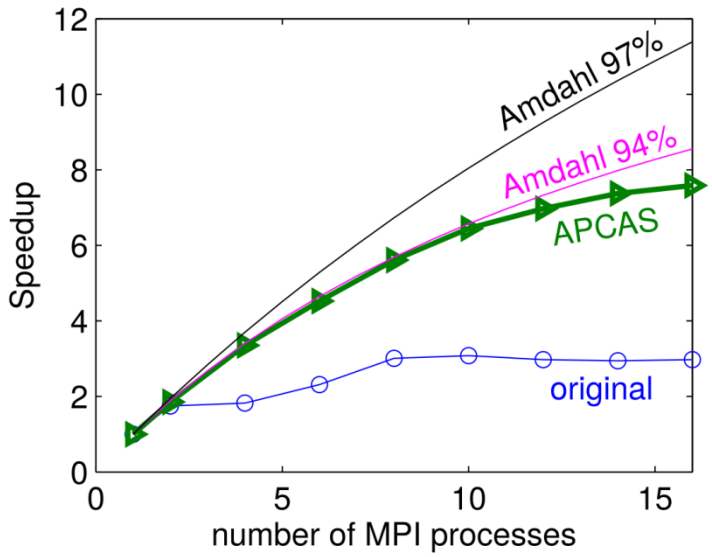

Fig. 14 Speedup of EIRENE as a function of MPI processes. If all processes calculate all strata (APCAS strategy) then the work is distributed evenly and we reach a higher speedup (green curve). The achieved speedup is compared to theoretical estimates of the maximum speedup based on Amdahl's law**.**

2.7.1. **Execution time and particle number**

The applicability of the APCAS workload distribution depends on how large fraction of the execution time is spent in the particle loop. To calculate all strata with all processes is the simplest workload distribution, and it was implemented in the past in EIRENE, but was not favorable for the cases studied at that time (Reiter D. , 2015). Therefore in this section we examine how would the APCAS workload distribution perform for our test cases if we change the particle number.

The execution time of the particle loop was measured as a function of the number of particles. The left side of Fig. 15 shows the fraction of the simulation time spent in the particle loop and in the strata loop for the ASDEX Upgrade test case. The code was executed serially. Our aim is to reach at least a speedup of six using the full node. Amdahl's law suggests, that we would need at least 90% parallelism to achieve that speedup (see Fig. 16). This 90% threshold is shown in Fig. 15 with the dashed horizontal line. The arrow marks the point with 157k particles (left subfigure), where the particle loop crosses the 90% threshold, above this, the APCAS parallelization strategy would work. The green vertical bar highlights the particle number for the original test case that we received from the project coordinator, which is a factor of four higher what is needed for a good APCAS parallelization scaling.

The right side of the figure shows the same test for the ITER case. Here the 90% threshold is reached with around 73k particles, which is also the same particle number that is used in the original test case. The APCAS parallelization strategy would theoretically work for the ITER test case too, but the MPI overhead will cause problems.

The strata loop contains the particle loop, plus some initialization and post processing steps specific for each stratum, therefore, it takes a larger share of the execution time than the particle loop. This is why the default parallelization strategy in EIRENE is to distribute the strata loop in parallel. Unfortunately, this fails for our test cases because of load imbalance.

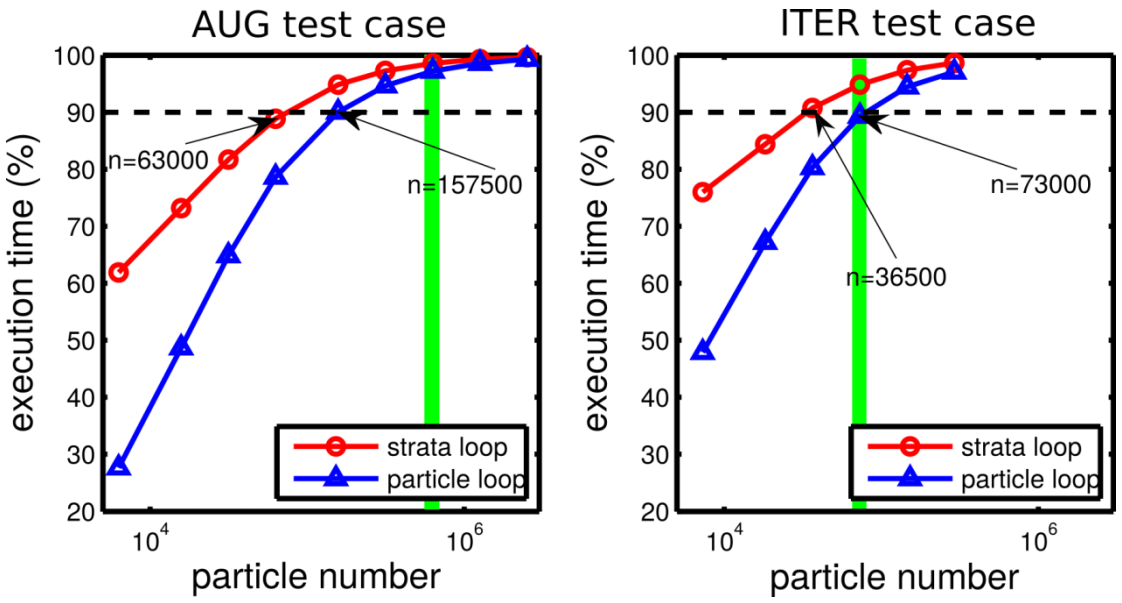

Fig. 15 Time spent in the strata loop (red curve) and in the particle loop (blue curve) as a function of the number of particles for the ASDEX Upgrade test case (left) and the ITER test case (right). The arrows indicate the closest point to the 90% threshold. The green bar shows how many particles are used in the original test case.

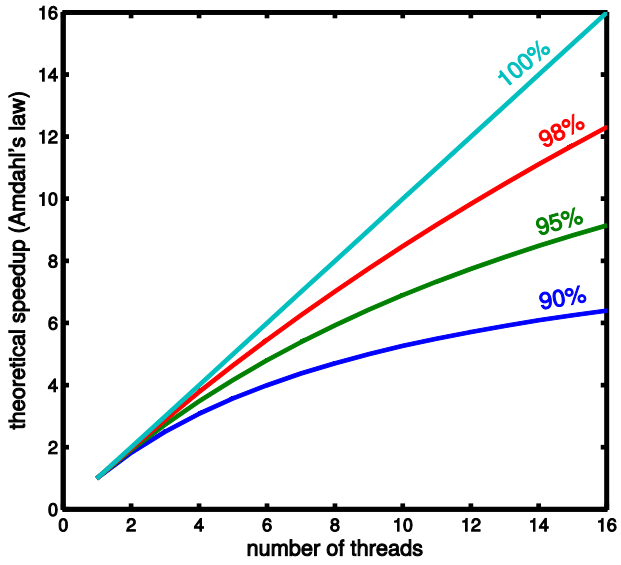

Fig. 16 Amdahl's law: speedup vs. number of parallel processes for different parallel fractions.

2.7.2. **MPI scaling of the ITER test case**

The MPI scaling of the ITER test case was performed by fixing the particle number to 73k. The speedup is shown in Fig. 17. The red curve show the speedup achieved by the proposed APCAS strategy. The particle loop has 89% of the sequential execution time, so the magenta curve shows the theoretical estimate for the speedup. This estimate does not include the overhead of the MPI communication, which decreases the parallel fraction to around 83% (black curve). Considering this overhead, the estimated speedup is very close to the achieved speedup. The MPI overhead should be decreased to improve the speedup.

The strata loop has 95% of the execution time, the green curve in Fig. 17 shows what would be the expected speedup if it would be possible to parallelize the strata loop ideally.

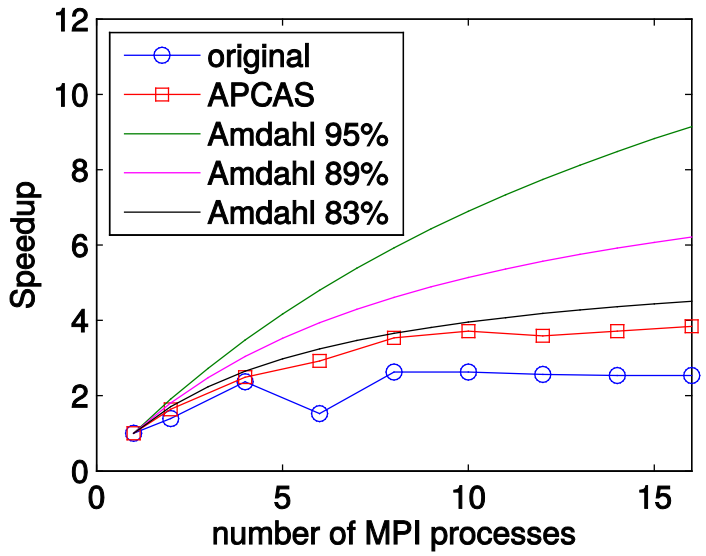

Fig. 17 Speedup of the ITER test case. The original parallelization strategy (blue) is compared to the APCAS strategy (red). Amdahl's law gives theoretical upper limit for the achievable speedup.

2.8. *MPI overhead for the ITER test case*

To improve the speedup for the ITER test case, the MPI overhead should be decreased or masked with computation. MPI communication takes place at three stages of the computation:

1. Initialization (subroutine *EIRENE_BROADCAST*)
2. Sum inside a stratum (subroutine *EIRENE_CALSTR*)
3. Sum over all strata

For the ITER test case (using the proposed APCAS workload distribution), most of the MPI time is spent in *EIRENE_BROADCAST* and *EIRENE_CALSTR*. This time was measured for a simulation using 16 threads. Fig. 18 shows the time spent in the subroutines *EIRENE_BROADCAST* and *EIRENE_CALSTR* on different MPI processes. On average 1.6 seconds are spent in MPI calls out of the 5.5 seconds of total execution time (Fig. 18 left side). Half of this execution time is spent in broadcasting the initial data. This step is supposedly only performed in the beginning, so the cost of this call should be amortized away if we use larger number of steps.

2.8.1. **Summing inside a stratum**

To sum the results inside a stratum takes the other half of the communication time. This communication is implemented in the *EIRENE_CALSTR* subroutine which performs an *MPI_REDUCE* on around 50 variables. There is room to improve the execution time of *EIRENE_CALSTR*.

As a first step, the unnecessary temporary buffers were removed and the *MPI_IN_PLACE* option was used instead. The communication of 2D arrays was implemented in a very inefficient way: the rows of the array (which in Fortran are non contiguous in memory) were copied to 1D buffers individually before calling the MPI subroutines. This had a large overhead. This was replaced with a single MPI call for the whole array. The execution time of the communication decreased significantly, as can be seen if we compare the red bars on the left and right side of Fig. 18.

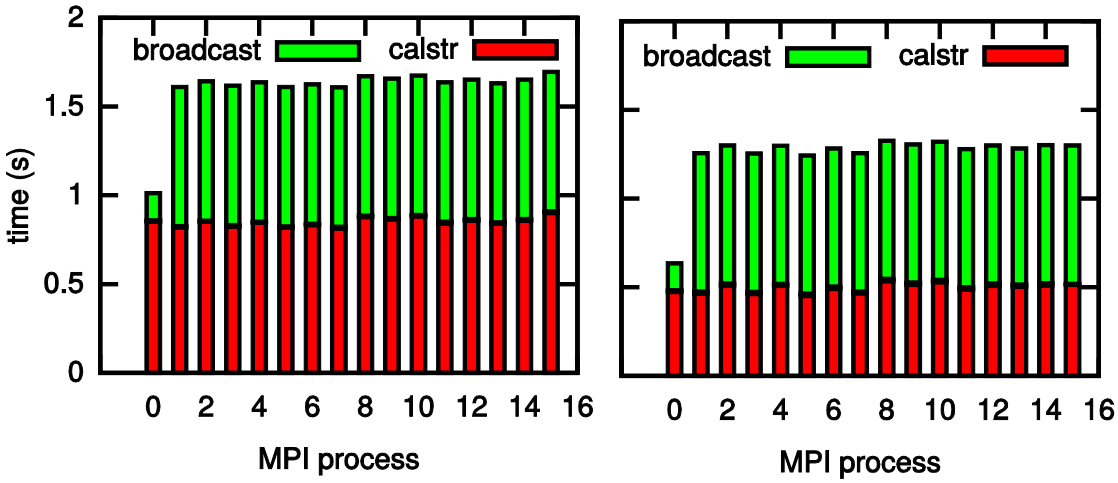

Fig. 18 Time spent in *EIRENE_BROADCAST* and *EIRENE_CALSTR* for a simulation with 16 MPI processes. Left side: original code. Right side: improved *EIRENE_CALSTR*.

There are around 50 calls to *MPI_REDUCE* in the *EIRENE_CALSTR* subroutine, and most of them send only a small buffer. If we consider how the execution time of *MPI_REDUCE* depends on the buffer size (Fig. 19), then we can see that up to 1kB, the execution time is nearly independent of the buffer size. This implies that it would be faster to pack smaller buffers into a single temporary buffer, and do the communication at once using a single buffer. This would take less time, even if we consider the time needed to copy the data into the temporary buffer. Additionally, one could try to use non blocking reduction operations. These options shall be tested in the future.

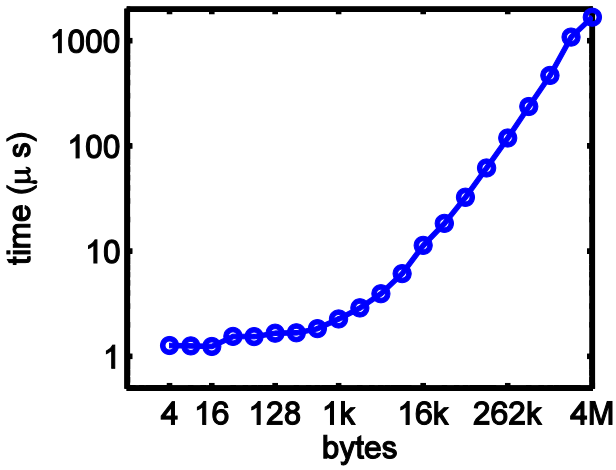

Fig. 19 Execution time of *MPI_REDUCE* as a function of the array size that is reduced (measured with the Intel MPI Benchmark (Intel, 2013), executed on a Sandy Bridge node using 16 MPI processes).

2.9. *Other parallelization strategies*

In the previous sections, we have proposed a simple workload distribution, where all MPI processes calculate all strata (APCAS). This way only the work of the particle loop is distributed among the MPI processes. There is some extra work outside the particle loop that is performed in the strata loop (Fig. 9). The more advanced schemes that are implemented currently in EIRENE (Fig. 10) aim to also distribute this work in parallel. The problem with these workload distribution strategies is that they do not scale due to load imbalance (as we have seen in Fig. 12) unless the input parameters for the strata are set up specifically in a way that would allow balanced execution.

There are two main factors that lead to load imbalance using the originally implemented parallelization strategies:

- The amount of work is different in each stratum (Fig. 11).
- The particles in a stratum are evenly distributed among the MPI processes that calculate that stratum.

The first point could be alleviated if we change the particle number in the strata to have more or less equal work in each stratum. There is already a flag (*NFILEK*) in the code which allows the rescaling of the particle number. The problem with this strategy is that this way we can have either too few particles in certain strata (inaccuracy) or too many particles in other strata (waste of resources).

The second point means that according to the original strategy, if we assign *N* MPI process to a stratum, then each process calculates *1/N* part of the total work. This is because using earlier MPI versions it is complicated to implement otherwise. After all particles are processed in the stratum, the results have to be communicated to one of the processes (the stratum leader). The original implementation has blocking collective communication, which blocks the execution until all the *N* tasks finish the calculation in the stratum. In such a setting it was only effective to evenly distribute the work among the MPI tasks that share the stratum.

The non-blocking collective communication routines that were introduced with the MPI 3.0 standard allow an easy way to overcome this restriction. Since the MPI tasks do not have to wait for each other after finishing a stratum, we can let them calculate different number of particles. This freedom can be used to assign work to each MPI process in a way that has minimal communication.

To distribute the work properly, we need to know how much computational time is needed for each stratum. It is difficult to estimate the workload of a stratum in advance, because it depends on the physical parameters in the stratum. There are two ways to solve this problem.

During a SOLPS simulation we calculate several time steps, and the workload of each stratum should stay similar to the previous time steps. Therefore, one could use the APCAS method for the first time step, measure the workload of each stratum, and then we can use this information in the next time step to optimize the workload distribution.

Alternatively, we could assign the particles dynamically in smaller chunks to the MPI processes, to avoid such initial measurement. This could be achieved with a simple client-server model.

2.9.1. **Client-server workload distribution**

There were some bugs in the implementation of the APCAS scheme, and while these were corrected, the feasibility of a client-server workload distribution was tested. The basic idea is illustrated in Fig. 20. The total work is divided into smaller work packages. The working processes request a work package from the server. A work package describes which stratum to calculate, how many particles to simulate, and whether there are any pre- or postprocessing steps to be done. After the work is finished, a new request is sent to the server, to get additional work. The server can distribute the work packages in a way to minimize collective communication and this way load balance can be achieved.

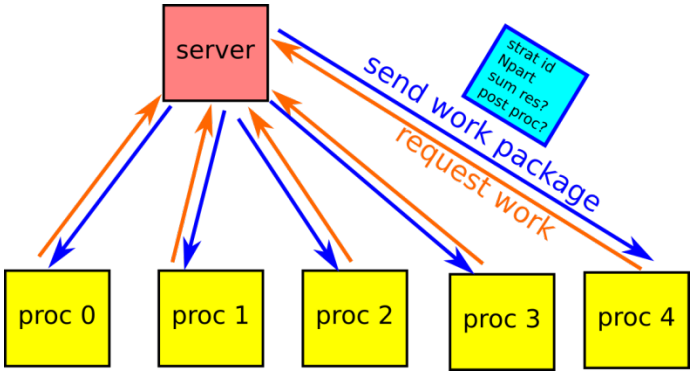

Fig. 20 Client-server workload distribution. The worker processes request work from the server, which sends a package describing the work to be done.

The implementation of this client-server workload distribution is actually not so complicated; a proof-of-principle version is already implemented for EIRENE. This does not give yet correct physical result, but the scaling properties could be analyzed. Fig. 21 shows that this method gives superior scaling to the APCAS method; this is because the amount of collective communication is decreased.

The additional speedup is encouraging, but the client-server model has one major drawback: the simple loop structure of EIRENE (shown in Fig. 11) is not visible anymore, it is replaced by a request to the server. Using only one MPI task, the server emulates the same loop as the original code executes, but in parallel mode the execution order is not trivial. This makes code maintenance more difficult. Because of this reason, it was decided not to pursue this line of optimization further. Instead, we will focus on the other optimization strategy: we will measure total CPU time used for each stratum, and optimize the workload distribution for the next time step.

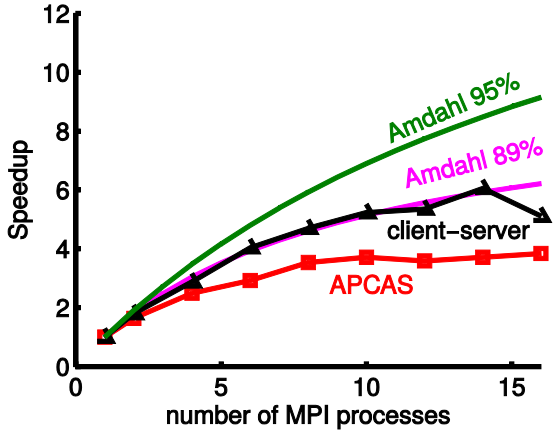

Fig. 21 Scaling of the client-server workload distribution for the ITER test case

2.10. *Correctness of the results*

If we have only one stratum, then the original MPI implementation should deliver exactly the same results as the APCAS scheme. Therefore, a simplified test case was prepared that is based on the AUG test case, but includes only one stratum and lower a particle number. The results from the original code and the modified code agree with each other as expected. Unfortunately, the results of the parallel version of EIRENE do not agree with the serial results, regardless of which parallelization strategy is used.

To compare the results of two simulations with different number of MPI tasks is a non trivial task. The result of the Monte-Carlo simulation depends on the random numbers that are used. If we initialize the random number generator differently, then result for the total pumped flux changes 2% for the AUG test case. Such change

could mask possible MPI related errors, therefore if we want to test the correctness of a parallel simulation, we have to ensure that the same random number sequence is used both in sequential and parallel calculations.

To guarantee reproducible results, we initialize the random number generator for each particle separately, based on a global particle index. This way the results should agree even if the calculation is distributed over many processors. We expect errors related to reduction operations, but this should introduce a difference that is close to machine precision. Contrary to this expectation, we see large errors in the pumped flux as shown in Table 4, and therefore we can conclude that the MPI implementation of EIRENE is not completely correct. Other users of EIRENE have also reported that they see wrong results with the parallel version of the code.

1 MPI task	2 MPI tasks
SURFACES, AT WHICH TEST PARTICLES FLUXES ARE REDUCED NO. SPECIES PUMPED FLUX 144 D 1.2440E-01 144 D2 4.4201E+01 -1 D 7.3568E+00 -7 D2 6.5334E+00 PUMPED FLUX (ATOMIC) PER SPECIES D 7.4812E+00 D2 1.0147E+02 TOTAL PUMPED FLUX (ATOMIC) PUMPTOT= 1.0895E+02	SURFACES, AT WHICH TEST PARTICLES FLUXES ARE REDUCED NO. SPECIES PUMPED FLUX 144 D 1.4643E-01 144 D2 4.4350E+01 -1 D 8.1293E+00 -7 D2 6.1371E+00 PUMPED FLUX (ATOMIC) PER SPECIES D 8.2757E+00 D2 1.0097E+02 TOTAL PUMPED FLUX (ATOMIC) PUMPTOT= 1.0925E+02

Table 4 Output of EIRENE in serial and parallel mode, the differences are highlighted in red. The pumped flux is a very important output parameter from EIRENE, the differences indicate an error in the parallel code.

2.11. *Plans for correctness checks*

To find potential errors in the parallel code is not trivial, because the code is parallelized on a global level. The problem is illustrated in Fig. 22. The blue boxes represent the subroutines where most of the execution time is spent. The yellow ellipses show different modules where the input parameters and the results of the calculation are stored as module variables. The total number of variables is denoted by the number below the subroutine name. A red arrow connecting a blue and a yellow node means that the subroutine uses variables from the module. In effect we have several hundred global variables that are used during the calculation. In parallel mode this data has to be shared properly among the MPI tasks. The reason for the slightly incorrect parallel results is most probably that some variables are not communicated between the tasks.

To find the errors, we will use the following strategy. Subroutines will be generated that save all the global variables from memory into a file. Then we can take snapshots of the global program status at different times during execution, and pinpoint which variables differ between the parallel and serial code version.

The unit test generator that was created for B2 is the starting point for this work. The parser needs to be updated to handle derived types and private variables, and the code generator will be modified to generate read and write subroutines for each module. A proof of principle framework is already in place and works for simpler cases. Fig. 22 was created using data from the new test program generator. In 2016 the testing framework will be finished and used to find the MPI errors in EIRENE.

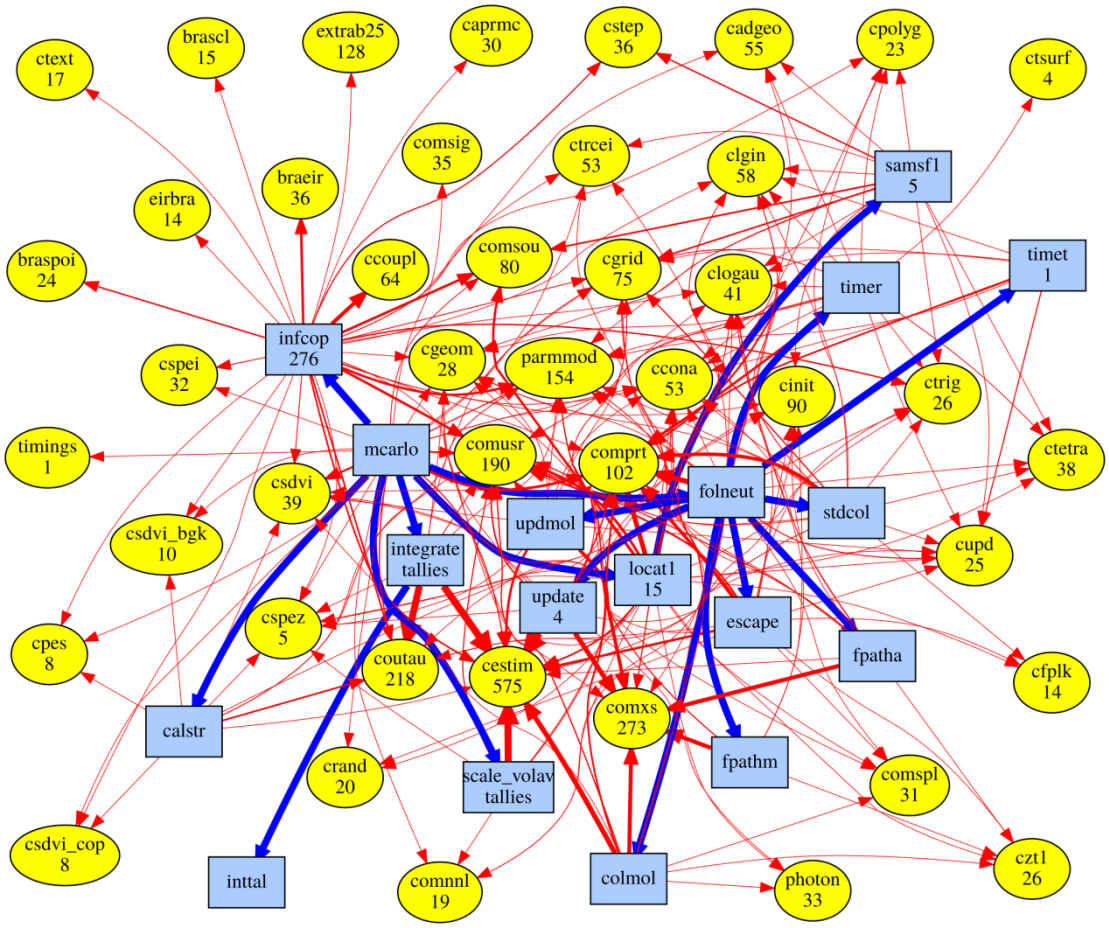

Fig. 22 Dependency on module variables. The blue boxes represent subroutines and the blue arrows between them denote subroutine calls. The yellow ellipses represent modules, and the number below the name is the number of variables defined in that module. A red arrow between a subroutine and a module denotes that the subroutine uses variables from the module, the width of the arrow is proportional to the number of variables used. Some subroutines have variables with *SAVE* attribute, the number of such variables are denoted below the subroutine name in the blue boxes.

2.12. ***Summary***

The PARSOLPS project and the first half of the SOLPSOPT project focused on improving the OpenMP parallelization of the B2 code. The approach was to incrementally increase the fraction of parallel regions in the code by parallelizing individual subroutines. The difficulty is that there are many subroutines which take a small share of the CPU time.

More than 25 subroutines have been parallelized, and 90% of parallelism was reached in the whole code. With these changes a factor of six speedup could be achieved for the ITER test case when executed on a single compute node. The execution time scales well for most of the subroutines as we increase the number of threads.

The OpenMP regions were introduced in a way to avoid large overheads. It was identified that a large part of the code is memory bandwidth limited. Several of the subroutines were optimized to reach a speedup which is close to the bandwidth limit. These optimizations lead to a speedup of the sequential version of the code too, it is now 20% faster than originally.

The individual subroutines have been tested separately with unit tests to ensure the correctness of the modified B2 code. Using the benchmarks from the SVN repository, the whole code was tested for both shorter and longer simulations, and the results agree with the original code.

The work continued by investigating the MPI version of the EIRENE code. Unfortunately, the version in the SOLPS 5.0 code package was not functioning, so it was decided to switch to the latest version of EIRENE from SOLPS-ITER. The parallel performance of the stand-alone version of EIRENE (from the SOLPS-ITER repository) was investigated for an ASDEX Upgrade and for an ITER test case. We have found that the currently implemented parallelization strategies do not scale for these test cases due to load imbalance. A simple and balanced parallelization strategy was implemented instead. Tests with different particle numbers show that it provides reasonable speedup for the two test cases. Possibilities for improving the parallelization were also considered, and will be implemented next year.

During the tests it was found that the parallel version of EIRENE gives incorrect results, even if we use the original code version. A testing framework is under construction to find the errors. This framework is based on the unit test framework created for the B2 code. The project will continue next year for three more months to fix this problem.

2.13. ***Bibliography***

Fehér, T. (2015). *Final report on HLST project PARSOLPS.*

Intel. (2013). *MPI Benchmark.* Retrieved 2015, from https://software.intel.com/en-us/articles/intel-mpi-benchmarks/

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

Reiter, D. (2009). *The EIRENE Code User Manual.*

Reiter, D. (2015). Private cammunication.

RWTH Aachen University. (2015). *MUST MPI Runtime Error Detection Tool*. Retrieved from https://doc.itc.rwth-aachen.de/display/CCP/Project+MUST

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

3.1. ***Introduction***

COCHLEA (*COmputational CHain-like LEApfrog code*) is a numerical code which calculates the magnetic and electric fields in a complex waveguide of a cylindrical symmetry with an arbitrary excitation. It has been written by the principal investigator (PI) Dimitris Peponis, currently a PhD student at the National and Kapodistrian University of Athens (UoA), under the supervision of Prof. Ioannis Tigelis. COCHLEA has been written in the procedural subset of C++11, as it is appropriate for such purposes. The underlying main numerical algorithm is the *Finite-Difference Time Domain (*FDTD) method, known to be apt to parallelism. The HLST had already experience working with FDTD-based codes; see [R14].

The purpose of this project has been to make a parallelization feasibility study and tutoring, in strict collaboration with the PI. Given the limited time assigned to the project (3 ppm in the last quarter of 2015) and the code size (≈ 20K lines of code), we have chosen to visit the PI at his institution for a week.

Section 3.2 describes our changes to the code; then in Sec. 3.3 we describe our activity during the visit. We continue in Sec. 3.4 by analyzing the current scalability and performance properties, then close the report by outlining ideas for further optimization and commenting on how a sequel project could build upon this one.

3.2. ***Course of the Project***

First steps have been guiding the PI in developing a simple metric for sampling performance without having to run the entire code, and skipping file I/O. A common access revision control code repository residing on a server maintained by the PI has been used to commit and track code changes.

We have identified the following major code changes as necessary:

- Exchange loop order for efficient arrays access
- Avoid in-loop conditional branches by e.g. breaking and rearranging them
- Change globally some of the notation and declarations allowing a replacement of the original arrays-of-arrays data structure with another one e.g. avoiding indirection
- Provide a ‘multidimensional array’-like class as an alternative
- Introduce scoped variables and thus easily OpenMP in the main (3D) FDTD loops

The amount of code impacted by such changes was relevant, and their acceptance could have been problematic. To avoid this, we have worked on a separate branch of the code, and focused our loop restructuring efforts on one representative function only. Having observed a marked performance improvement, we have set up a plan in order to deploy these changes together with the PI on the scale of the entire code. During this ‘separate’ investigation plus planning, the PI had the chance to become accustomed with certain features of C++ which we identified as being useful for the proposed solution.

Our custom 2D, 3D ‘multidimensional array’ data structures is a series of template classes which can be accessed with the syntax of arrays-of-arrays, but use contiguous memory instead, and allow the compiler to optimize for direct addressing. The old syntax could be preserved, but nevertheless we have agreed to use a special notation which allows easier switching between the different addressing operators.

These interventions are reversible and fully respect the spirit of working non-invasively. The next section will comment on the deployment of these changes with the PI; the section after on the performance gain.

3.3. *Visit to the Principal Investigator at the UoA*

To make best use of the short time of this project, it has been chosen to include a one-week 'consolidation' visit to the project principal investigator (PI) just before the end of the project time, in a similar fashion to previous projects.

The session began with a detailed explanation of the changes we proposed, with the help of a short presentation. Examples of running the HLST tentative version with partial OpenMP parallelism were given, and the consequences of the main changes were discussed. A fork of COCHLEA has been then initiated by using the latest trunk version as a basis, with the intention of applying changes to the entire code gradually.

Such an accelerated approach has been possible because:

- We could check each change for results divergence and detect problems early
- The *changesets* necessary could be obtained from the HLST SVN repository 'tentative' branch. We could use a shortcut approach by reimplementing only the strictly necessary ones, e.g. avoiding unnecessary preprocessor-based conditionals.
- Only one extra (header) file with HLST specific code had to be introduced, so to be well separated from the original COCHLEA code.

Thanks to this approach, the underlying data structure in COCHLEA could be modified transparently of the rest of the code, which syntactically remains the same.

During this *assisted* deployment work, the most delicate changes, like introducing our 'array type agnostic' declarations have been performed by HLST. After initial loop interventions performed together, in the last days of our visit the PI could parallelize further routines by himself. Except for the FDTD loops, the entire code flow has been left intact.

We consider the above described *first planning and experimentation, then peer programming* work style to foster acceptance of the code, and therefore a good practice.

3.4. *Performance and scalability of COCHLEA + OpenMP*

To appropriately quantify the performance improvement introduced by our changes we need to introduce a metric that is applicable to cases of different sizes and for different thread counts. Let such metric be the number of grid cells processed by COCHLEA in a second of runtime (cps). The amount of arithmetic operations per single cell grid is roughly uniform across cases, therefore the cps rate should ideally stay uniform across cases.

Experimentally we observe that the original code performance (Fig. 23, in red) can degrade by ≈ 75% from the smallest (a few MB in total) to the largest case (≈ 45 GB). Reason for this is the sequence of memory accesses in the FDTD loops: rearranging them for locality (Fig. 23, in blue) corrects most of that performance loss. Further switching our custom array classes with direct addressing on (Fig. 23, in magenta) improves performance additionally by a few percent points.

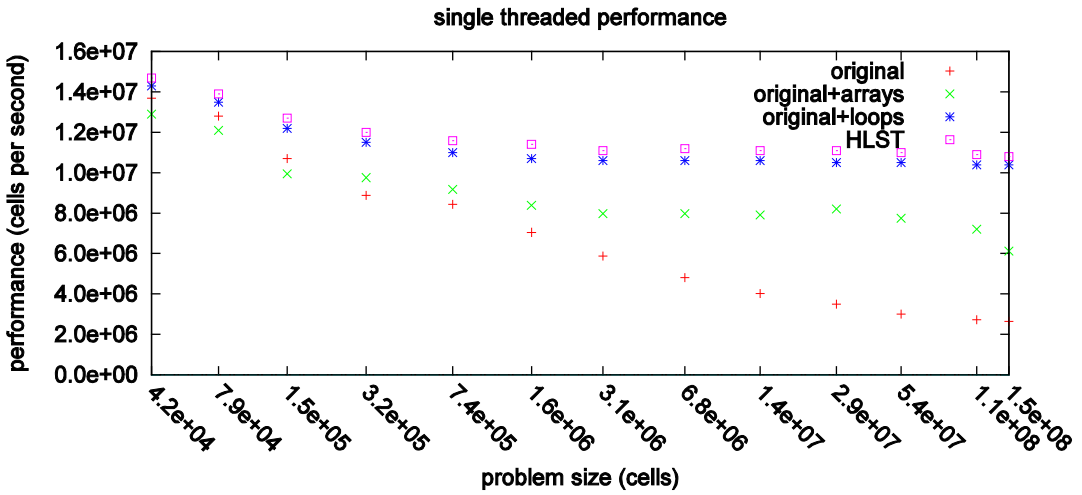

Fig. 23 Serial performance from tiny cases to largest ones for: the original version (in red); the HLST version (in magenta); original with the arrays data structure (in green); original with the restructured loops.

It is interesting to note that: 1) introducing the custom array classes alone (green) yields a lesser improvement than restructuring loops only; 2) the combined improvement (magenta) is weaker than the product of the two; 3) At the 3.1e+06 cells the HLST version speedup over the original trespasses ≈ 2x and for larger cases continues to increase until ≈ 4x.

We are also interested in the OpenMP scaling property of our HLST version with 2, 4, 8, 16 threads on a HELIOS-like node (see Fig. 24). We observe that 16 threads cater a speedup of ≈ 10x, mostly independently of the case size.

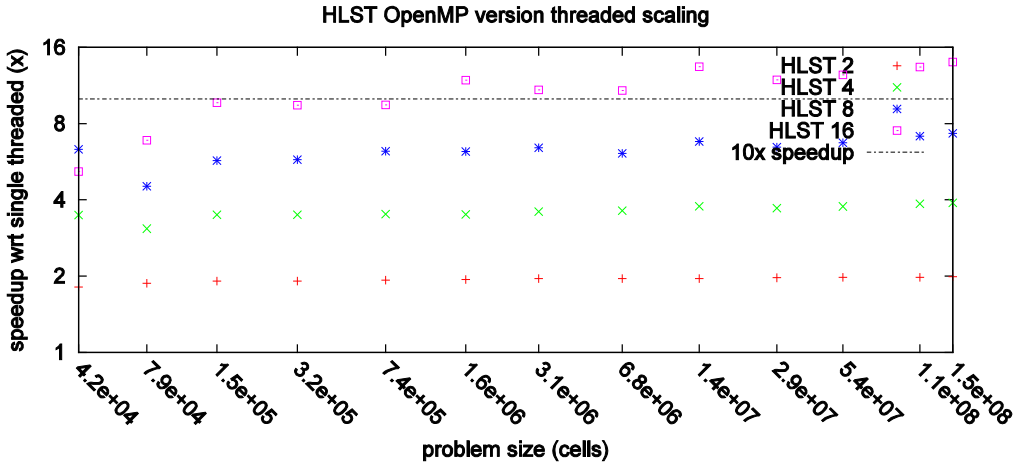

Fig. 24 Speedup of the HLST version of COCHLEA for 16 OpenMP threads (Intel Sandy Bridge).

Combining the 2–4x and ≈ 10x speedups gives 20x to 40x that can now be used to run existing cases faster. For instance, after HLST intervention, COCHLEA is expected to run a simulation of the last part of the ITER gyrotron beam tunnel (14 mm x 12 mm) using a 159 x 1000 x 136 grid and completing in ≈ 6 days on HELIOS. Previously it would have required a few months. Our OpenMP parallelization is currently based on the outermost dimension (*nr*), on which 2D (*nf x nz*) slices are the unit of serial work. The cases considered here have *nr* stretching from 8 to 143: these are the current maximal limits to thread parallelism. A parallelization building on the current one will have to consider the inner loops as well, e.g. by using the `collapse` OpenMP clause.

3.5. *Conclusions and outlook*

The serial speedup from our changes on relevant cases is ≈ 2–4x, and combined with a scalability of ≈ 10x on a HELIOS node with 16 threads, this makes for a ≈ 20–40x total speedup.

Considering the short project time we had to adapt our working style to achieve a full OpenMP parallelization: first planning the key changes and pilot testing, then a full deployment together with the PI.

Our main changes to the code have been FDTD loops restructuring and the introduction of 'array classes' globally. Further easy optimizations as grouping of common sub-expressions, avoidance of divisions have been identified and left to the PI. Further optimizations and special cases (e.g. smooth cylindrical waveguide) can be implemented by using e.g. template programming techniques. Also a 'double buffering' technique could be applied to save memory copy operations for a few arrays. The aforementioned 'custom data type' approach can be developed further in several directions.

When considering an MPI parallelization, one could think of encapsulating handling of *ghost cells* (representation, indexing and exchange) by adapting these classes suitably. Future PI's development plans for COCHLEA include a more complex geometry, 4D and 5D arrays, both of which might benefit from further specialized data structures.

3.6. *References*

[R14] *Final Report on HLST Project REFMUL2P*, part of the HLST core team report 2014

[F14] *Final Report on HLST Project FWTOR*, part of the HLST core team report 2014

4. Final report on HLST project FWTOR-15

4.1. *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. Without distributed memory parallelism FWTOR cannot handle an ITER sized test case. The goal of the present project has been overcoming this obstacle by obtaining a parallel version of FWTOR. This would allow exploiting the much larger memory and the compute power of a machine like HELIOS. As a result, cases that are more physically relevant could be simulated.

This project follows a preliminary study carried out in the last quarter of 2014, which focused on assessing parallelization and optimization possibilities. By retaining the original code structure, a tentative shared memory parallelization by OpenMP and certain optimizations were obtained back then [F14].

First we worked on a plan to obtain a distributed memory version of FWTOR, involving both data distribution and communication. Distributing data among tasks is unavoidable to enable grid sizes not fitting in the memory of a single node and to speed up code execution. As a side effect of such a distributed grid, cells at the subdomains borders have to be replicated (ghost cells) and kept up-to-date by means of inter-node communication (ghost exchange).

In general FWTOR has proven apt to distributed memory parallelization: no algorithmic changes were required. Nevertheless, the meaning of several variables (mainly arrays with their indices and bounds) had to be changed: certain variables still continue to describe the global domain while others pertain to the local domain (and arrays) only. We were interested in a least-invasive intervention to the original code. Most of the current notation and variables (indices identifiers, array bounds, loop labels) were thus reused. The code structure and readability has been retained by avoiding breaking too many loops or introducing too many conditionals. Source code changes due to communication were limited: this shall keep the code maintainable. The shared memory (OpenMP) parallel constructs introduced at the end of 2014 did not conflict with the changes we have implemented with MPI; they were left mostly unaffected.

An integral part of our work consisted in a visit to the project coordinator Christos Tsironis at the National Technical University of Athens, Greece (NTUA) short before the end of the project. At that time, we have dealt with the following:

- Integration / acceptance of the proposed changes to FWTOR.
- Training in minimal maintenance / debugging of the now-parallel code.
- Providing guidelines in implementing new physics without breaking the parallelism.

The success of the project will depend surely on the technical qualities of the MPI enabled FWTOR, but ultimately on the acceptance it will gain with the project coordinator over time. Therefore we provide some details about that visit in Section 4.4.

Section 4.2 describes the bulk of the transition work; section 4.3 comments on experimental results and section 4.5 draws conclusions and recommendations for the future.

4.2. *Transitioning to MPI + OpenMP (hybrid) parallelism*

As a first step, we identified a set of variables, indices and bounds to be impacted. The required communication primitives were also identified and the code was annotated with information on necessary ghost updates and the impacted arrays.

With this information we have continued working along two distinct and independent lines.

One has introduced a number of variables with strictly local meaning. These variables are mainly domain/array dimensions. When running the code in a non-MPI configuration their meaning is global, so they do not differ from their original counterparts. When running in an MPI configuration, their meaning is local, so that data and related computations occur on the local subdomain only.

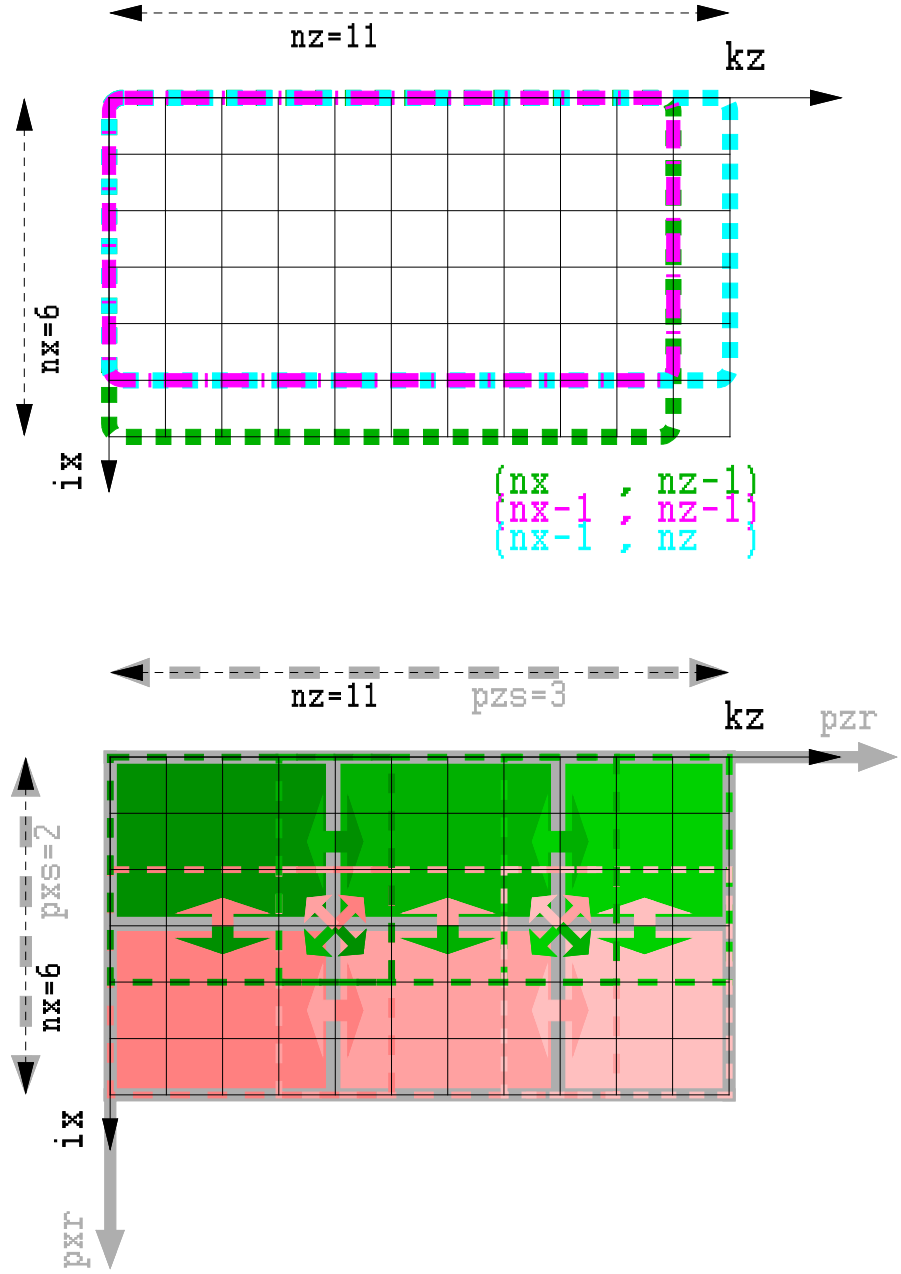

Fig. 25 Top: Delimited by different colors are the bounds of the arrays. Our parallel version employs the same bounds and index space. Bottom: A 6x11 grid distributed on six processes laid as 2x3 (gray delimited, each with a different color). Ghost cells are marked with a process' color being present on other processes. Ghost exchanges are marked with arrows of the sender's color.

Independently, an 'HLST MPI FWTOR' Fortran module (HMF) was being developed in order to host the required communication primitives. These are either *ghost cell exchanges* or *boundary ghost exchanges* on the staggered grid arrays of FWTOR. Fig. 25 exemplifies the staggered arrays, the domain grid, its partitioning, and a ghost exchange primitive. The HMF module can be tested independently of FWTOR by using a stand-alone program to mimic a representative subset of FWTOR's inter-

process communication. This shall keep it testable and ease its replacement or upgrade.

After having introduced the bulk of the new identifiers and having developed a version of HMF, the two had to be integrated, by means of FWTOR calling the HMF subroutines and using its data. The validation tool written by the FWTOR author has been therefore employed to detect numerical discrepancies between results computed with different code/parallelization setups. This has eased the integration phase, as we could detect problems early on. The comparisons became possible only after we adapted the MPI version to the output results in the original format, and in the same format as the original. Our solution consisted in the distributed array slices being gathered on the root node (one row at a time), and then written using ASCII serial output to a single file. By retaining the output file format compatibility, this solution meets our current needs in development, testing, and debugging of small cases. But for the larger runs being now possible, this gather-based solution is too slow to be acceptable. Consequently, a parallel I/O solution will have to be developed in the future. This might be based on ADIOS by means of our HLST-ADIOS-Checkpoint [HAC] module.

In order to keep the boundary conditions code readable while working with the parallel version, extra conditionals have been introduced. Their performance impact is negligible, if compared to communication, as the next section shows.

4.3. *MPI-FWTOR: strong scaling results*

We have performed strong scaling tests of the now hybrid parallel (MPI + OpenMP) version of FWTOR on the HELIOS computer. That is, we use fixed cases and increase MPI parallelism in running them. We have chosen three relevant cases, differing in grid size and number of time steps to termination. Our base case is the largest one fitting on a single HELIOS node, with 7128 x 7128 cells and needing 611k steps to completion. The two other cases (‘half’ and ‘quarter’) use respectively a quarter and sixteenth of the cells and complete in respectively half and a quarter of the time steps.

As we will now comment, the three cases scale differently, due to the interaction of different technological factors (see performance plots in Fig. 26). Originally, FWTOR would complete the large case in more than one year; with OpenMP it would still require over one month; now with MPI and OpenMP a few hours suffice. Looking at the largest parallel run (512 nodes, one MPI process per node), the speedup w.r.t. MPI alone is over 200x; w.r.t, OpenMP also, of over 2500x (with circa 12x being the OpenMP speedup); MPI-wise this is near to 50% parallel efficiency. The quarter case scales with less than 20% of the parallel efficiency, the half case lies in between. Up to 32 nodes, parallelism efficiency remains at over 50% in all cases; afterwards it decreases further, more markedly for the smallest case. Reason for this seems to be the increasing fraction of each time step time being spent in MPI communication.

In an experiment conducted with all communication off, execution speed at 32 nodes was twice as fast as with communication, thus indicating that already there half of the time is spent in communication. This fraction increases as the time step computation duration reduces to the range of milliseconds. In the mentioned no-communication experiment we have even observed a super-linear scaling, likely because of the increasing reuse of data in the cache memory; indeed, at maximum parallelism, each node running the quarter case can host all of the data in the last level cache memory. But in a normal run, the message passing overhead ends up shadowing the super-linear scaling effect.

Inspecting the timings for the boundary condition routine, it is evident that this is only relevant when communication occurs; in other words, the conditional statements we have included to make the code parallel do not pose any significant overhead at this stage.

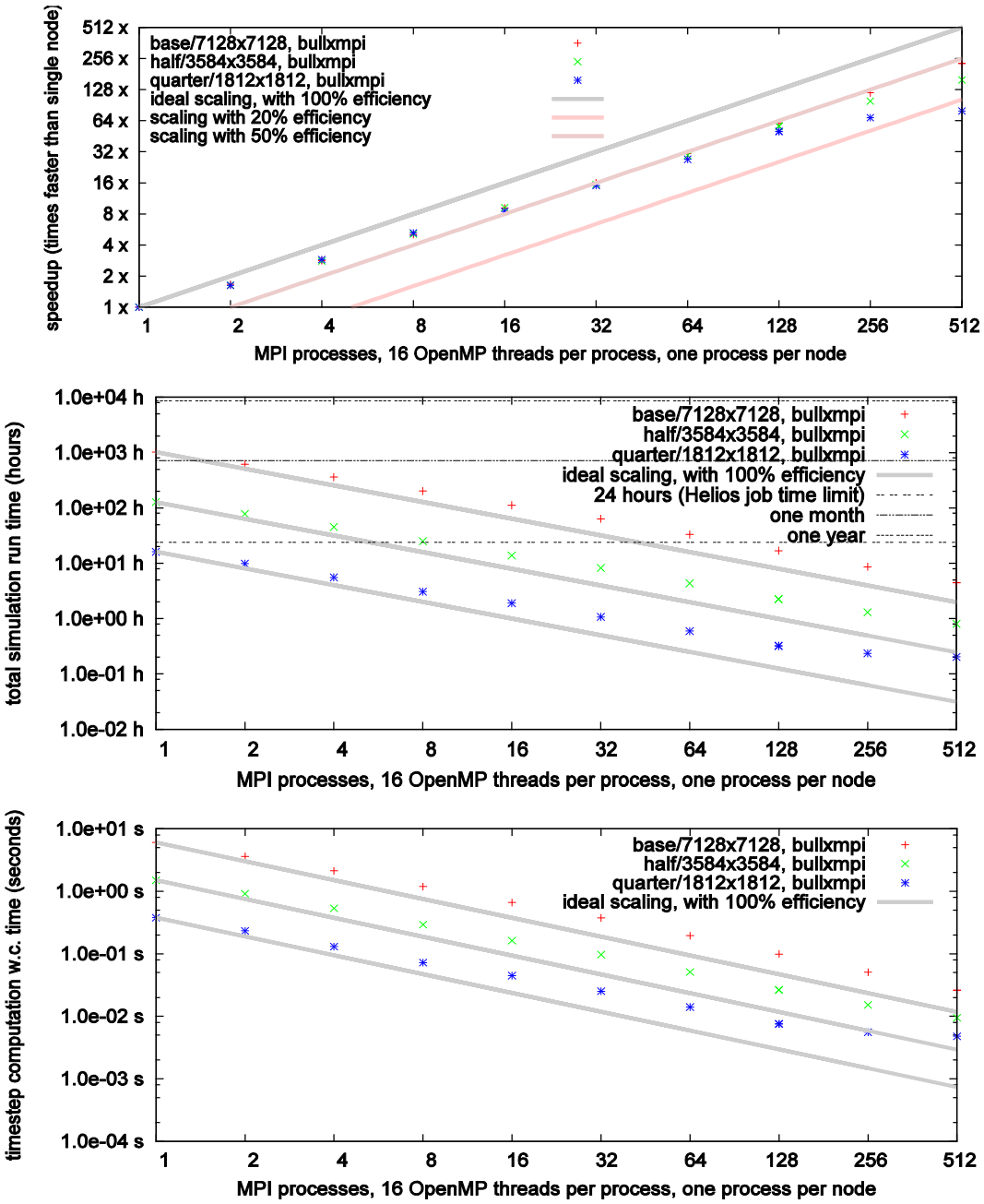

Fig. 26 Top: Speedup: serial computation time per time step divided by parallel computational time; Middle: Projected total runtime of the three suggested cases. Notice how cases requiring over a year of serial computation now complete in a few hours; Bottom: Duration of the computation of a time step. The shorter this is, the larger the fraction of time spent in MPI communication, mostly because of the latency.

4.4. *Visit to the principal investigator*

In order to ease the handover of the now MPI-parallel FWTOR code, it has been decided to include a one-week visit to the project principal investigator (PI) just before the end of the project time, in the first week of September, 2015.

The session began with a detailed explanation of the parallelization concepts to the PI, with support of a beamer backed presentation. The MPI programming details have been kept out of the discussion; emphasis has been put on the new ‘MPI process local’ variables in the code, topical to the PI. After the discussion, a parallel fork of FWTOR has been re-programmed by using the OpenMP-enabled version as a basis, and applying changes gradually. This approach has been possible only because:

- A working MPI version was already present, having been developed at HLST over the course of 2015
- The changes necessary to obtain that version could be obtained from the SVN repository (e.g. changes concerning the choice of 'local' variables, or locations of code where communication was needed)
- We could use a 'shortcut' approach by reimplementing only the strictly necessary changes, and avoiding much of the preprocessor-based conditionals
- Most of the MPI code resides is in a separate HLST-MPI-FWTOR module (HMF) intended to hide MPI details from FWTOR, which has been included 'as it is'

Thanks to this approach, the numerical schemes of FWTOR can be modified to a good degree without having to deal with MPI. However, domain decomposition poses certain limitations; guidelines have been provided in how to cope with them.

During this 'reimplementation' work the fundamental changes have been 'reintroduced' by the PI under our guidance, accompanied by discussion of their implications. This method of work has been chosen to foster acceptance of the code and we regard it as successful.

4.5. *Conclusions: state of MPI-FWTOR and further activities*

FWTOR is now able to run cases previously precluded, primarily due to run time requirements. The entire computational part of FWTOR is now parallel, via both OpenMP and MPI. The readability and the entire structure of the code are still fairly intact.

Parallelism now allows running cases which are large enough to be interesting, yet treatable with the current serialized I/O. The very large cases, destined to future tokamak experiments, may expose the I/O as inefficient (e.g. several minutes), and are advised to be handled with a parallel I/O technique. Due to time constraints we had to postpone this.

We list here shortly ideas about how the scaling property of the code can be improved:

- **Piggy-backing from multiple array exchanges in one**

  At each time step, the ghost cells of twelve arrays are exchanged, by groups of three, for a total of twelve boundary exchanges. One may reduce latency here by aggregating the three arrays communication in one.

- **Finer grain exchanges**

  The current ghost exchange primitive is general: with communication in all directions, which is not always required. One might *specialize* the ghost exchange routines to prevent unnecessary updates on a per-loop basis.

- ***n>1*-thick ghost layer**

  Using a thicker ghost layer (say, *n>1*) would allow a ghost cell exchange to occur every *n* steps instead of *n=1* as now. Each of these exchanges would involve *n* times the data, but being latency and not volume the problem, this would likely pay off in efficiency. This technique would also pose a very slight computational overhead.

Using the validation tool we observed results changing according to the optimization level and run size. In some cases we observe the parallel version results to be unacceptably diverging from serial. We consider an investigation to exclude a bug in

the code causing this to be important. Testing feedback from the project coordinator is essential here.

The compilation flags we would recommend to use at the moment with the Intel Fortran compiler are `-fast -ipo -fp-model=precise`.

We advise that future HLST optimization interventions should focus on the parallel I/O and the communication optimization. If larger runs are more important priority should be given to the former activity; otherwise to the latter. After getting acquainted with the new code, the PI will be able to decide on this.

4.6. *References*

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

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

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

In modern high performance computing, accelerators are widely used due to their high peak performance and efficient energy consumption. Therefore, the HELIOS supercomputer was recently (February 2014) upgraded with the new Intel Xeon Phi co-processor. The total number of such nodes is 180. Part of the BEUIFERC project is dedicated to analyze the performance of the new hardware by means of different micro-benchmark tests. A problem with Infiniband connections between distinct MIC cards was identified and resolved. This allows to achieve three times higher memory bandwidth and symmetric network performance. It was also observed that OpenMP overhead is about ten times larger on the MIC in comparison to the Intel Sandy Bridge. Finally, the execution time of a test *N-Body* code was analyzed and compared in different computational modes (*host*, *offload* and *native*) on the MIC partition of HELIOS.

5.1. ***The Intel Xeon Phi hardware***

In 2010 Intel announced a new family of processors named 'Many Integrated Core' (MIC) that provides a high peak performance with an efficient energy consumption. Later in 2012 Intel gave the brand name 'Intel Xeon Phi' to such processors. MIC is a co-processor unit, in the same fashion as the GPU systems from NVIDIA, which are already broadly used in high performance computing. The first card from the MIC family was a prototype product with the codename 'Knights Ferry' that was released in 2010 as a proof of concept. The second generation of the MIC cards was the first commercial product. It was released in 2011 under the name 'Knights Corner'. In the beginning of 2014 these accelerators were installed on the HELIOS supercomputer. The next generation of MIC cards with the codename 'Knights Landing' should be released in the end of 2015. For simplicity the following nomenclature will be used equivalent: MIC = Intel Xeon Phi = Knights Corner = co-processor = accelerator.

5.1.1. **The Intel Xeon Phi vs Intel Sandy Bridge**

Table 5 shows the comparison of the main hardware characteristics between Intel Xeon Phi and Intel Sandy Bridge processors, which are installed on HELIOS. A more detailed description of the MIC co-processor can be found in Ref. [1].

Processor	**Intel Sandy Bridge**	**Intel Xeon Phi**
Number of cores	8	60
Memory	32 GB	8 GB
Frequency	2.6 GHz	1.24 GHz
Peak performance	173 GFlops/s	1 TFlops/s
Memory bandwidth	40 GB/s	160 GB/s
Instruction execution	Out-of-order	In-order

Table 5 Hardware comparison between Intel Xeon Phi and Intel Sandy Bridge.

The memory bandwidth of the Intel Xeon Phi is four times larger than that of the Intel Sandy Bridge. One can also notice the improved factor of six for the peak performance. However, the memory bandwidth per core is much lower for the Xeon Phi (130 MB/core) compared to the Sandy Bridge (4 GB/core). The MIC peak performance per core of 16.6 GFlops/s is also smaller in comparison to the 21.6 GFlops/s obtained for the Sandy Bridge. Moreover, the 'in-order' instruction execution of the Intel Xeon Phi may lead to empty stages of pipelines. Consequently, two to four threads per core must be hosted on the Xeon Phi in order to get a significant fraction of its peak performance, whereas one thread per core is enough on the Sandy Bridge. Consequently, this makes programming of parallel modules much more complicated for the Xeon Phi in comparison to the Sandy Bridge.

5.1.2. **Intel Xeon Phi architecture**

The general architecture of the Intel Xeon Phi co-processor is depicted in Fig. 27 [2] including its main components: computation units, memory and network connections. The size of the boxes representing the computing elements is proportional to their peak performance and the size of the boxes representing memory components is proportional to the amount of available memory. The thickness of the lines corresponds to the memory bandwidth. The current generation of Xeon Phi co-processors (Knights Corner) is a co-processor; it is connected to the main CPU through the PCIe bus. The next generations of MIC cards, Knights Landing and Knights Hill will be available as stand-alone processor, which will significantly decrease the communication latency.

Two distinct MIC nodes are connected by means of the Infiniband (IB) port. At present, two configurations are available on the market with one or two IB links. In the next section the results of the network performance tests are shown for the MIC partition of the HELIOS supercomputer, which is equipped with two IB ports.

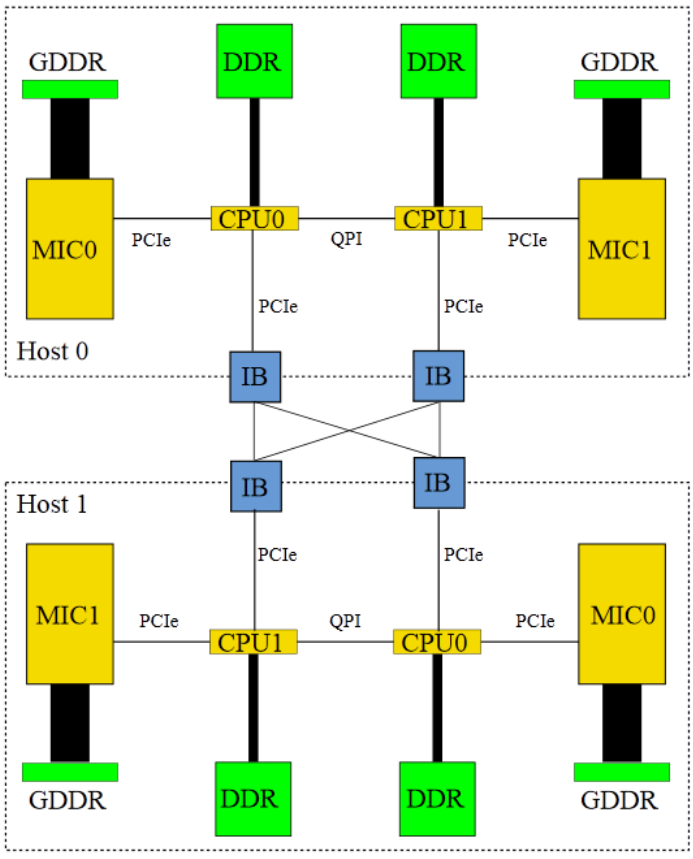

Fig. 27 The general architecture of two MIC nodes on the HELIOS supercomputer.

5.2. *MIC network performance*

The PingPong test from the Intel MPI Benchmark suite [3] was used to test the MIC 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* functions. When the message is received, process two sends the same data back to process one. The communication time is calculated afterwards as $time = \Delta t/2$, where Δt is the necessary time for the transaction from one process to another. The test is repeated 100 times and averaged values are estimated. The latency is defined as the time spent to send a one byte message. The bandwidth is estimated from the time spent to send a message of 4.2 MB between two processes.

First, the intra node network performance was tested and the results are shown in Table 6. The latency time between two processes running on the same CPU (0.31 µs) is about nine times smaller than on the MIC (2.70–2.84 µs). The latency increases further for the CPU–MIC communication and reaches its maximum (6.00 µs) when the message is transferred between MIC0–MIC1. According to Fig. 27 a message sent between two MICs must pass through one QPI and two PICe ports, which makes such communication slow with respect to its intra node counterparts where only up to one port is involved in the transfer process.

The memory bandwidth shows a similar behavior as the latency. The highest bandwidth (6531 MB/s) is reached when two tasks are running on the same CPU. The bandwidth measured inside the MIC (1928–1984 MB/s) is about three times lower than that. It further degrades for the communication between CPU–MIC and reaches its minimum (414 MB/s) for the MIC0–MIC1 transfer.

One can notice also that intra node network performance for the MIC is inhomogeneous between two different computing units. The latency and the memory bandwidth drastically degrade when MIC cards are involved in communication in comparison to the simple CPU–CPU transfer.

Host 0	CPU0	0.31		
	MIC0	3.4	2.70	
	MIC1	3.83	6.00	2.84
Latency (µs)		CPU0	MIC0	MIC1
		Host 0		

Host 0	CPU0	6531		
	MIC0	1618	1928	
	MIC1	480	414	1984
Bandwidth (MB/s)		CPU0	MIC0	MIC1
		Host 0		

Table 6 Intra node network performance of the MIC partition on the HELIOS supercomputer using the default set up. Left table represents the latency, while the right shows the memory bandwidth.

The inter node network performance was tested next. The MIC cards on the HELIOS supercomputers are connected by means of two Infiniband ports, as shown in Fig. 27. Therefore, the latency and the memory bandwidth for the host0mic0→hos1mic0 and host0mic1→host1mic1 tests should be very similar i.e. show symmetric results.

Table 7 represents the inter node network performance between two distinct MIC nodes. One can see that communication between two distinct MIC cards degrades further with respect to the intra node communication, from 2.84 µs to 6.95 µs for the mic1–mic1 case. This degradation is even more pronounced for the memory bandwidth that decreases from 1984 MB/s to 272 MB/s. Moreover, the expected symmetric results were not achieved. In all tests the communication between mic0–mic0 on two different nodes has shown better performance than between mic1–mic1 under the same conditions.

Host 0	CPU0	1.24		
	MIC0	3.80	5.97	
	MIC1	4.15	6.47	6.95
Latency (µs)		CPU0	MIC0	MIC1
		Host 1		

Host 0	CPU0	5943		
	MIC0	1640	984	
	MIC1	416	415	272
Bandwidth (MB/s)		CPU0	MIC0	MIC1
		Host 1		

Table 7 Inter node network performance of the MIC partition on the HELIOS supercomputer using the default set up. Left table represents the latency; right table shows the memory bandwidth.

The inter node results on the MIC partition of HELIOS were benchmarked with the results obtained from the identical PingPong test on the *robin* cluster, which is

equipped with a single IB port between MIC nodes [2]. Surprisingly the results were very similar as if only one IB ports was working properly. Therefore, we decided to investigate this issue in more detail. The first candidate for testing was the Direct Access Programming Library (DAPL) provider that defines the API functions that are used for a given communication between computational elements [4]. Intel MPI can select automatically the provider list but it does not necessarily yield the best performance. However, the provider list can be controlled manually through the environment variable *I_MPI_DAPL_PROVIDER_LIST*. Different sets of the provider combinations were tested and it was detected that they play an important role for the memory bandwidth. Table 8 represents the inter node memory bandwidth after using the following provider list, which gave the best performance: *ofa-v2-mlx4_0-1u, ofa-v2-mcm-1*. The memory bandwidth of the communication involving different MICs increased by a factor of three in comparison to the results obtained with the default provider (Table 7). However, the network performance was still not symmetric with 3447 MB/s memory bandwidth for mic0–mic0 transfer and 1349 MB/s for MIC1–MIC1.

Host 0	CPU0	5836		
	MIC0	4135	3447	
	MIC1	1780	2235	1349
Bandwidth (MB/s)		CPU0	MIC0	MIC1
		Host 1		

Table 8 The Inter node bandwidth of the MIC partition on the HELIOS supercomputer using the optimized DAPL provider list.

In order to test the two Infiniband connections between MIC hosts, it was decided to perform the PingPong test between all possible pairs of CPUs, which are placed on two distinct nodes. As one can see from Table 9, now both the latency and the memory bandwidth are very similar, proving that two IB ports are working properly and provide symmetric results.

Host 0	CPU0	1.27	1.23
	CPU1	1.28	1.25
Latency (µs)		CPU0	CPU1
		Host 1	

Host 0	CPU0	4987	5029
	CPU1	5075	5058
Bandwidth (MB/s)		CPU0	CPU1
		Host 1	

Table 9 The inter node network performance of the MIC partition on the HELIOS supercomputer between all possible CPUs pairs.

A low level debugging was necessary to clarify the different results of tables 4 and 5. It has shown that for the communication between CPUs the system automatically chooses the fastest way from one CPU to another using different IB ports. However, when the MIC cards are involved in the communication the system constantly uses only one IB port, which is assumed to be connected to CPU0. This explains why the results from the inter node PingPong test were not symmetric with respect to the memory bandwidth, which was three times higher when the message was sent between mic0–mic0 than between mic1–mic1 (Table 7).

The problem was reported to the CSC-Support team. They recommended to use additional files namely ‘dat.conf’, which should be copied on each MIC card and include the explicit specification of the Infiniband providers that would be used by the co-processors. The content of these files for MIC0 and MIC1 is the following:

- dat.conf for MIC0:
  ofa-v2-mcm-1 u2.0 nonthreadsafe default libdaplomcm.so.2 dapl.2.0 'mlx4_0 1' ''
  ofa-v2-mlx4_0-1u u2.0 nonthreadsafe default libdaploucm.so.2 dapl.2.0 'mlx4_0 1' ''
- dat.conf for MIC1:
  ofa-v2-mcm-1 u2.0 nonthreadsafe default libdaplomcm.so.2 dapl.2.0 'mlx4_1 1' ''
  ofa-v2-mlx4_1-1u u2.0 nonthreadsafe default libdaploucm.so.2 dapl.2.0 'mlx4_1 1' ''

Afterwards, the path to these files should be specified in the environment variable *DAT_OVERRIDE* before the program execution.

Finally, the results from the PingPong test between MIC cards on two distinct hosts became symmetric and provided low latency and high memory bandwidth (see Table 10). Nevertheless, the point is that such high performance should be achieved automatically, which makes this still an open issue. The same tests were also performed on another supercomputer named *supermic,* which is located at the Leibnitz-Rechenzentrum (LRZ) in Garching and has a similar MIC partition structure to HELIOS, i.e. two IB ports. The results show similar asymmetric network performance as obtained initially on HELIOS. It seems that the problem is connected to the hardware driver, which is currently under investigation.

Host 0	MIC0	5.88	5.87
	MIC1	5.91	5.94
Latency (µs)		MIC0	MIC1
		Host 1	

Host 0	MIC0	3340	3338
	MIC1	3345	3330
Bandwidth (MB/s)		MIC0	MIC1
		Host 1	

Table 10 The network performance of the MIC partition on the HELIOS supercomputer between all possible MIC pairs, which are located on two distinct nodes.

The *intelmpi4.1* library was used in the previous tests. Thus, it was obvious to check also the network performance with the new *intelmpi5.0.3.048* library, which is available on HELIOS. An unexpected, interesting behavior of the PingPont benchmark was detected. The first test between two distinct MIC cards, for example mic0–mic0 provided always low memory bandwidth. All following tests for the same accelerators gave a high memory bandwidth. However, when the test was switched to other MIC cards, for example mic1–mic1, the memory bandwidth became low again. The following tests for the same cards produced again a high bandwidth as it is shown in Fig. 28. The issue was reported to CSC support and is still under investigation.

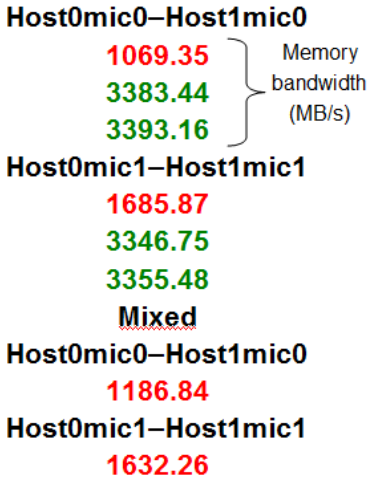

Fig. 28 The memory bandwidth between two distinct MIC nodes from the PingPong test using the *intelmpi5.0.3.048* library.

5.3. *OpenMP overhead micro-benchmark on the MIC partition of the HELIOS supercomputer*

During the establishment of a parallel region using the OpenMP library some overhead appears. The overhead time of different OpenMP constructs was measured by means of the Edinburgh Parallel Computing Center (EPCC) OpenMP micro-benchmark suite [5]. In this benchmark the overhead is calculated as the difference in execution time between the parallel and sequential version of the program. The benchmark is developed in such a way that each thread of the parallel version performs the same amount of work as the sequential one.

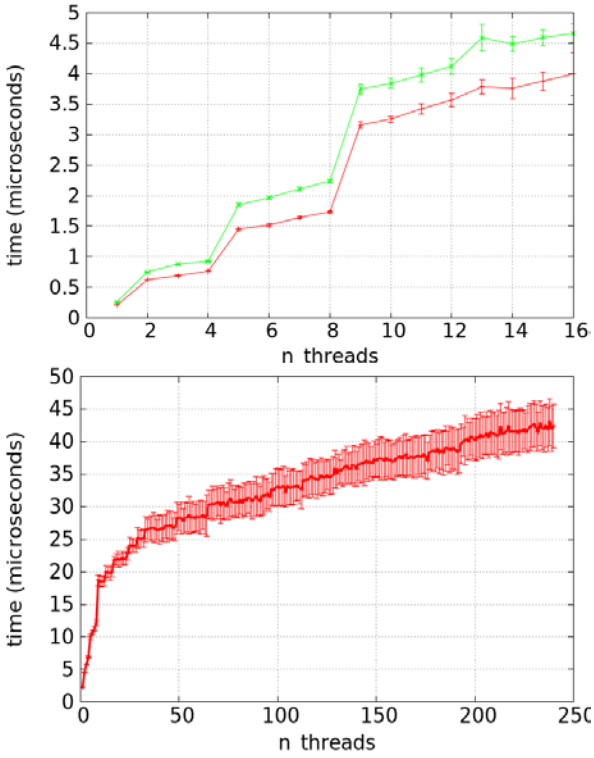

Fig. 29 The overhead time for the 'reduction' OpenMP pragma as a function of the number of threads. Top figure represents the results from the Intel Sandy Bridge (red line – normal Helios node; green line – host of the MIC co-processor). Bottom figure shows the results from the MIC native mode.

The overhead for the 'reduction' OpenMP pragma is plotted in Fig. 29. One can see that the overhead time for the normal Intel Sandy Bridge node (Fig. 29 top, red line) is a little bit smaller than for the Sandy Bridge node on the MIC partition (Fig. 29, green line). The overhead time increased about ten times when the test was launched on the Intel Xeon Phi (Fig. 29, bottom) and reached more than 40 µs, when 240 threads were used.

The overhead time can be even more pronounced when the OpenMP functions share a large dataset between the threads. In some cases it can be of the same order as the computational kernel time and results in a significant bottleneck. Fig. 30 shows the overhead time of the 'firstprivate' OpenMP pragma. For this function the master thread shares the dataset between all slave threads. For the test shown a 60 kB array was chosen. The overhead time did not grow much on the Sandy Bridge processor and stayed between 30–40 µs when the number of threads was increased from one to sixteen. However, on the MIC card the overhead time rose significantly (~600 µs) when more than ten threads were involved in the data sharing and reached almost 1.8 ms for 236 threads.

It was also observed that the overhead time depends linearly on the array size for the 'firstprivate' pragma. Therefore, when performing computations on the MIC partition

the user should carefully check the overhead time of different OpenMP pragmas, especially when sharing large datasets among the threads.

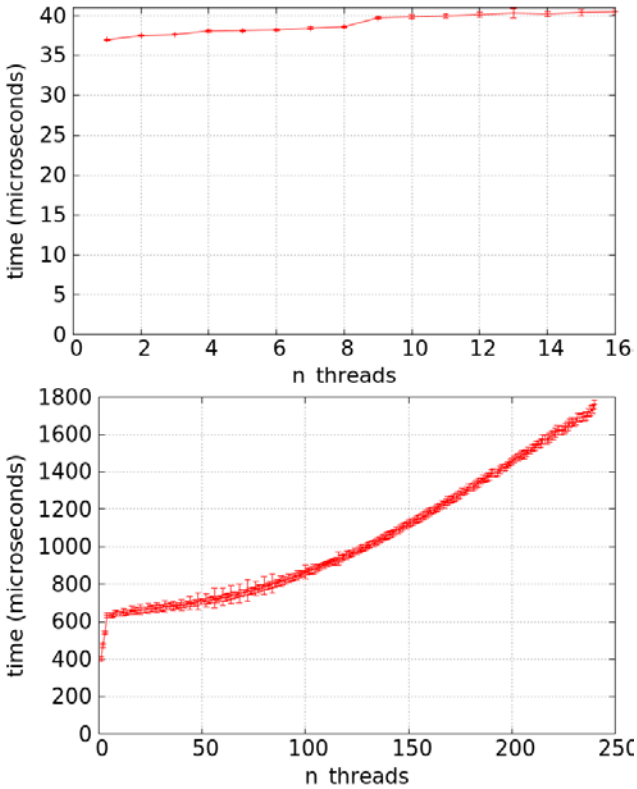

Fig. 30 The overhead time for 'firstprivate' OpenMP pragma as a function of the number of threads. Top figure represents the results from the Intel Sandy Bridge processor; bottom figure shows the results from the MIC accelerator. A 60 kB array was used in this test.

5.4. *Host, offload and native computation modes on the MIC partition*

At present, three operational modes are available on the MIC co-processor: *offload*, *symmetric* and *native.* In the *offload* mode the host offloads a part or all the computation on the co-processor. The OpenMP 4.0 library provides directives for offloading procedure where users can specify the offloading target and the data that should be sent to the co-processors. These functions should simplify the code development. Unfortunately, the data transfer between the host and the MIC is a strong bottleneck in this mode. Moreover, the simultaneous computation on both the CPU and the MIC requires asynchronous offload pragmas that are often a source of load balancing problems. However, the next generation of the MIC family might be part of the socket and the co-processor nomenclature would disappear for the Intel Xeon Phi together with the offload mode.

In the next mode named *symmetric,* the application runs on both the MIC and the host using two different binaries, which should be compiled independently for the host and the Xeon Phi. They communicate by means of the MPI functions and the system treats the MIC cards as another node in the cluster environment.

One of the differences between the MIC co-processor and GPU accelerator is that a Linux micro operating system is running on the Intel Xeon Phi, whereas GPUs are operated by the system of the host. Therefore, this allows to treat the MIC card as another node on the supercomputer. The program execution using only the MIC co-processor without the host is called *native* mode. The advantage of such a mode is that in order to transfer MPI/OpenMP application on the MIC the user need only to re-compile the code with the flags for the Xeon Phi operation environment. The main disadvantage lies in the amount of the available memory on the MIC card. Two versions of the Knights Corner are available on the market, with a memory of 8 GB

(installed on HELIOS) and 16 GB respectively. In comparison, the Intel Sandy Bridge nodes on HELIOS have 2 x 32 GB.

Since November 2014 the *native* execution mode is available on the MIC partition of the HELIOS supercomputer. It was decided to compare the execution time of a test application using different operation modes (Table 11). For this test an *N-Body* code [4] was used.

Number of threads	Intel Sandy Bridge	Intel Xeon Phi (offload)	Intel Xeon Phi (native)
1	55	130.61	126.60
2	28	66.47	62.69
4	14	33.78	30.75
8	7	18.78	15.86
16	3.5	12.02	9.97
32		7.44	4.72
64		6.19	3.46
128		4.09	1.59
236		3.96	1.39

Table 11 The execution time of the test *N-Body* code on Intel Sandy Bridge and Intel Xeon Phi using different computational modes and number of cores.

As one can see, the execution time using one core on the Sandy Bridge is about 2.29 times faster than with the Xeon Phi *native* mode. This is coherent with the processor clock speed, which is 2.1 times higher on the Sand Bridge in comparison to the Intel Xeon Phi. Using the maximal number of cores on the Sandy Bridge (16) the execution time decreases to 3.5 s. The execution time shrinks from 130.61 s to 3.96 s when 236 threads were used for computations on the MIC card in the *offload* mode. Such result show that performing simulations on the MIC card when using the maximal number of threads in the *offload* mode can provide worse results than on the Sandy Bridge node. However, in *native* mode on the Intel Xeon Phi with the maximal number of threads, the execution time decreased to 1.39 s, which is 2.5 times faster than on the Sandy Bridge and 2.85 times faster than with the *offload* mode.

5.5. *Tests of MPI 3.0 standard*

New features of MPI 3.0 standard were also tested on the HELIOS cluster as a part of the current project. Among them are non-blocking collective communication and one-sided communication with focus on a shared memory (RMA – remote memory access). The tests were done by means of the Intel MPI Benchmark 4.1 suite.

5.5.1. **Non-blocking collective communication**

A collective communication of MPI 1.0 and MPI 2.0 standards was extended in MPI 3.0 with a non-blocking collective (NBC) communication: *MPI_IBcast*, *MPI_IReduce*, *MPI_IGather*, *MPI_IAlltoall*, etc. Example of a pseudo-code for the non-blocking *IBcast* is shown in Fig. 31. In comparison with the blocking *Bcast* the non-blocking *IBcast* has one extra parameter *REQUEST*, which is used for task synchronization afterwards. In the beginning the non-blocking MPI function is called (*CALL MPI_IBCAST(array1,…)* in our example). After this call all tasks can perform any calculations, except that such calculation should not use data that is involved in the non-blocking transfer, without synchronization of the *IBCAST* call. Finally, the outstanding *IBCAST* is completed with *MPI_WAIT* call. The advantage of such non-blocking communication is that the calculation can overlap with the communication improving code performance. Such overlap can be measured with the Intel MPI Benchmark 4.1 suite.

```
INTEGER REQUEST, COMM, STATUS(MPI_STATUS_SIZE), IERROR
INTEGER, DIMENSION(0:9) :: array1
INTEGER, DIMENSION(0:9) :: array2
INTEGER root=0
...
CALL MPI_IBCAST(array1, 10, MPI_INT, root, COMM, REQUEST, IERROR)

! do some calculation with any data except array1
! CALL CALCULATION(array2)

CALL MPI WAIT(REQUEST, STATUS, IERROR)
```

Fig. 31 Pseudo-code for a non-blocking MPI IBcast.

The Intel MPI intelmpi/5.0.3.048 library which includes all new features of the MPI 3.0 standard was tested on the HELIOS cluster. In order to maximize overlap between communication and calculation the asynchronous progress support has to be enabled. This can be done for the Intel MPI library via the environment variable: *export MPICH_ASYNC_PROGRESS*=1. Fig. 32 shows the communication-calculation overlap with *ASYNC*=0 (red line) and *ASYNC*=1 (green line). The overlap was calculated by using the following formula:

$$overlap = 100.* max(0, min(1, (t\_pure+t\_CPU-t\_ovrl) / min(t\_pure, t\_CPU)),$$

where *t_pure* is the time of a pure communication operation; *t_CPU* is the time computation takes to complete when run concurrently with the nonblocking communication operation; *t_ovrl* is the time of the nonblocking communication operation takes to complete when run concurrently with a CPU activity. For more details of this benchmark refer to [3,6]. One can see that with the asynchronous progress support the overlap is about 50 percent for small message sizes and reaches almost 100 percent for large messages, while without the asynchronous progress the overlap is low for small and large messages and reaches 70 percent for intermediate message sizes.

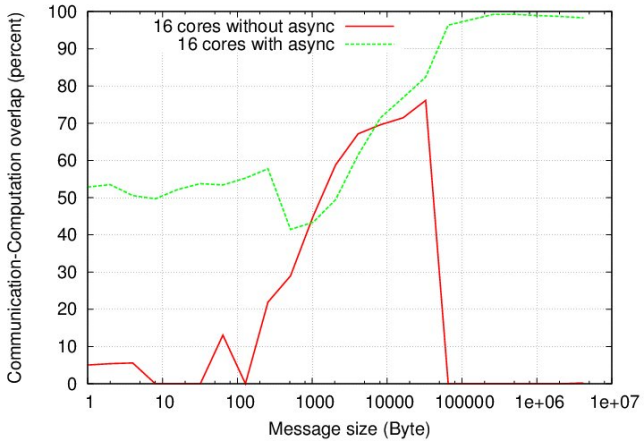

Fig. 32 The communication-computation overlap vs. the message size from the test of MPI *Ialltoall* function with (green line) and without (red line) the asynchronous progress support.

The number of processes being involved in the communication and how they are pinned over the computational nodes play an important role for the communication-calculation overlap of the non-blocking MPI functions. We tried to find the best configuration on the HELIOS cluster, which is equipped with the Intel Xeon E5 processor, Sandy-Bridge EP 2.7GHz (eight cores each, two processors per node).

We performed these tests on four nodes evenly distributing MPI tasks over them. Fig. 33 shows the communication-calculation overlap for the MPI *Ialltoall* function using different number of processes. One can see that the number of tasks being involved in the computation has a significant effect on the overlap. However, for all cases the overlap for small message sizes is lower than for large messages. Moreover, pinning 16 tasks on one node gave us much smaller overlapping values than distributing

them among four nodes. This is due to the larger communication time between the tasks if they are located on different nodes. Nevertheless, we measured the fastest absolute computation time when the 16 tasks were pinned on one node.

From Fig. 33 we determined that the best condition (the largest overlap) for the non-blocking *Ialltoall* function is 8–16 processes, which are evenly distributed among four nodes. Therefore, we decided to use the same configuration to test other non-blocking collective functions from MPI 3.0. Fig. 34 shows the communication-computation overlap of different MPI 3.0 non-blocking functions as a function of the message size. One can see a similar behavior for all functions. The overlap is relative low for the small message sizes (between 20 and 55 percent) and increases to 85–99 percent for the large messages. These results are in agreement with similar calculation made by Intel [6].

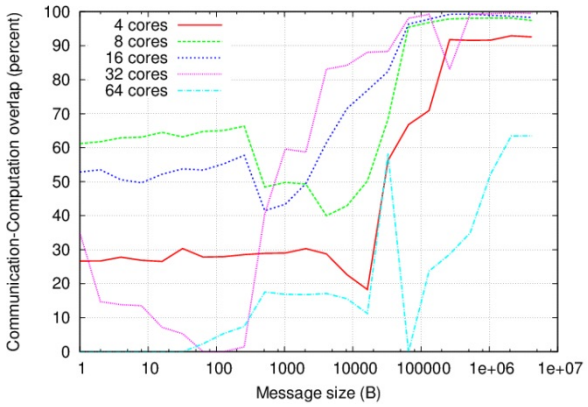

Fig. 33 The communication-computation overlap vs. message for the MPI *Ialltoall* function using different numbers of processes.

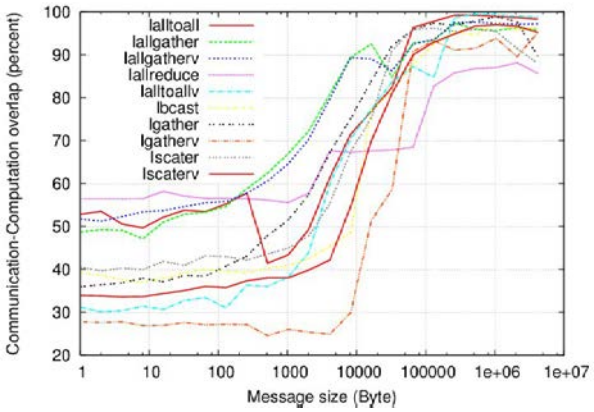

Fig. 34 The communication-computation overlap vs. message size for different MPI functions using 16 processes evenly distributed over four nodes.

5.5.2. **Remote memory access**

The remote memory access (RMA) or one-sided communication interface was significantly extended in the MPI-3 standard. Several major features were introduced including different memory models, new communication and synchronization calls, and new ways of creating RMA windows described below. The major update of the synchronization was performed in the implementation of the passive target communication mode, which was tested on HELIOS.

One-sided communication decouples data transfer and synchronization in such a way that unneeded synchronization is eliminated allowing larger concurrency. For this communication type only one process specifies all communication parameters, both for the sending side and for the receiving side. Data is directly read or written to the memory of a remote process. The memory that a process allows other processes to access by one-sided communication is called window. In Fig. 35 it is shown how

process number one reads data from the memory window of process number two and write data to the memory window of process number three. Memory windows belonging to the different processes can have different sizes. Processes define their local window and inform other processes about it by calling the collective function *MPI_Win_create*.

There are two types of one-sided communication: active and passive target communication. In the former both processes are explicitly involved in the communication. All data transfer arguments are provided by the first process, while the second participates only in the synchronization. In the passive target communication only the origin process is explicitly involved in the transfer.

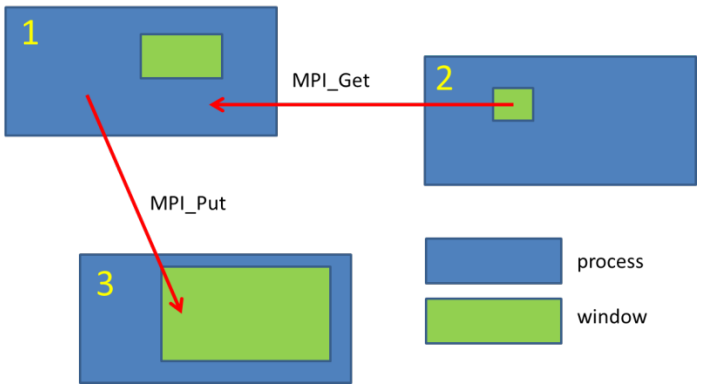

Fig. 35 Schematic view of the one-sided communication using a memory window.

The passive target communication was tested on HELIOS by means of the Intel MPI Benchmark 4.1 suite. The *truly_passive_put* test was performed with and without the asynchronous progress support described above. This test measures two things. The time consumed by the origin process to complete the MPI function *MPI_Put()* while the target process is sitting on the MPI stack (*MPI_Barrier*). We call the measured time as *time pure*. The second quantity that was measured in this test was the time consumed by the origin process to complete the MPI function *MPI_Put()* while the target process performs computations outside the MPI stack and then calls *MPI_Barrier()*. The computational time is about the same as the function call to *MPI_Put()*. We call this time as *time with computation*. Therefore, without any improvement techniques the *time with computation* should be greater than *time pure* (about two times). But if the MPI implementation provides truly passive mode of the remote memory access, both timings should be about the same.

Time pure and *time with computation* for the *truly_passive_put* test are shown in Fig. 36, top for small message sizes and bottom for large messages. The test was performed with and without asynchronous progress support. One can see that asynchronous progress brings significant overhead for small messages (green and violet colors). For messages up to 8 KB, both measured timings are about two times larger than their counterpart without the asynchronous progress support. For the large message size this overhead vanishes.

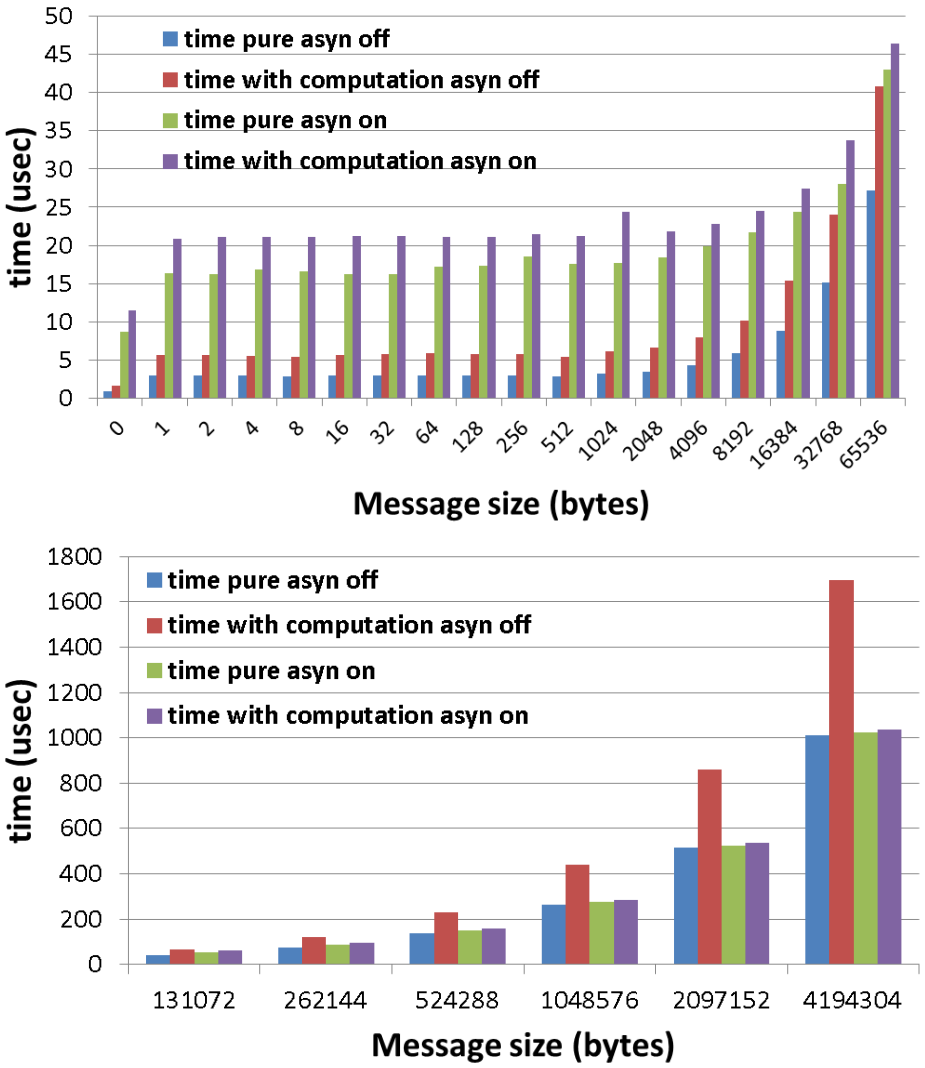

Fig. 36 *Time pure* and *time with computation* for the *truly_passive_put* test with and without asynchronous progress support. Top figure for small message sizes, bottom for large messages sizes.

In Fig. 36 one can also see that *time pure* is always 1.5–2 times smaller than *time with computation* when asynchronous progress support is switched off. Therefore, one-sided communication does not work perfectly in this case. However, with asynchronous progress support the difference between *time pure* and *time with computation* is about 10–20 percent for small messages and less than five percent for large message sizes. Thus, we can summarize that the passive target communication for remote memory access of the MPI 3.0 standard works properly only when asynchronous progress support is switched on. Nevertheless, it is noteworthy that for small message sizes it can be more efficient in absolute terms to switch off the asynchronous progress support due to the overhead introduced. These results are in good agreement with similar tests performed by Intel [6].

5.5.3. **Shared memory**

The usage of shared memory segments was also introduced in the MPI standard 3.0. The shared memory is a portion of memory that is attached to some address space and is accessible for all processes within a group for usage. Such opportunity will allow to save memory space in a pure MPI program if MPI tasks have many common data. Moreover, an important speed-up can be achieved in a particular program if the MPI tasks should exchange large data within one computational node.

The establishment of the shared memory segments by means of the MPI 3.0 standard was tested on HELIOS. For this purpose the MPI function

MPI_WIN_ALLOCATE_SHARED was used. It is a collective call executed by all tasks in the same group. Each MPI task allocates memory that is shared among all processes in the group and returns a pointer to the local allocated segment. The local allocated memory can be used then for load/store accesses by remote processes. The address of the local allocated shared memory can be obtained by other MPI tasks in the group by means of the MPI function *MPI_WIN_SHARED_QUERY*.

In the test code one MPI task creates a shared memory segment with the size of 100 MB and fills it with an array of double precision elements. Subsequently all MPI tasks calculate a sum of all elements by using the shared memory access. The results show identical sums for all MPI tasks thereby confirming correctness of the test code and the shared memory access in the MPI 3.0 standard.

The test presented above was performed using 16 MPI tasks within one computational node. Next step was to test the establishment of the shared memory segments simultaneously on many nodes using a large number of MPI tasks. For this purpose the code was modified. The MPI 3.0 standard provides a function for splitting all MPI tasks (*MPI_Comm_split_type*), which are associated with one *mpi_comm_world* into disjoin subgroups. Within each subgroup the processes are ranked. Therefore, each MPI task gets two rank numbers: the global rank and the local rank. Using the parameter *MPI_COMM_TYPE_SHARED* in the function *MPI_Comm_split_type* allows to automatically create subgroups of MPI tasks, which are pinned on the same node with new *comm* communicator. Afterwards each subgroup defines its own shared memory segment. The same test described above (sum of elements of the double precision array) was performed using 512 nodes (8192 MPI tasks). Therefore, 512 separated shared memory segments were created and each MPI task calculated the sum accessing the data in the shared memory. The test code worked correctly and provided accurate results.

Many tests were performed that deliberately pushed the test code to crash. For example, processes wrote and read data outside the shared memory segment. In all cases segmentation faults occurred and the code crashed. Thus, developing production codes using the shared memory segments requires careful evaluation of the necessary memory and careful manipulation of the pointers that are used to access such memory.

The last test was to measure the overhead time of the MPI function *MPI_WIN_ALLOCATE_SHARED* which is used to establish the shared memory region. It was found that the overhead time does not depend on the size of the allocated shared memory segments. We tested one byte, one MB and one GB shared memory allocation. In all these tests the overhead time was about 1.2–$3*10^{-4}$ s.

All aforesaid tests were performed by means of the *intelmpi/5.0.3.048* library. Unfortunately, the latest available *bullxmpi/*1.2.9.1 library not yet includes the functions for the shared memory segments.

5.6. ***Test of MPI 2.0 standard***

During the current project we provided support to the CSC team in order to resolve the ticket number #5523 – "*A ticket in RT: error when changing number of CPU in a job*". An error appears when the code reads a checkpoint (binary file) with double (or more) the number of nodes than those that have been used for writing it. The code crashes if the number of nodes being involved in the reading procedure is >=512. For the code compilation the *bullxmpi* library was used. Analysis by the owner of this ticket shows that the error came from a call to *MPI_FILE_READ_ALL*.

We decided to create two small codes in order to test the MPI 2.0 standard (collective IO operation) on the HELIOS cluster. The first program produces a binary file and each task writes a double precision array with ten elements (80 bytes) in the file. In this code we used the following MPI 2.0 functions: *MPI_File_Set_View* –

changing tasks view of data in file and *MPI_File_Write_All*. The second code performs data reading from the previously generated binary file by means of the MPI function *MPI_File_Read_All.* Both codes were tested with the *intelmpi* and *bullxmpi* libraries.

At the beginning the code for creating the binary file was tested. It was found that until 128 nodes (2048 MPI tasks) the code was working fine using both *intelmpi* and *bullxmpi* libraries. However, when 256 nodes (4096 MPI tasks) were involved the code crashed within *intelmpi.* With the help of the CSC support team it was detected that the problem occurred in the *intelmpi/5.0.3.048* library. Re-compiling the code with new the *intelmpi/5.1.1.109* resolved the problem with 256 nodes but the code crashed again with 512 nodes (8192 MPI tasks). Debugging showed that the code hanged during execution of the MPI function *MPI_File_Open.* The problem was reported to CSC support and was escalated afterwards to the HPC support. They could identify the problem, which was connected to the *srun* command and the *DAPL_UD* provider. The problem could be avoided by launching the code with *mpirun* instead of *srun* with the following additional environment variables:

- export I_MPI_FABRICS=shm:dapl
- export I_MPI_DAPL_UD=1
- export I_MPI_DAPL_UD_PROVIDER=ofa-v2-mlx4_0-1u
- export I_MPI_DAPL_UD_RDMA_MIXED=0

Finally, the test code for writing a binary file was tested on 1024 nodes (16384 MPI tasks). Even by using all aforesaid corrections the code crashed again. The problem was sent to CSC support. It is still under investigation. In contrast to the *intelmpi* library, *bullxmpi* works without any problems with all tested of nodes (till 512) and provided a correct output binary file.

Next step was to test the code for reading the binary file. For this test we took the file which was created on the previous step by using 4096 MPI tasks (256 nodes). The test code for reading the binary file was launched using different number of nodes from 16 to 512. All runs completed normally and provided a correct output.

The test codes and output results were sent to the ticket owner and to the CSC support. It was reported that these kernel routines correctly represent the part of his program where the problem occurred. The reason why the MPI collective file reading with different node numbers crashed in the ticket owners program and not in the test code was detected afterwards by the CSC support. The bug was neither present in the program nor in the MPI function *MPI_File_Read_All* but instead in the *stripe count* of *Lustre* file system installed on HELIOS. The problem was resolved by increasing *stripe count* to the maximum (*setstripe -c -1*).

5.7. *Conclusions*

The Intel Xeon Phi co-processor has a theoretical peak performance six times higher and a memory bandwidth of four times larger than the Intel Sandy Bridge processor. This makes such accelerator in principle very attractive for high performance computing. However, in order to achieve such benefit a huge effort in programming must be invested.

In this project the network performance was tested on the MIC partition of the HELIOS supercomputer by means of the Intel MPI Benchmark suite. It was found that the two Infiniband ports per node do not work properly and provide asymmetric results during the PingPong test between two distinct MIC nodes. An additional low level manual set up had to be introduced in order to achieve symmetric results with high memory bandwidth. Moreover, a problem with the *intelmpi5.0* library was identified that in particular cases resulted in low memory bandwidth. The Direct Access Programming Library provider was also tested. The results show that it plays a significant role for the memory bandwidth enhancement that could be increased by a factor of three with the correct provider list.

The different OpenMP pragma overheads were measured on both the Sandy Bridge node and the MIC card using the EPCC micro-benchmark suite. The overhead time increased at least by a factor of ten on the MIC in comparison to the Sandy Bridge due to the large number of threads. For some OpenMP pragmas the overhead time on the MIC card, when using the maximal numbers of threads, can be in the same range as the computation kernel.

Finally, the *host*, *offload* and *native* modes of execution of the MIC partition nodes were tested by means of an *N-Body* code. In the *offload* mode the execution time using the maximal number of threads on the MIC card is higher than on the Sandy Bridge processor. However, when the application was launched in the *native* mode the execution time decreased by a factor of more than two in comparison to both the Sandy Bridge and the MIC in the *offload* mode.

The non-blocking collective communication, the remote memory access and the shared memory segments from MPI 3.0 standard were tested on HELIOS. Efficient communication-calculation overlap (85–99 percent) can be achieved only with large message sizes (>10 KB), while for small messages this overlap is between 20 and 55 percent. The passive target communication in the remote memory access of the MPI 3.0 standard works properly only when the asynchronous progress support is switched on allowing to achieve more than 95 percent overlap between calculation and synchronization. However, for small messages (<8KB), the overhead increases when using the asynchronous progress support resulting in longer absolute execution time. The implementation of the shared memory segments by means of the MPI standard 3.0 was also tested on HELIOS. The test code provided correct results showing that all tasks can have direct access to the shared memory region (read/write data), which was created before by the master task. This technique will allow in MPI programs to save an important amount of memory if all MPI tasks share large common data.

The parallel IO from the MPI 2.0 standard was also tested on HELIOS during the support of resolving the ticket number 5523. Two test codes were developed for parallel writing/reading a binary file. By means of these codes few bugs of the *intelmpi* library were found. They were resolved with help of the CSC support and the codes were successfully tested afterwards up to 512 nodes.

5.8. ***References***

[1] Rahman R., *Intel Xeon Phi Coprocessor Architecture and Tools,* Apress Media, LLC, 2013.
[2] Haefele M., *Final report on HLST projects BEUIFERC/GOMIC*, 2014
[3] https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[4] James Reinders, Jim Jeffers, *High Performance Parallelism Pearls*, Morgan Kaufmann, 2015.
[5]https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite
[6] Mikhail Brinskiy et al., “Mastering Performance Challenges with the New MPI-3 Standard”

6. **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 present, it is 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.

6.1. ***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:
  n_tri_p = 4**nv***n_points**2*(*n_R*+*n_Z*-2)
- Number of triangles in the wall: *n_tri_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 *n_tri_p*, (ii) *nwu* and *nwv* for *n_tri_w*, and (iii) *n_harm.* A large scale production run should finally correspond to the parameters: *n_tri_p*=$2*10^5$, *n_tri_w*=$5*10^5$, *n_harm*=11.

6.1.1. **Memory consumption analysis**

Fig. 37 shows the memory consumption of the most important individual subroutines during the scan of the parameter *n_tri_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. 38 represents the memory consumption of the same subroutines as it was shown in Fig. 37, however, this time with a parametric scan in the number of triangles within the plasma (*n_tri_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 *n_tri_p*=$2*10^5$.

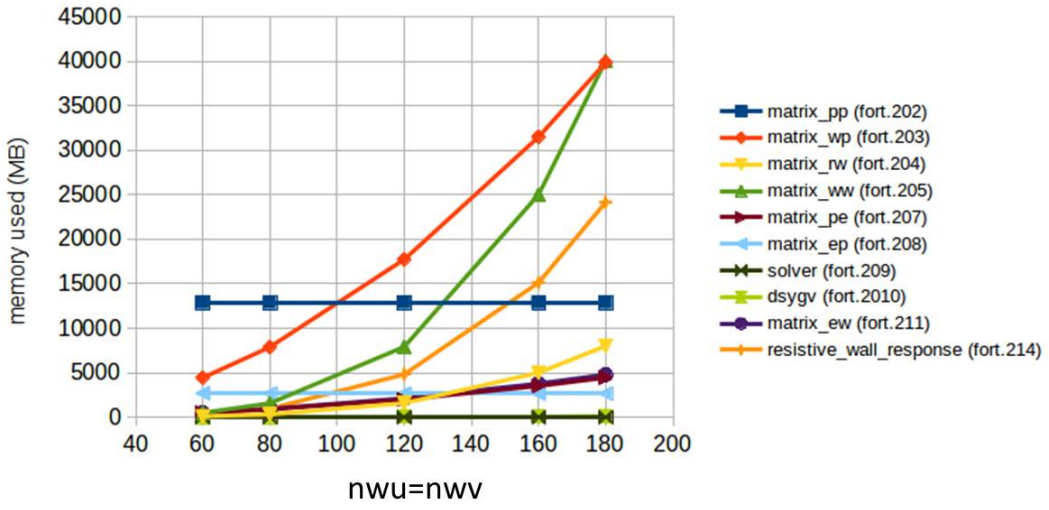

Fig. 37 The memory consumption of individual subroutines of the STARWALL code during the scan over the number of the triangles discretizing the wall (*n_tri_w*=2**nwu***nwv*).

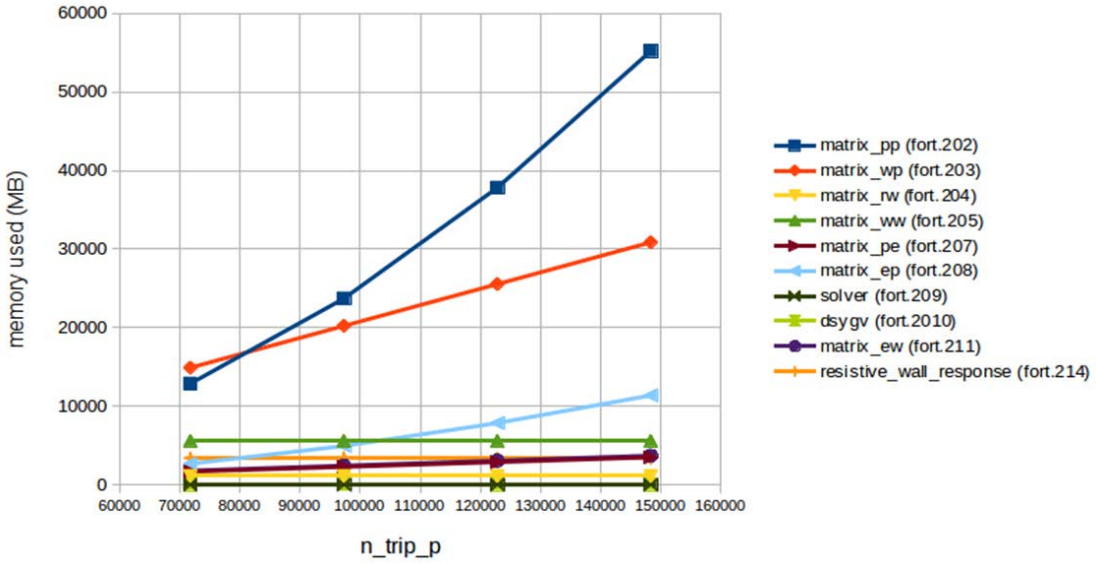

Fig. 38 The memory consumption of individual subroutines of the STARWALL code during the scan over the number of the triangles within the plasma (*n_tri_p*).

The last parameter tested was the number of sin/cos harmonics (*n_harm*). Fig. 39 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 test case 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.

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.

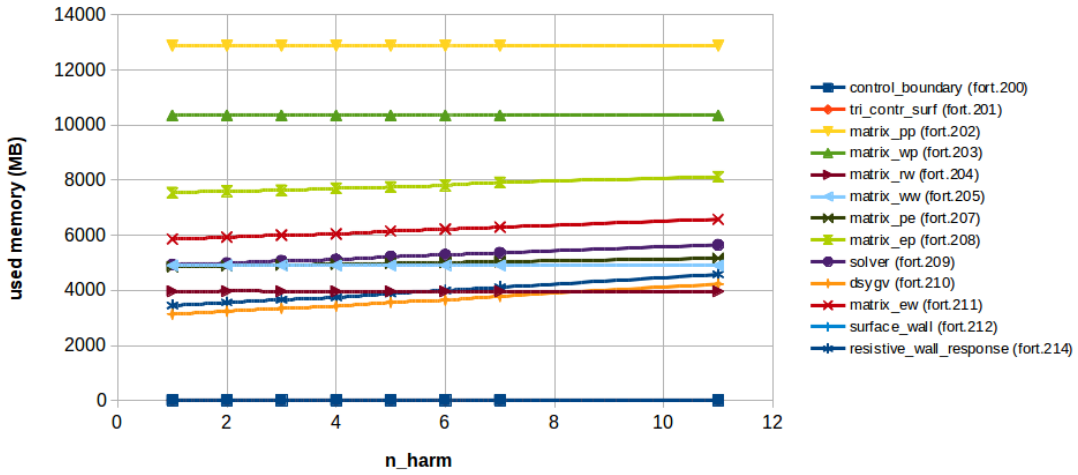

Fig. 39 The memory consumption of individual subroutines of the STARWALL code during a scan over the number of sin/cos harmonics.

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 HELIOS cluster.

6.1.2. **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. 40 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.

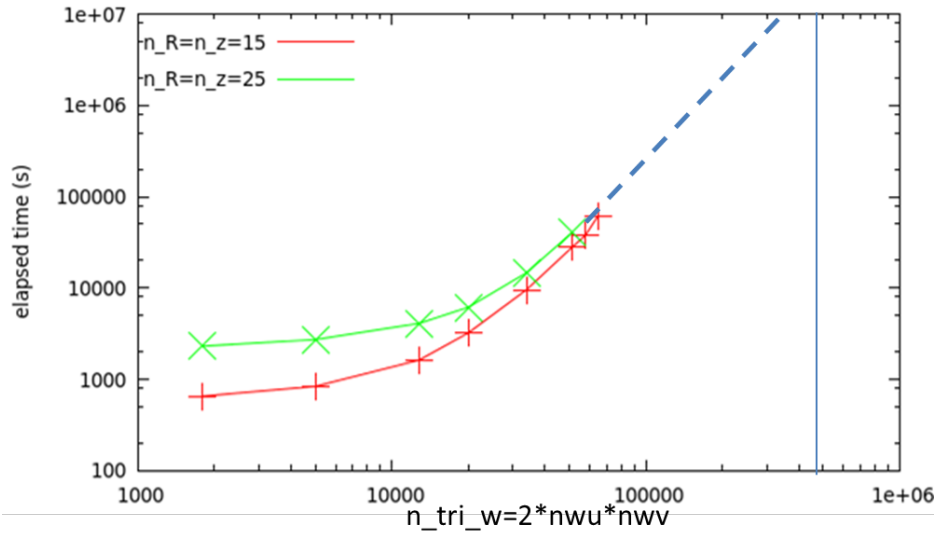

Fig. 40 The wall clock time versus the number of triangles in the wall (n_tri_w) for different numbers of triangles within the plasma: *n_R=n_Z*=15 shown as red line, *n_R=n_Z*=25 shown as green line. The solid blue line shows the targeted numbers of triangles for a production run, while the dashed blue 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 > 80%. Hence, this subroutine became the first candidate for parallelization effort and improvement.

6.1.3. **OpenMP parallelization analysis**

STARWALL is partially parallelized by means of OpenMP directives. Its parallelization efficiency is shown in Fig. 41. 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.

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. 42. 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.

6.1.4. **LAPACK subroutines**

As it was discussed earlier the code spends most of the computation 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.

6.1.5. **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 behavior 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.

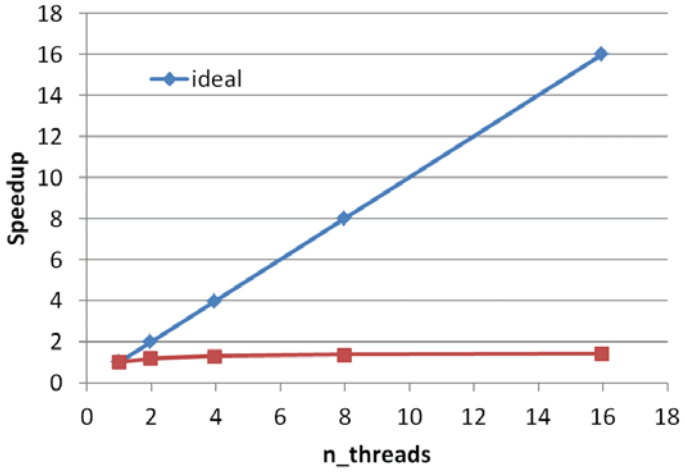

Fig. 41 Speed-up of code versus number of OpenMP threads.

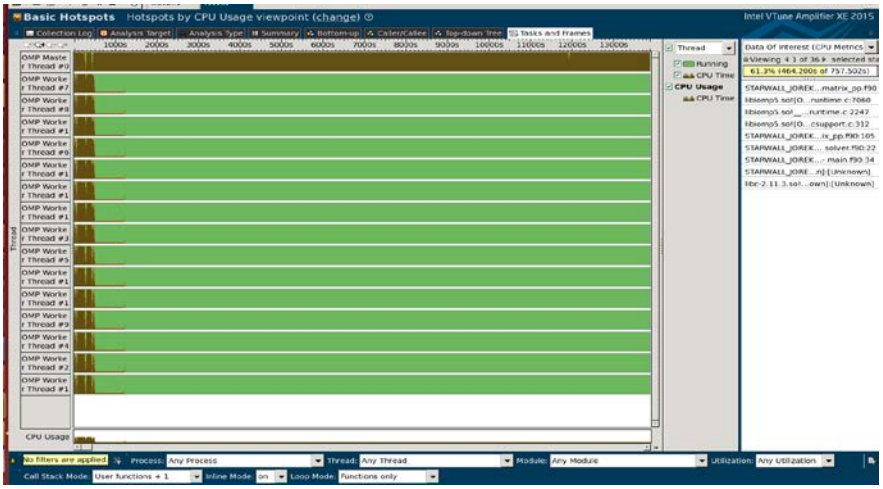

Fig. 42 Basic Hotspots analysis from the Intel Vtune amplifier using 16 OpenMP threads. Brown color shows the working status of the process, while green color corresponds to the idle state.

6.2. *MPI parallelization*

6.2.1. **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 60% 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].

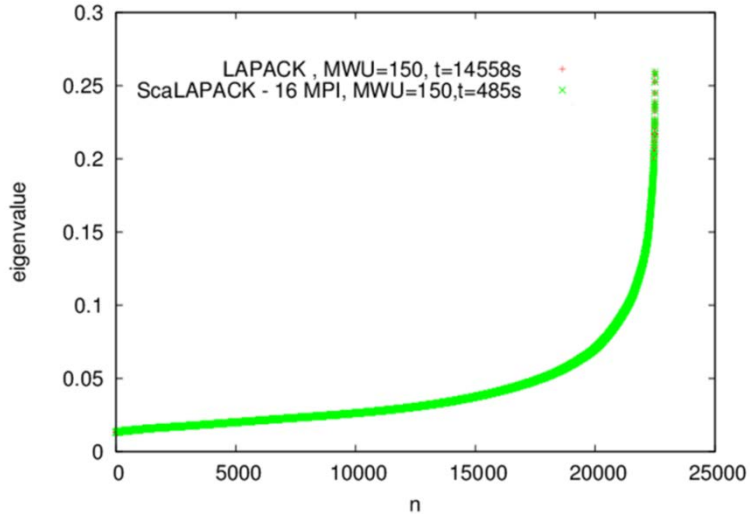

Fig. 43 Eigenvalues from the sequential LAPACK *dsygv* (red points) and the parallel ScaLAPACK *PDSYGVX* (green points) subroutine.

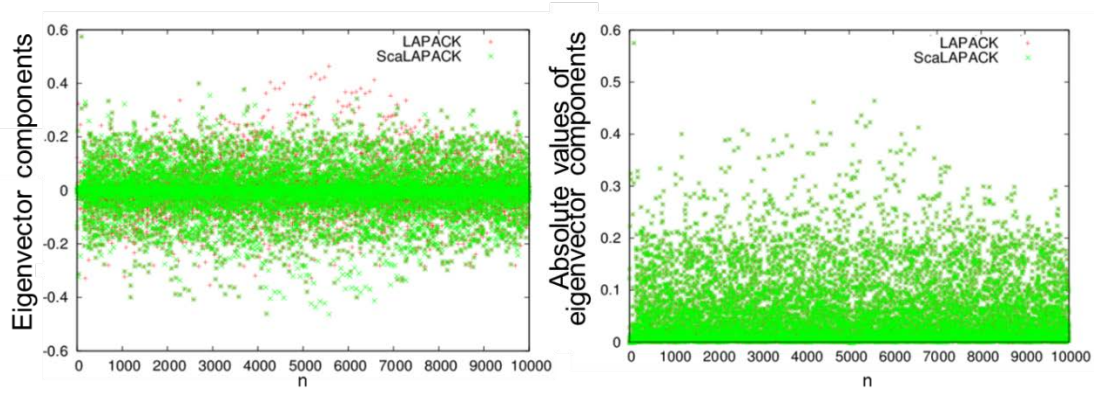

Fig. 44 Eigenvector components on the left, and their absolute values on the right, from the sequential LAPACK routine *dsygv* (red points) and the parallel ScaLAPACK routine *PDSYGVX* (green points).

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. 43 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.

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. 44 on the left). They are still correct eigenvectors as can be seen in Fig. 44 on the right, where the absolute values of all eigenvector components are shown. However, sometimes eigenvectors have even different values of their components. Such behavior 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}.

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. 45 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.

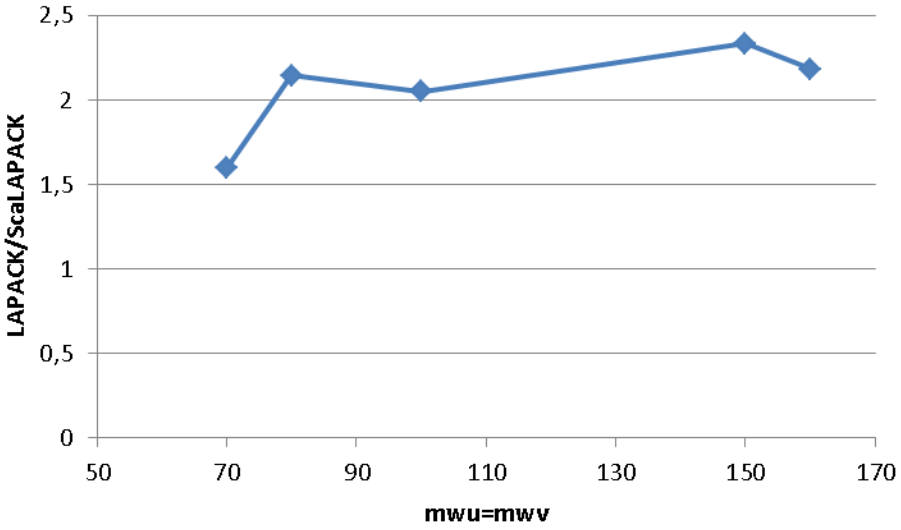

Fig. 45 Comparison of the eigenvalue solver execution time between ScaLAPACK using one process and the LAPACK library for different problem sizes.

The parallelization efficiency of the *PDSYGVX* subroutine is shown in Fig. 46 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.

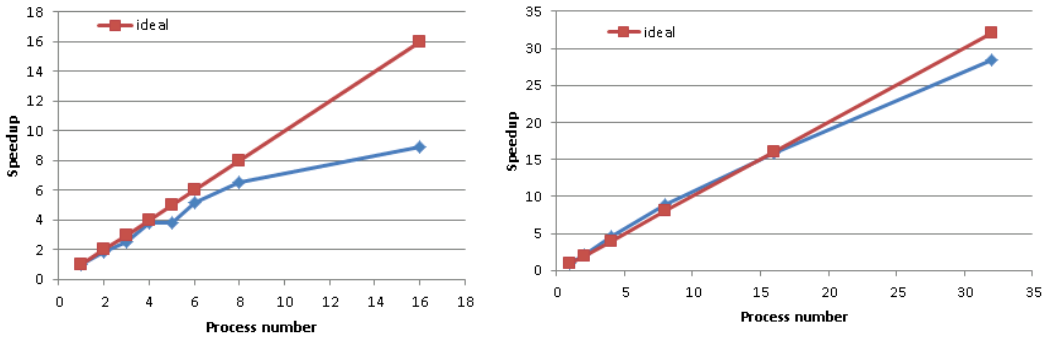

Fig. 46 *PDSYGVX* parallelization efficiency. On the left, small problem size with *nwu*=*nwv*=70*;* on the right, large problem size *nwu*=*nwv*=160*.*

6.2.2. **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 ~62.5 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 ~250 GB 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. 47. 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.*

```
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. 47 Original kernel loop that builds the matrix *a_ww*.

We tried to find some patterns between the *a_ww* and *dima* matrices such 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. 48 left). Each pink rectangle represents the global *a_ww* matrix, and the yellow rectangles the sub-matrices be 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. 48 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*.

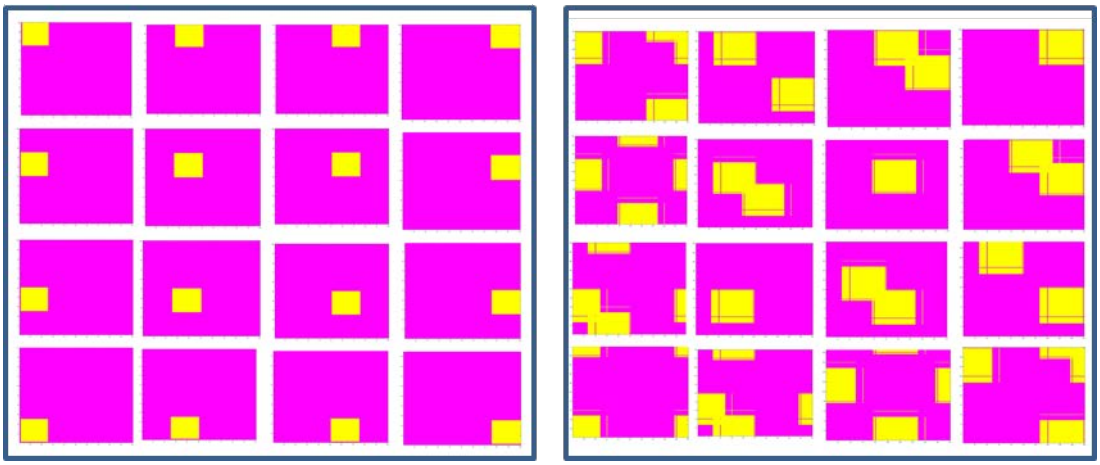

Fig. 48 Distributed matrix *a_ww* on 16 processors (left) and the corresponding indexes of the matrix *dima* that are used to calculate the local part of *a_ww* (right).

6.2.2.1. *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. 47), where it has already four nested loops, computation 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 computation 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. 49. 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 computation time.

```
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. 49 Matrix *dima* free kernel loop that builds the matrix *a_ww*.

Fig. 50 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 computation time increases in about two times for all problem sizes. For a large production run with *n_tri_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.

The next step was to check the parallelization efficiency of the kernel loop. This test is shown in Fig. 51. One can see that a speed-up factor of ~110 can be reached

when 256 tasks are involved. Therefore, the computation time of the kernel loop without the *dima* matrix using 256 cores would be about one hour.

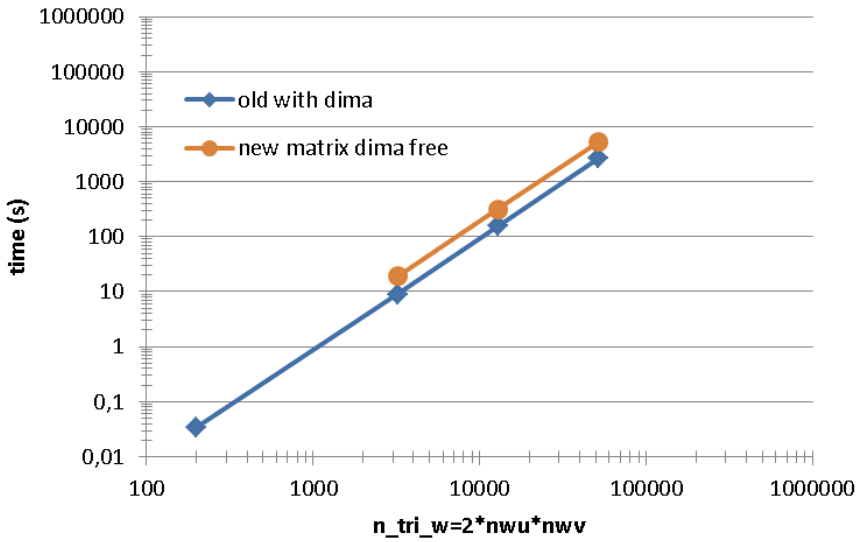

Fig. 50 Computational time of the kernel loop of the subroutine *matrix_ww* versus the problem size using the old code version (with *dima* matrix) – blue line and modified kernel loop (with *dima* free format) – orange line.

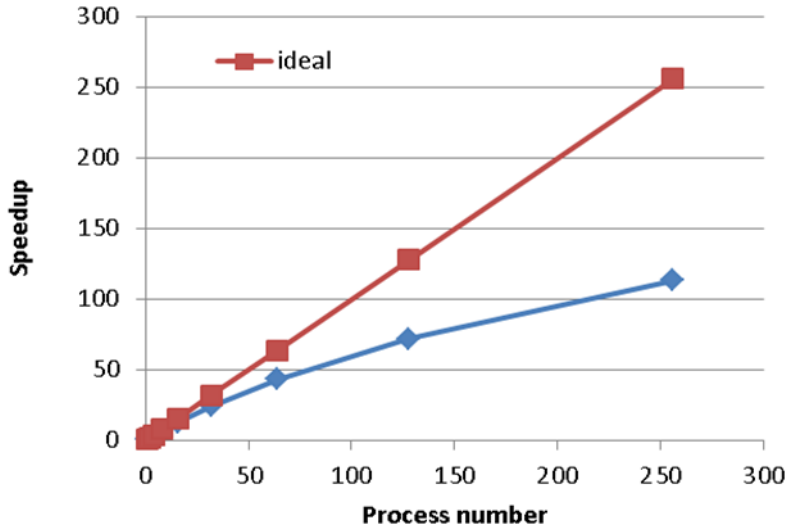

Fig. 51 Speed-up of the kernel loop versus number of MPI tasks.

6.2.2.2. *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. 52 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. 52 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. 49. Therefore, this kernel loop was rewritten one more time to ensure good scalability with the ScaLAPACK mapping scheme (Fig. 53). 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. 53 ScaLAPACK index mapping *dima* free kernel loop that builds the matrix *a_ww*.

6.2.3. **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*=200 000 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. 54.

The scalability of this kernel loop, depicted in Fig. 54, is shown in Fig. 55. 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 20 minutes.

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

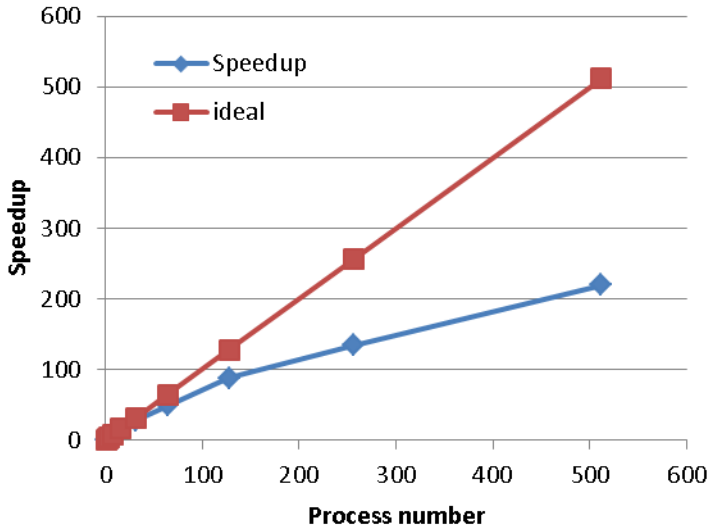

Fig. 55 Speed-up of the kernel loop in the *matrix_pp* subroutine versus number of MPI tasks.

6.3. *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 computational time for different subroutines.

Six sequential LAPACK subroutines were analyzed to be replaced by their parallel analogues from the ScaLAPACK library. All these subroutines must be replaced in the final code version because of the large input matrices size.

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 RZG where most of the calculations were performed.

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 eigenvalues. In addition, the correctness of the results was proven by the 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* and *tri_induct* were re-written in order to avoid of the largest matrix in the code named *dima*. This allows to save significant fraction of the memory that will bring the possibility to perform calculations for large problem sizes. The subroutines were parallelized by MPI taking into account the output index format for matrices which is necessary for ScaLAPACK subroutines. A good scalability was achieved in both subroutines with a speed-up factor of more than 210 when 512 cores were involved in the computation.

6.4. *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

7. **Final report on the 3DPSOLV project**

7.1. ***Introduction***

Three dimensional kinetic modelling of the plasma edge is one of the most ambitious topics in numerical plasma study. It is required for performing ab initio modelling of the linear plasma devices and divertor plasmas, for answering of a number of long-standing questions on the role of drifts in the plasma wall transition layer (PWT) and for formulating reliable multi-dimensional boundary conditions in the PWT. This simulation study will contribute to better understanding of plasma dynamics in linear plasma devices and in divertor plasmas of large fusion devices, to precisely estimate particle and power fluxes to the plasma facing components and to improving plasma edge simulating codes requiring boundary conditions in the PWT.

The BIT2 code is a code for Scrape-off-Layer (SOL) simulations in 2-dim real space + 3-dim velocity space. Due to the enhanced 2D geometry, the modeling of the SOL is more realistic than before. This is a requirement for the prediction of particle and energy loads to the plasma facing components (PFC), for the estimation of corresponding PFC erosion rates and impurity and dust generation rates. It is mandatory that the Poisson solver used in BIT2 scales to very high core numbers in order to maintain a good scaling property of the whole code.

The 3D massively parallel Particle in Cell and Monte Carlo code BIT3 represents a generalization of the BIT2 code to three dimensions. Its simulation geometry of elongated rectangular parallelepipeds allows a 1D domain decomposition for its parallelization. The extension of BIT2 routines to 3D is a more or less straightforward task, with the exception of the 3D massively parallel Poisson solver.

The aim of the project is to develop a 3D Poisson solver based on experience gained in developing the corresponding 2D solver in BIT2. The simulation domain of BIT2 is very thin (usually with an aspect ratio of (L_x/L_y) > 4096) as shown in Fig. 56. Its domain is distributed along the *x*-direction for parallelization. In this one-dimensional distribution, each MPI task handles a square domain with the same length and mesh size in each direction, i.e., the local aspect ratio is the same for every MPI task. This distribution allows to reduce the complexity of the code and to optimize the data communication, but restricts the number of MPI tasks in the current BIT2 code structure (each MPI task handles a square domain with a square mesh).

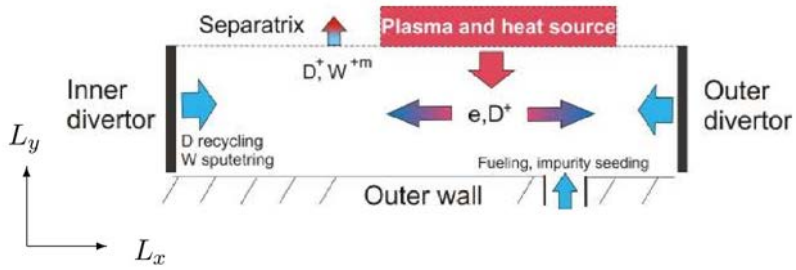

Fig. 56 Simulation domain of the BIT2 code.

The new solver will be based on the multigrid method using a mixed domain decomposition/finite-element-discretization. The optimized 2D solver achieved good scaling properties for up to a few thousands of cores. Contrary to the 2D version, the 3D solver is to be applied to a more simplified geometry not containing an inner empty space. Therefore, the expected scaling of the 3D solver should exceed the scaling of the 2D solver. This will allow to keep the overall scalability of the BIT3 code well above 1000 cores, which is required for ab initio modelling of linear plasma devices.

7.2. *Discretization for 3D problem*

To develop a 3-dimensional multigrid solver for second order partial differential equations (PDEs), we consider the parallelepiped domain Ω given by with L_x, L_y, and L_z shown in Fig. 57. In particular, we will focus on the second order PDE

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

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

To develop a 3D multigrid solver a general parallelpiped domain is discretized with a mesh size of $h_x= L_x/n_x$, $h_y = L_y/n_y$, and $h_z=L_z/n_z$ as shown in Fig. 58. The discretized function ϕ is defined on $P(x_{j,i,k}) = P(x_j,y_i,z_k)$ for $x_j = jh_x$, $y_i=ih_y$, $z_k = kh_z$, $0 \le j \le n_x$, $0 \le i \le n_y$, and $0 \le k \le n_z$ as shown in Fig. 58.

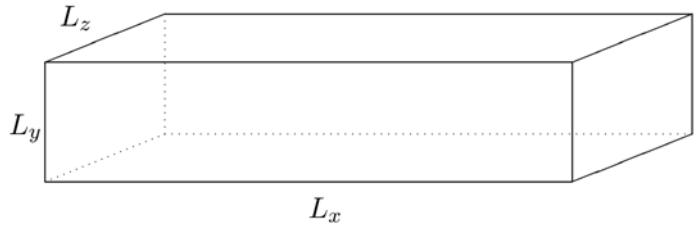

Fig. 57 Simulation domain of the BIT3 code.

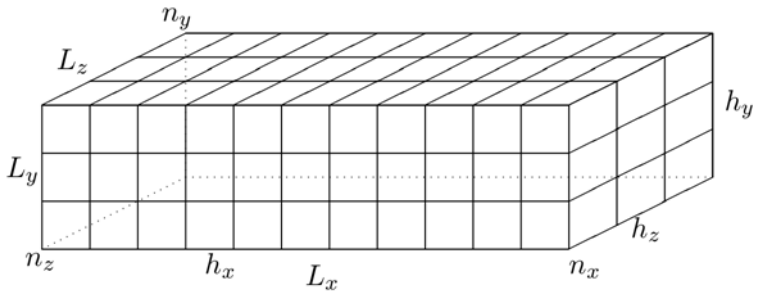

Fig. 58 Discretized domain of the BIT3 code.

We assume that $\varepsilon(x,y,z)$ is defined on each cell (x_j,y_i,z_k) which has its center point at $((j+\frac{1}{2})h_x,(i+\frac{1}{2})h_y,(k+\frac{1}{2})h_z)$ for $0 \le j \le n_x-1$, $0 \le l \le n_y-1$, and $0 \le k \le n_z-1$, in a similar way to the previous 2D case. This shall be denoted by $\varepsilon_{j+\frac{1}{2},i+\frac{1}{2},k+\frac{1}{2}}$.

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

7.2.1. **Finite difference discretization**

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

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

And the according finite difference formulations of the second order derivatives

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

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

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

where

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

and $\boldsymbol{\varepsilon}_{j\pm\frac{1}{2},i\pm\frac{1}{2},k\pm\frac{1}{2}}$ are the dielectric constants defined at the cell center as show in Fig. 59.

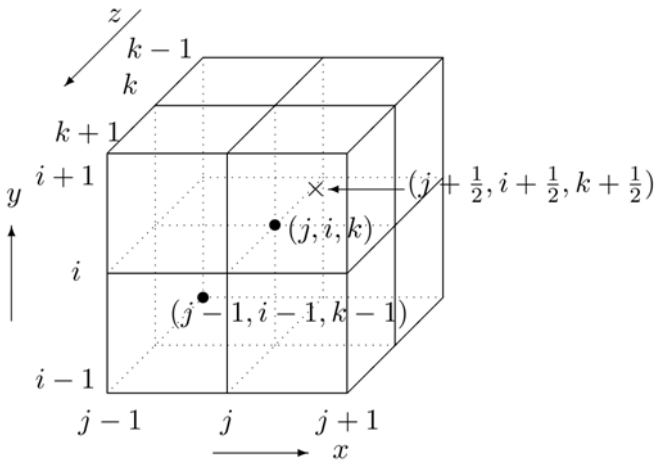

Fig. 59 Local notation for discretization of 3D domain

By summing these equations, we have

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

Concerning the Dirichlet boundary condition on the boundaries

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

including the internal conductor area, the values of the function are constants, i.e.,

$$\phi_{i,j,k} = \phi_c(\mathbf{x}_{i,j,k}).$$

On the boundary points with the Neumann boundary conditions, we use the central difference scheme (see Fig. 60) with ghost points at the outside of the domain.

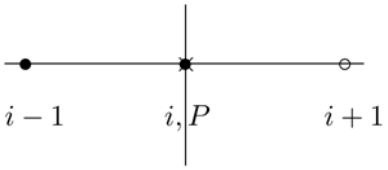

Fig. 60 Notation for the finite difference scheme on the Neumann boundary condition within the x+ direction (• : real nodal points, × : boundary point, ○ : ghost points)

For example, on the boundary with

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

we have

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

i.e.,

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

Finally, we get the discretization system

$$Ax = b.$$

The linear restriction operator is defined by

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

and the linear prolongation operator is defined by

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

according to the notations in Fig. 61.

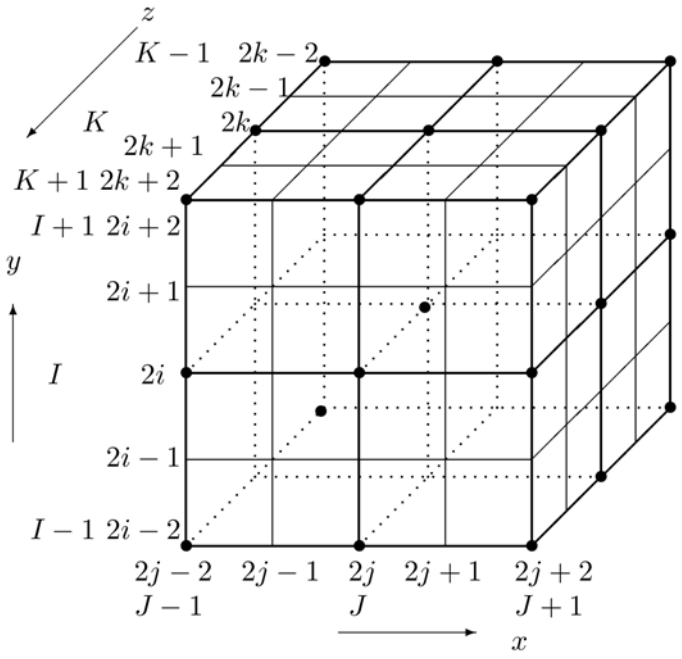

Fig. 61 Local notation for the intergrid transfer operators of the 3D problem (• : coarser node points)

The intergrid transfer operators are defined on the boundary in accordance with the boundary condition. The values of the function on the Dirichlet boundaries are all zero because we solve a problem with a zero-Dirichlet boundary condition except on the finest level. The values of the function on the Neumann boundary are defined in the same way as in case of a finite different schemes with a zero-Nuemann boundaries condition, i.e., the values at the ghost points are the same as the reflective real points

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

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

7.2.2. **Finite element discretization**

To define a finite element scheme, we consider the weak formulation of the problem described in Sec. 7.2.1. By multiplying with the test function $\psi(\mathbf{x})$ and integrating on both sides, we get

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

By using Green's theorem, we get

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

i.e.,

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

We consider the piecewise trilinear element space which is continuous and trilinear polynomial spaces on each rectangular parallelepiped (Fig. 62) with a finite element bases $\phi_{p,q} = \delta_{p,q}$.

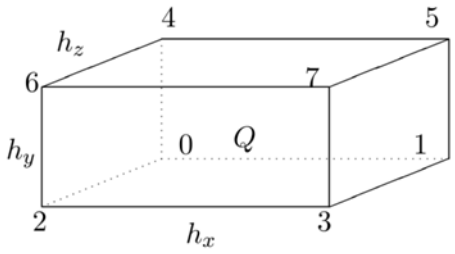

Fig. 62 A rectangular parallelepiped element Q for the FEM of a 3D problem.

First of all, we compute the integrations on a retangular parallelopiped as shown in Fig. 62 with each basis functions:

$$a_{p,p} = \frac{\epsilon_Q}{9}\left[\frac{h_x h_z}{h_y} + \frac{h_x h_y}{h_z} + \frac{h_y h_z}{h_x}\right], \quad p = 0,\ldots,7$$
$$a_{p,q} = \frac{\epsilon_Q}{18}\left[\frac{h_x h_z}{h_y} + \frac{h_x h_y}{h_z} - \frac{2h_y h_z}{h_x}\right], \quad (p,q) = (0,1),(2,3),(4,5),(6,7)$$
$$a_{p,q} = \frac{\epsilon_Q}{18}\left[\frac{h_y h_z}{h_x} + \frac{h_x h_y}{h_z} - \frac{2h_x h_z}{h_z}\right], \quad (p,q) = (0,2),(1,3),(4,6),(5,7)$$
$$a_{p,q} = \frac{\epsilon_Q}{18}\left[\frac{h_x h_z}{h_y} + \frac{h_y h_z}{h_x} - \frac{2h_x h_y}{h_z}\right], \quad (p,q) = (0,4),(1,5),(2,6),(3,7)$$
$$a_{p,q} = \frac{\epsilon_Q}{36}\left[\frac{h_x h_y}{h_z} - \frac{2h_x h_z}{h_y} - \frac{2h_y h_z}{h_x}\right], \quad (p,q) = (0,3),(1,2),(4,7),(5,6)$$
$$a_{p,q} = \frac{\epsilon_Q}{36}\left[\frac{h_x h_z}{h_y} - \frac{2h_x h_y}{h_z} - \frac{2h_y h_z}{h_x}\right], \quad (p,q) = (0,5),(1,4),(2,7),(3,6)$$
$$a_{p,q} = \frac{\epsilon_Q}{36}\left[\frac{h_y h_z}{h_x} - \frac{2h_x h_y}{h_z} - \frac{2h_x h_z}{h_y}\right], \quad (p,q) = (0,6),(1,7),(2,4),(3,7)$$
$$a_{p,q} = \frac{\epsilon_Q}{36}\left[-\frac{h_x h_y}{h_z} - \frac{h_x h_z}{h_y} - \frac{h_y h_z}{h_x}\right], \quad (p,q) = (0,7),(1,6),(2,5),(3,4)$$

To generate a matrix equation, we assemble all computations on all retangular papallelpiped elements and get a discretized linear system

$$Ax = b.$$

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

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

7.3. *Conclusions and outlook*

Kab Seok Kang has visited the project coordinator to discuss the current status of the BIT2 code and the future plans. The project coordinator is currently testing the BIT2 code with the multigrid solver which was developed in the last project KinSOL2D-3. It

was especially designed to suit the current BIT2 code structure. Kab Seok Kang will further support the project coordinator to bring BIT2 in its multigrid version to production stage.

The implementation of the 3-dimensional multigrid solver with the finite difference and finite element method is at the moment at the stage of building up the according matrix equation and intergrid transfer operators. However, the project duration comes to an end and the work has to stop here. A succeeding project will be needed to finish the implementation and testing of the 3D multigrid solver in BIT3. This is not critical as the project coordinator is still busy with testing BIT2 with its 2D multigrid solver. In parallel, he will further develop the BIT3 version. Therefore, he did not apply for a new HLST project in the support period of 2016 to continue the work on the 3D multigrid solver. Fortunately, the MGBOUT3D project in 2016 is also about an implementation of a 3D multigrid solver. Thus, the work will continue in a more global context with clear synergetic effects for a future BIT3 multigrid project to be probably submitted in 2017.

8. **Final report on the MGBOUT project**

8.1. ***Introduction***

The aim of this project is to implement in the BOUT++ code multigrid techniques to improve its performance and to prepare for new physical studies. BOUT++ is a flexible, modular framework to solve plasma physics relevant differential equations. It includes problem specific differential operators, boundary conditions and geometry. The CCFE version is used to study turbulence and coherent structure motions at the edge of magnetic fusion machines.

For the 2D Laplacian inversion, one of the implementations in BOUT++ uses the PETSc library to solve the problem iteratively (GMRES with restarts by default, but other algorithms are available). Without good preconditioning this is a very inefficient method which sometimes does not converge at all. At the moment the best option is a preconditioner based on a Fourier-decomposed solver. Unfortunately using Fourier transforms is an obstacle for parallelising in the toroidal direction and is less efficient as a preconditioner when the coefficients vary strongly in the toroidal direction. Moreover, Fourier solvers are based on a direct (parallel Thomas algorithm) inversion, which limits scaling performance. Multigrid preconditioning is an attractive alternative that should avoid these drawbacks.

Due to the modular structure of BOUT++, the PETSc-based Laplace solver can be modified without affecting the rest of the code (only ~1 000 lines out of ~85 000 needed to be looked at). The current implementation constructs explicitly the matrix to be inverted. While in the long term it is probably desirable to move to matrix-free methods, the current method has the advantage that relatively straightforward modifications of the existing code will allow matrices corresponding to coarser grids to be constructed within the solver class (independently of the main code) and the fields on the coarse grids could naturally be stored as a few extra PETSc vectors. Hence, the implementation of a prototype multigrid solver should be straightforward, if not perfectly elegant, in the short term.

In the long term, it is planned to upgrade the code, by refactoring the mesh implementation, to be able to use multigrid methods more widely. Application of 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 (RHS) evaluations needed in a fixed amount of simulation time, which should decrease in direct proportion the run-time. Multigrid methods should enable a scalable, iterative electrostatic potential solver to be implemented which would allow 3D-coupled equations to be solved (a perpendicular Laplacian, but in strongly non-orthogonal coordinates so that the differential operator contains first order derivatives in three dimensions, but second order derivatives in only two). This is necessary for accurate solutions of the electrostatic potential near X-points and therefore of fundamental importance for scape-off layer (SOL) physics.

The redesign process with an efficient multigrid implementation should identify things like: appropriate representation of the fields in memory; methods to iterate over and operate on fields; optimizing the topology of distribution of the grid(s) across processes to minimize communication overhead; how to maintain the flexibility to alter easily the physics model (i.e. the differential equations to be solved).

Underlying to all this work and as a prerequisite to a fruitful collaboration, training activities are necessary in order to allow CCFE staff to gain familiarity with multigrid techniques and HLST staff to learn the structure and the functionalities of BOUT++.

Under the above long term plan, we started with training activities and Kab Seok Kang implemented the multigrid solver under the current structure of BOUT++ and investigated the numerical performance on HELIOS. In this report, we summarize the structure of BOUT++ which is related to the 2D Laplacian inversion, the implementation of the multigrid solver, and the numerical performance results.

8.2. *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 data structures of BOUT++ which are related to the second order partial differential equation (PDE) on 2D for 3D problems.

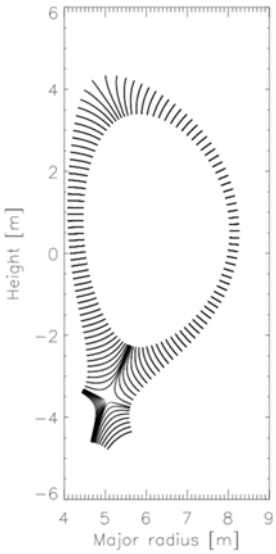

Fig. 63 *The coordinates in the poloidal plane of a tokamak in the BOUT++ code. The x and y coordinates lie in the poloidal plane. The solid lines denote x coordinate lines.*

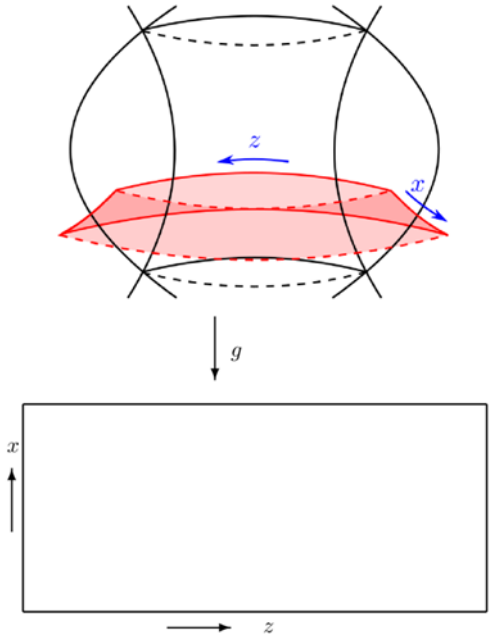

Fig. 64 Computational domain and its conformal mapping

To solve 3D tokamak shaped problems in BOUT++, the domain is sliced in the poloidal direction (perpendicular to the *y*-direction) and the second order PDEs are solved on each slice (*x*-*z* space) as shown in Fig. 63. Each slice can be transformed to the reference domain (rectangular or unit square) with periodic boundary condition along the *z*-direction by the conformal mapping *g* as shown in Fig. 64.

We consider the following second order PDE

$$d\nabla \cdot \nabla_\perp u + \frac{1}{c_1}\nabla c_2 \cdot \nabla_\perp u + au = f \tag{2.1}$$

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 poloidal fields are perpendicular to the toroidal field, the components in red can be neglected and we can solve the represented 2D problems.

To get the discretized solution of (2.1), BOUT++ uses cell centered second order finite difference schemes on the reference domain (Fig. 65):

$$\begin{aligned}
\left(\frac{\partial^2 u}{\partial x^2}\right)_{ik} &= \frac{u_{i-1,k} - 2u_{i,k} + u_{i+1,k}}{dx^2} \\
\left(\frac{\partial^2 u}{\partial z^2}\right)_{ik} &= \frac{u_{i,k-1} - 2u_{i,k} + u_{i,k+1}}{dz^2} \\
\left(\frac{\partial^2 u}{\partial x \partial z}\right)_{ik} &= \frac{u_{i-1,k-1} - u_{i-1,k+1} - u_{i+1,k-1} + u_{i+1,k+1}}{4dxdz} \\
\left(\frac{\partial u}{\partial x}\right)_{ik} &= \frac{-u_{i-1,k} + u_{i+1,k}}{2dx} \\
\left(\frac{\partial u}{\partial z}\right)_{ik} &= \frac{-u_{i,k-1} + u_{i,k+1}}{2dz}.
\end{aligned}$$

By combining the computations, we have a linear finite system with the following entries in stencil notation

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

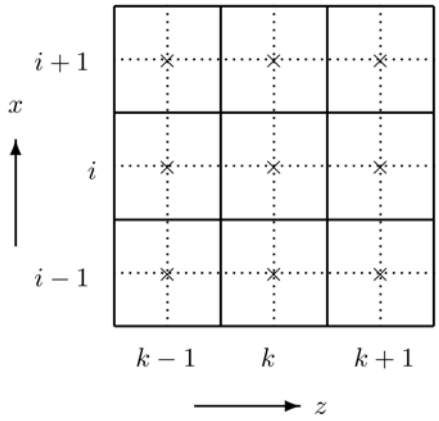

Fig. 65 Notation for discretization

With

$$
\begin{aligned}
a_{i-1,k+1} &= -\frac{d_{ik} g_{ik}^{xz}}{2\Delta x \Delta z} \\
a_{i,k+1} &= d_{ik}\left(\frac{g_{ik}^{zz}}{(\Delta z)^2} + \frac{G_{ik}^{z}}{2\Delta z}\right) + \frac{1}{c_1}\left(\frac{g_{ik}^{xz} c_{2x} + g_{ik}^{zz} c_{2z}}{2\Delta z}\right) \\
a_{i+1,k+1} &= \frac{d_{ik} g_{ik}^{xz}}{2\Delta x \Delta z} \\
a_{i-1,k} &= d_{ik}\left(\frac{g_{ik}^{xx}}{(\Delta x)^2} - \frac{G_{ik}^{x}}{2\Delta x}\right) - \frac{1}{c_1}\left(\frac{g_{ik}^{xx} c_{2x} + g_{ik}^{zx} c_{2z}}{2\Delta x}\right) \\
a_{i,k} &= -2d_{ik}\left(\frac{g_{ik}^{xx}}{(\Delta x)^2} + \frac{g_{ik}^{zz}}{(\Delta z)^2}\right) + a_{ik} \\
a_{i+1,k} &= d_{ik}\left(\frac{g_{ik}^{xx}}{(\Delta x)^2} + \frac{G_{ik}^{x}}{2\Delta x}\right) + \frac{1}{c_1}\left(\frac{g_{ik}^{xx} c_{2x} + g_{ik}^{zx} c_{2z}}{2\Delta x}\right) \\
a_{i+1,k-1} &= -\frac{d_{ik} g_{ik}^{xz}}{2\Delta x \Delta z} \\
a_{i,k-1} &= d_{ik}\left(\frac{g_{ik}^{zz}}{(\Delta z)^2} - \frac{G_{ik}^{z}}{2\Delta z}\right) - \frac{1}{c_1}\left(\frac{g_{ik}^{xz} c_{2x} + g_{ik}^{zz} c_{2z}}{2\Delta z}\right) \\
a_{i-1,k-1} &= \frac{d_{ik} g_{ik}^{xz}}{2\Delta x \Delta z}
\end{aligned}
$$

where $G_{ik}^{x} = g^{ij}\Gamma_{ij}^{x}$, $G_{ik}^{z} = g^{ij}\Gamma_{ij}^{z}$, $c_{2x} = [(c_2)_{i+1,k} - (c_2)_{i-1,k}]/(2\Delta x)$, and $c_{2z} = [(c_2)_{i,k+1} - (c_2)_{i,k-1}]/(2\Delta z)$.

From now on, we consider the parallelization concept to solve (2.1) in BOUT++. The considered problems in BOUT++ have periodic boundary conditions in *z*-direction and Dirichlet or Neumann boundary conditions in *x*-direction and are discretized with $N_{x\_global} \times N_{z\_global}$ meshes as shown in Fig. 66.

The current BOUT++ is parallelized only in the *x*-direction as shown in Fig. 67 and needs to communicate data between neighboring MPI tasks in the *x*-direction. Even if it is not parallelized in the *z*-direction, it needs ghost cells at the ends of the *z*-direction of the domain due to the periodic boundary condition.

According to the above explained parallelization concept, we generate the discretized system of (2.1) for each MPI task.

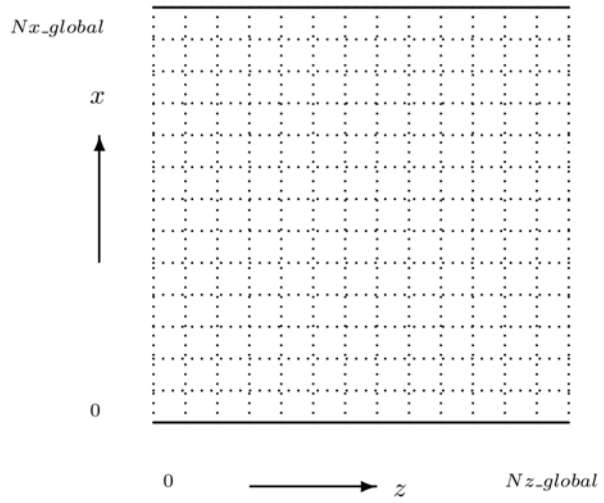

Fig. 66 Global numbering

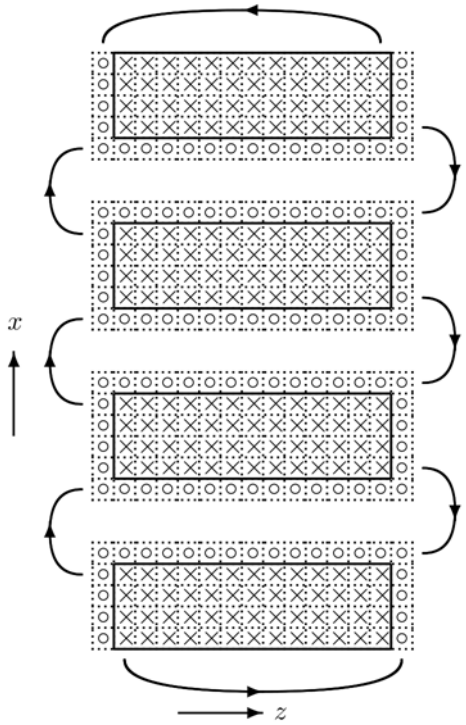

Fig. 67 One dimensional parallelization and its data communications

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

In this section, we explain how to implement multigrid methods to solve the discretized system of (2.1) in a general framework.

According to the object oriented framework of BOUT++, we define a C++ class to implement the basic multigrid algorithm for the parallel cell centered finite difference scheme on a rectangular domain. We assume that each MPI task handles the same number of mesh points in each direction. From this basic class, we derive a 1D parallel, a 2D parallel, and a serial version of the multigrid algorithm by assigning a number of MPI tasks in each direction.

8.3.1. **Basic class for multigrid algorithm in BOUT++**

Among the 1D parallel, the 2D parallel and the serial versions, the most general one is the 2D parallel version. So the basic class of the multigrid algorithm is defined on *xNP* × *zNP* sub-domains where each domain has *lnx* × *lnz* cells as shown in Fig. 68 (×). For data communication, we add guard (ghost) cells (○) at each end which are needed for the matrix-vector multiplication, smoothing operations, and intergrid transfer operators. We plot the data communication pattern in Fig. 69.

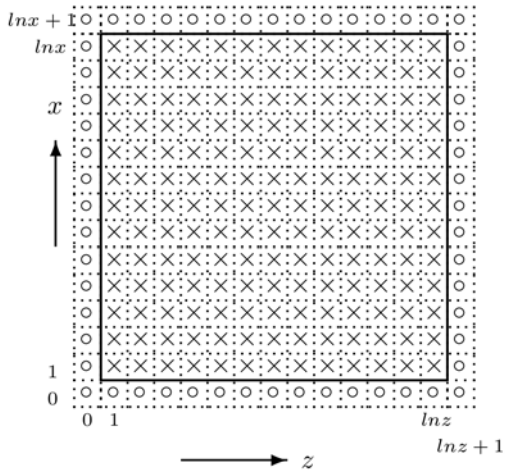

Fig. 68 Local numbering on each MPI task

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: $r_H = R r_h = R(f_h - A_h \bar{u}_h^{(k)})$ {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: $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + P e_H^{\mu}$ {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**

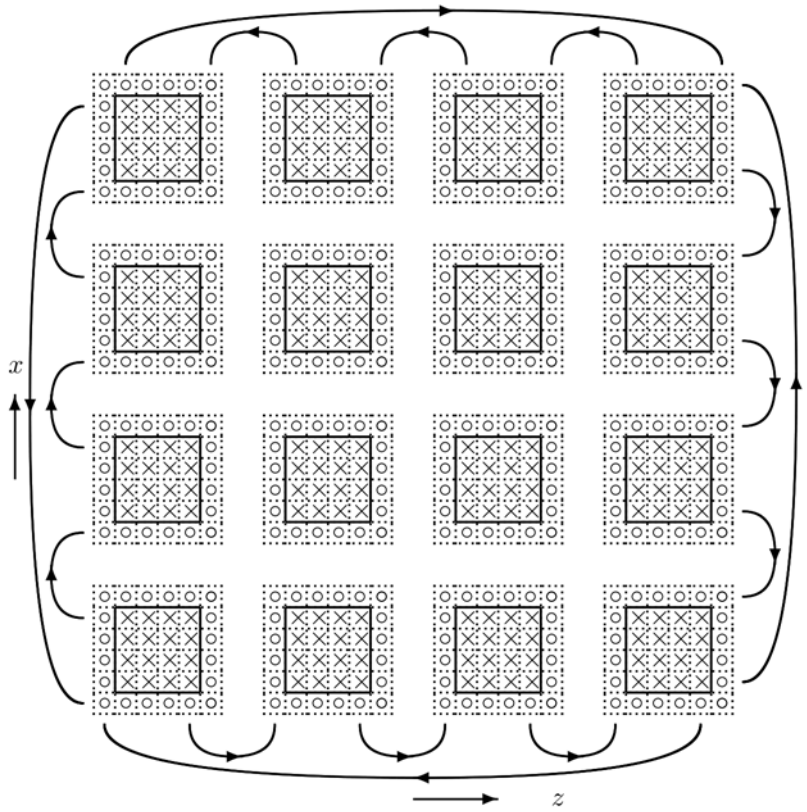

Fig. 69 Basic data communication for the parallel multigrid algorithm

```
class MultigridAlg{
public:
  MultigridAlg(int ,int ,int ,int ,int ,MPI_Comm ,int);
  ~MultigridAlg();
  void setMultigridC(int );
  void getSolution(BoutReal *,BoutReal *,int );
  void cleanMem();
  int mglevel,mgplag,cftype,mgsm,pcheck,xNP,zNP,rProcI;
  BoutReal rtol,atol,dtol,omega;
  int *gnx,*gnz,*lnx,*lnz;
  BoutReal **matmg;
protected:
  int numP,xProcI,zProcI,xProcP,xProcM,zProcP,zProcM;
  MPI_Comm commMG;
  void communications(BoutReal *, int );
  void setMatrixC(int );
  void cycleMG(int ,BoutReal *, BoutReal *);
  void smoothings(int , BoutReal *, BoutReal *);
  void projection(int , BoutReal *, BoutReal *);
  void prolongation(int ,BoutReal *, BoutReal *);
  void pGMRES(BoutReal *, BoutReal *, int , int);
  void solveMG(BoutReal *, BoutReal *, int );
  void multiAVec(int , BoutReal *, BoutReal *);
  void residualVec(int , BoutReal *, BoutReal *, BoutReal *);
  BoutReal vectorProd(int , BoutReal *, BoutReal *);
  virtual void lowestSolver(BoutReal *, BoutReal *, int );
};
```

Fig. 70 Class MultigridAlg

To implement the above algorithm in C++, we construct the class **MultigridAlg** as a basic class. We show the definition of **MultigridAlg** and the function **MultigridAlg::cycleMG()** which is the main multigrid algorithm in Fig. 70 and Fig. 71.

```
void MultigridAlg::cycleMG(int level,BoutReal *sol,
     BoutReal *rhs){
  if(level == 0) {
    lowestSolver(sol,rhs,0);
  }
  else {
    BoutReal *r,*pr,*y,*iy;
    r = new BoutReal[(lnx[level]+2)*(lnz[level]+2)];
    pr = new BoutReal[(lnx[level-1]+2)*(lnz[level-1]+2)];
    y = new BoutReal[(lnx[level-1]+2)*(lnz[level-1]+2)];
    iy = new BoutReal[(lnx[level]+2)*(lnz[level]+2)];
    smoothings(level,sol,rhs);
    residualVec(level,sol,rhs,r);
    projection(level,r,pr);
    for(int i=0;i<(lnx[level-1]+2)*(lnz[level-1]+2);i++)
      y[i] = 0.0;
    cycleMG(level-1,y,pr);
    prolongation(level-1,y,iy);
    for(int i=0;i<(lnx[level]+2)*(lnz[level]+2);i++)
       sol[i] += iy[i];
    smoothings(level,sol,rhs);
    delete [] iy;
    delete [] y;
    delete [] pr;
    delete [] r;
  }
  communications(sol,level);
  return;
}
```

Fig. 71 Function MultigridAlg::cycleMG()

The intergrid transfer operators, i.e., the restriction and prolongation operators, are one of the key factors in the multigrid algorithm and depend on the discretization scheme for the geometric multigrid method. 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 for one coarse cell. We can write according to the notation in Fig. 72

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

i.e., in stencil notation.

$$I_H^h = \left]\begin{matrix}1 & 1\\ 1 & 1\end{matrix}\right[, \qquad I_h^H = \begin{bmatrix}\frac{1}{4} & \frac{1}{4}\\ \frac{1}{4} & \frac{1}{4}\end{bmatrix}$$

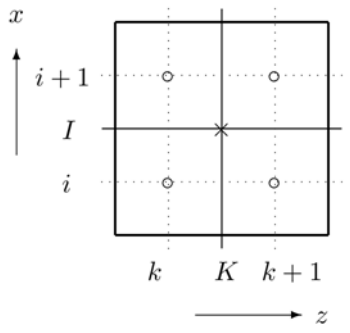

Fig. 72 Notation for the intergrid transfer operators

As shown in the **Multigrid Algorithm**, 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. To avoid computing the conformal mapping values on each coarser grid, we use an algebraic relation for the coarse and fine grid where *c* can be chosen to get good performance.

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

In this implementation, we choose c to make A_H the same operator as for the geometric version at the simplest problem. Usually the Krylov subspace method (CGM or GMRES) and direct methods can be used as coarsest level solver. These solvers are well known and are fast for small problems but are not scaling with respect to the number of degrees of freedom (DoF).

The smoothing iterations have 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 it does not depend on the number of MPI tasks. Therefore, we can use the damped Jacobi iteration for debugging purposes on parallel codes. In practical computing, we use the Gauss-Seidel iteration as a smoother. The multigrid method can be used as a solver and as a preconditioner in the Krylov subspace method. The multigrid method does not guarantee to converge as a solver for some cases, so we also consider the Krylov subspace method. For this purpose, we implemented the Preconditioned GMRES which guarantees the converge even for nonsymmetric problems.

Table 12 The required number of iterations for the PGMRES solver with a multigrid preconditioner (GMRES) and a multigrid solver (MG) according to the number of levels with a problem of 1024 × 1024 grid cells. (GS) or (J) denote what smoother are used, Gauss-Seidel (GS) or Jacobi (J) smoothers

Levels	DoF(C)	MG (GS)	GMRES(GS)	GMRES(J)
4	16 384	18	9	11
5	4 096	19	9	12
6	1 024	20	10	12
7	256	20	10	12
8	64	20	10	12

To look at the convergence of the multigrid method as a solver and as a preconditioner, we choose a problem with 1 024 × 1 024 grid cells and report the required numbers of iterations to achieve convergence in Table 12 according to the number of grid levels. In general we have very good convergence results. The required number of iterations in Table 12 hardly dependents on the number of levels which is the superior property of the multigrid method. However, in the case of a multigrid solver using a damped Jacobi smoother convergence could not be achieved.

8.3.2. **Multigrid solver according to BOUT++ 1D parallelization**

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. 73.

For the parallel multigrid method, the coarsest level has at least one real cell and the selection of the coarsest level is limited and the number of DoF on the coarsest level is greater than or equal to

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

where N_c is a number of MPI tasks. We show 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 13.

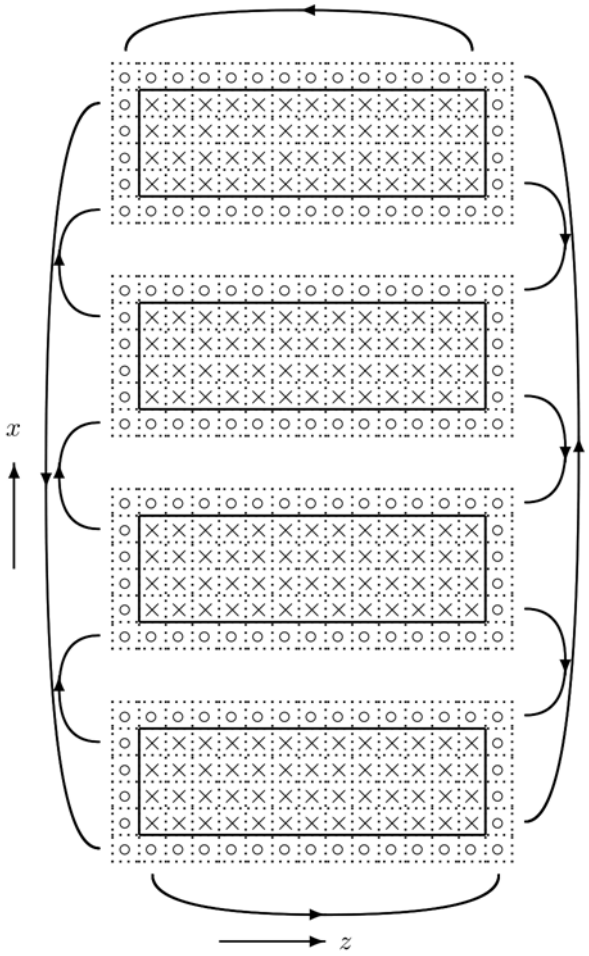

Fig. 73 One dimensional parallelization and its data communication for the multigrid algorithm

Table 13 The number of DoF on the coarsest level and the required number of iterations of GMRES

# MPI	1	2	4	8	16	32	64	128	256
# DoF	1	4	16	64	256	1 024	4 096	16 348	65 536
# GMRES	1	2	4	8	16	32	64	128	256

The number of iterations required in the GMRES solver increases with the square root of # DoF on the coarsest level and the # DoF increases with the square of the number of MPI tasks. From this limitation on the coarsest level, we have either to use a serial algorithm after gathering the data or use a 2D parallel algorithm to use more levels which have better scaling properties. 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 number of MPI tasks.

The desired algorithm is to use a 2D parallelization starting from the finest level, but it needs a 2D parallelization of the whole BOUT++ code which implies 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 (class **Multigrid1DP**).

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

8.3.3. **Multigrid solver with 2D parallelization and data gathering**

The basic multigrid class **MultigridAlg** is implemented based on a 2D parallelization. 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()**.

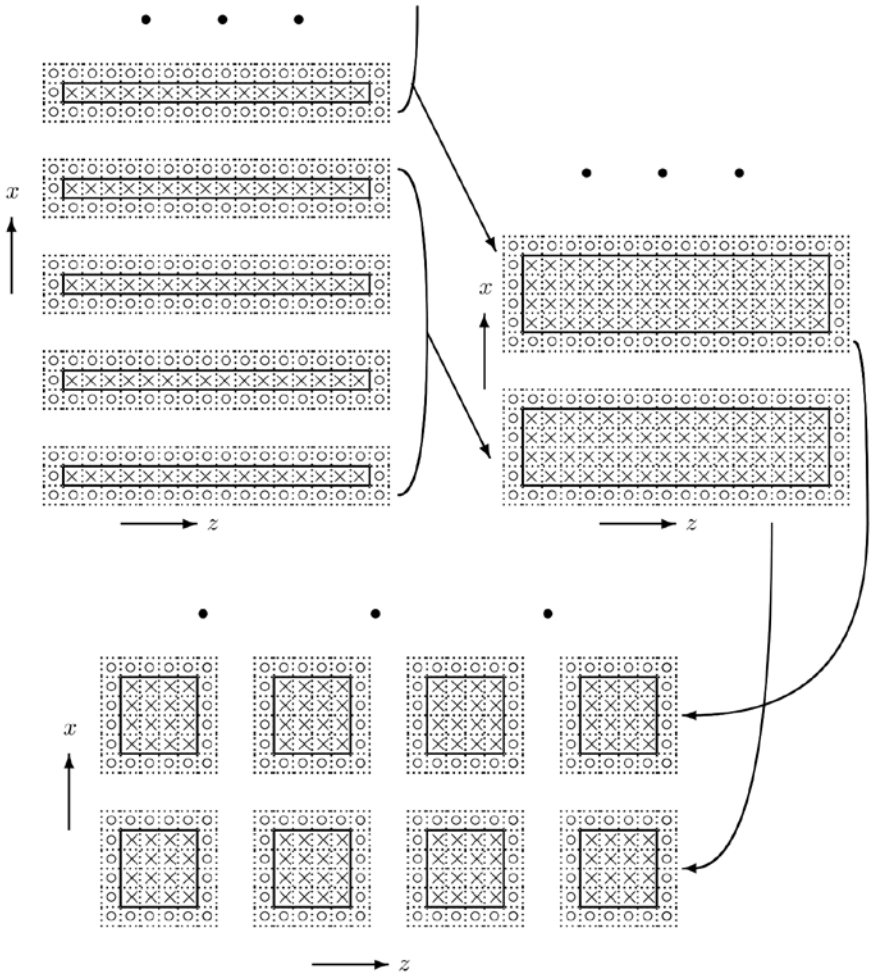

Fig. 74 Change from 1D parallelization to 2D parallelization

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. 74. 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 plot 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. 74.

To valuate the performance of the described algorithm, we measure two quantities: the time to solution and matrices setting time, which is the time that is needed to generate matrices on all levels. We consider the strong scaling of these quantities for a problem size of 4096^2 DoF. The corresponding results are shown in Fig. 75. First,

we use the 1D parallelization algorithm, denoted — and - - - (case one). In addition, the problem size on the lowest level is increased 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). At last, we use the 1D parallelization and the 2D parallelization algorithm, denoted by ○ ○ ○ and ◊ ◊ ◊ (case three).

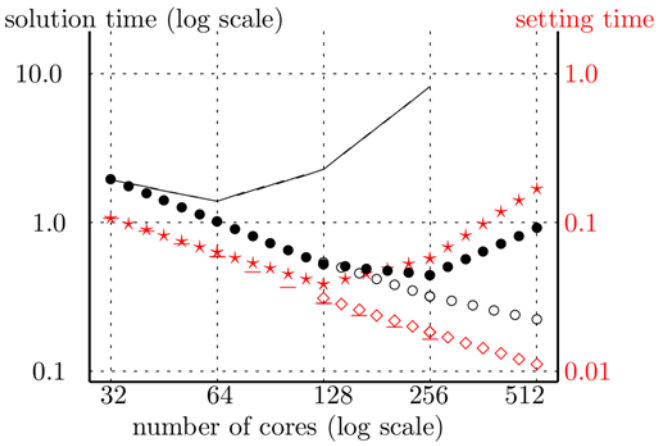

Fig. **75** The solution time in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver (in black) and matrices setting time (in red) as a function of the number of cores for a domain of 4096^2 DoF (DoF of the lowest level is 1024) with different parallel combinations.

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 of the lowest level is increased, 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. 75 —. As expected, the setting times (matrix building times) are reduced as the number of MPI tasks is increased (Fig. 75 - - -).

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 setting and 256 MPI tasks for 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. 75○ ○ ○ and ◊ ◊ ◊ .

From Fig. 75, 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 might be used on middle range numbers of MPI tasks.

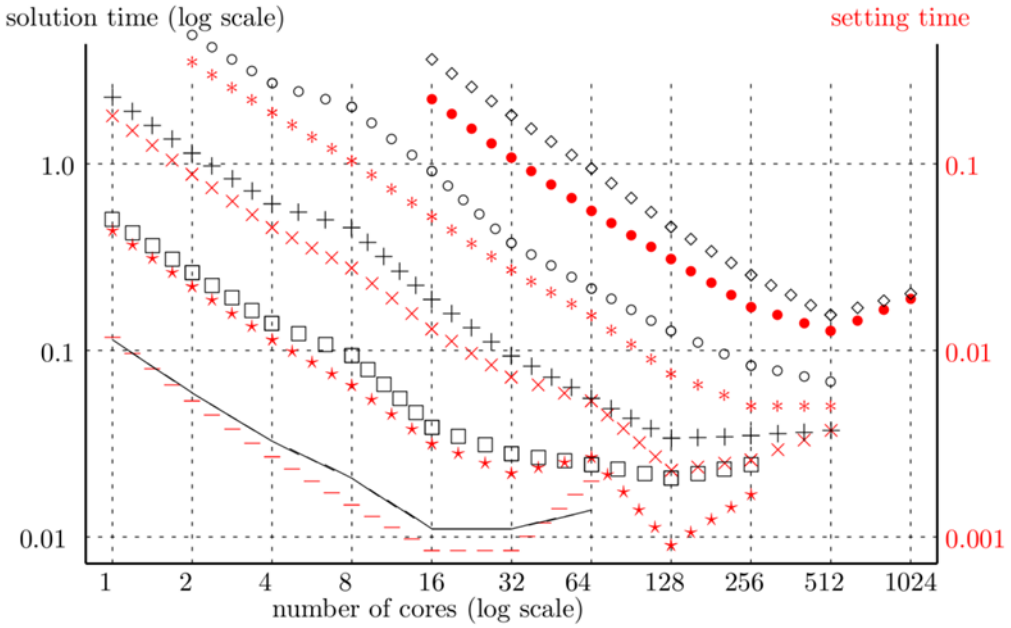

Fig. 76 (Strong scaling) The solution time (in black) and matrix setting time (in red) in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver as a function of the number of cores.

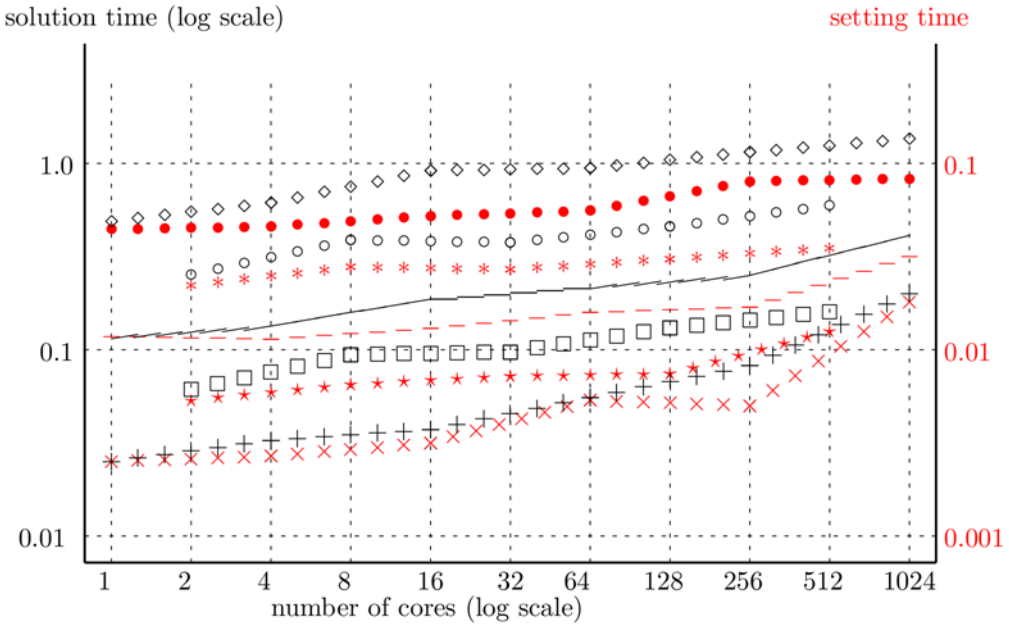

Fig. 77 (Weak scaling) The solution time (in black) and matrix setting time (in red) in seconds of the multigrid method with a Gauss-Seidel smoother as a preconditioner for the PGMRES solver as a function of the number of cores.

Next, we consider the strong and weak scaling properties of the third algorithm which uses a 2D parallelization 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. 76 and Fig. 77 we show the solution time in black and the matrix setting time in red. In Fig. 76 we consider a strong scaling for five cases with 256^2 DoF (—, - - -), 512^2 DoF (□, *), 1024^2 DoF(+, ×), 2048^2 DoF(○,✳), and 4096^2 DoF(◊,•) on the finest level. For each case, the solution time and setting time have a good strong scaling property up to 512 MPI tasks. In Fig. 77 we consider a weak scaling for five cases with 16384 DoF(+, ×), 32768 DoF (□, *), 65536 DoF (—, - - -), 131072 DoF (○, ✳), 262144 DoF (◊, •) per MPI task on the finest level. Over

all cases we have a good weak scaling property, especially, for large numbers of DoF per MPI task which is a typical property of the parallel multigrid algorithm.

8.4. *Conclusions*

At the start for the project, Kab Seok Kang visited the CCFE and gave lectures on basic numerical methods and multigrid methods to the BOUT++ team. During his visit, he attended a workshop on the BOUT++ code to learn the structure and the functionalities of BOUT++.

After this visit, Kab Seok Kang implemented a 1D parallel multigrid solver according to the BOUT++ structure in C++ and tested it on a problem which was provided by the BOUT++ team. The results of the 1D parallel multigrid solver showed that there was a need for a more enhanced parallel multigrid solver.

So the structure of the multigrid solver was adapted in the following way. Starting with the highest mutigrid level a 1D parallelization is used. From a certain level on the problem becomes too small for a 1D parallelization. Therefore, the data are redistributed and the parallelization concept is switched to a 2D parallelization of the domain. Finally, for the coarsest grid level a serial solver is being used. The numerical results showed that the new hybrid solver has a very good weak scaling property which enables the user to solve larger problems on a large number of MPI tasks in reasonable time scale. The implementation was sent to the project coordinator for testing within BOUT++.

Now, the BOUT++ team is considering to add a *z*-directional parallelization and to refactor the library. In an already allocated follow-up project Kab Seok Kang will give advice on the parallelization concept and modify the multigrid solver.

9. **Final report on HLST project TOPOX2**

9.1. ***Introduction***

Turbulence in fusion plasmas is known to be affected by the shaping of the magnetic flux surfaces. Modern tokamaks have strongly shaped diverted magnetic structures in which the last closed flux surface is a separatrix with an X-point. This leads to high magnetic shear near the X-point region, whose effect on drift-wave turbulence is believed to be severe. A complete numerical demonstration of this is yet to be made due to the outstanding challenge of resolving all the dynamically relevant spatial scales involved. Alternatives to the standard field-alignment techniques, upon which well resolved turbulence computations depend nowadays, are in need since they break down at the X-point. This constitutes the framework within which the project TOPOX2 was devised, namely the challenge of building a gyrofluid turbulence code capable of simulating from the magnetic axis to core to edge to scrape-off layer (SOL) including the divertor. The project TOPOX2 in particular is concerned with the extension to the open field-line region of a triangular grid in *RZ*-space, with the points arranged along flux surfaces which are topologically treated as hexagons. This grid is currently being used in the Grad-Shafranov equilibrium solver GKMHD, and the goal of the project is to continue its predecessor project (TOPOX) in order to generalize the scheme to make it work beyond the separatrix into the SOL, including the X-point. Therefore, the results of both projects are presented together, such to maintain the natural continuity between them. This constitutes one step towards the ultimate goal (beyond the scope of the project) of extending the code to treat global turbulence using novel treatments of the parallel derivatives with non-Clebsh coordinates.

9.2. ***GKMHD code***

One of the motivations to build the GKMHD code was to have a simplified (and extendable) test-bed for the geometrical techniques described before, which are needed for a future global turbulence code capable of resolving the whole plasma column, from the magnetic axis to the SOL in diverted magnetic structures, i.e. including the X-point(s). The GKMHD code is an axisymmetric Grad-Shafranov equilibrium solver. Its underlying model was derived consistently from the gyrokinetic theory, making easy its future extension to a full gyro-fluid turbulence model. The code uses a strongly structured triangular grid in *RZ*-space, with the points arranged along flux surfaces which are topologically treated as hexagons. This has the advantage over the more usual quadrilateral grid to eliminate grid singularities (at the magnetic axis) while keeping the poloidal spacing approximately constant on all flux surfaces. The Laplacian differential operators are expressed in the form of closed line integrals following Ref. [1]. A large part of the method's machinery consists of finding the node-index ordering and orientation of the nearest-neighbor hexagons.

9.3. ***Numerical method***

The idea is to write the standard Poisson equation in its weak form over a closed domain. Then, noting that the Laplacian operator on the left hand side (l.h.s.) can be expressed as the divergence of a gradient, one invokes the Gauss's divergence theorem to recast the volume integral into a surface integral over the enclosing surface. In two dimensions, this becomes a line integral, and if the domain, as represented in Fig. 78, is an infinitesimal hexagon, then it can be approximated with the sum

$$(\nabla^2 u)^j = \frac{1}{s_*^j} \sum_{m=1}^{6} \overline{Q_m^j Q_{m+1}^j} \frac{u_m^j - u_0^j}{\overline{P_0^j P_m^j}} \tag{1}$$

as was introduced by Sadourny, Arakawa and Mint in Ref. [1]. The six parcels correspond to the six nearest neighbors labelled by the local index *m*, where P_m^j are the grid-nodes and Q_m^j are the dual-space nodes (Voronoi diagram). On a surface discretized with a hexagonal mesh, this formula can be applied at every grid-node *j*.

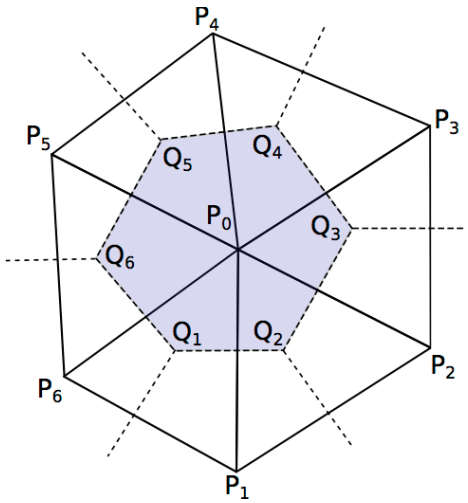

Fig. 78 Elemental hexagonal grid-mesh, showing the six nearest neighbors P_m^j of grid-node P_0^j and the corresponding six dual-space grid nodes Q_m^j, from which the Voronoi diagram element area $s_\star^j$ (blue) is calculated. The superscript *j* was dropped in the diagram.

9.3.1. **Closed flux surface region**

The global *j* index that labels each grid-node in the mesh is obtained by counting the grid-nodes from the centre (magnetic axis) following an outward spiral in the counter-clockwise direction. In practice, to obtain *j* for a particular node on flux surface *k* one just needs to start counting from the branch-cut (poloidally counter-clockwise) within that flux surface and then add the number of nodes belonging to the enclosed inner flux surfaces. The latter number is given by the formula 3*k*(*k*-1) noting that the origin (magnetic axis) is labelled as flux surface 1. For instance, for a case with four flux surfaces around the magnetic axis, this number is 3*4(4-1)=36, as shown in Fig. 79 in blue for the outmost vertices of the six main triangles.

As for the local orientation of the six nearest-neighbors of each grid-node, the *m*-index, the choice made sets the first neighbor to be the forward point (poloidally counter-clockwise) in the same flux surface, as illustrated by the numbering of the nodes highlighted in magenta in the three examples shown. This determines the column indexes of the Laplacian (sparse) matrix elements which arise when the set of discrete equations is written in matrix form. Further invoking flux conservation properties, it can be shown that this matrix is also symmetric. The solution can then be found using the solver method of choice. More details on this topic can be found in section 9.4.

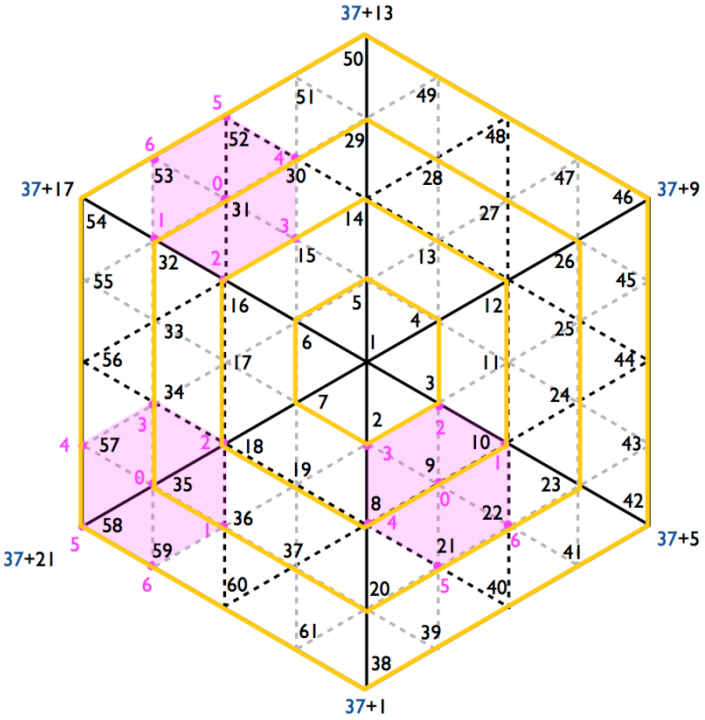

Fig. 79 Illustration of the hexagonal grid currently used in GKMHD (yellow flux surfaces). The global numbering is shown. The local number ordering and orientation of the nearest-neighbor hexagons is also shown with a few examples (magenta hexagons and numbers).

9.3.2. **Open flux surface region**

In terms of extending the method to go across a magnetic separatrix, whose domain encompasses the surfaces represented in red in Fig. 80, the modification proposed here is one which implies minimal changes to the method just described. The first step is to define the branch-cut to pass through the magnetic axis and the X-point (green dot symbol), and extend it to the private flux region, i.e. the region below the X-point. Then, an artificial boundary is set between the intersection point in the outermost considered flux surface in the private flux region and a location on each leg of the outermost SOL flux surface considered (yellow dot symbols), whose connection is depicted in cyan in Fig. 80. Everything outside this domain is discarded. This means that the divertor itself is not included, and will have to be modelled via physical boundary conditions (sheath models) on the regions corresponding to the cyan lines in future turbulence simulations based on this grid. On the other hand, this choice still keeps the necessary magnetic structure, namely the SOL region, the private flux region and the X-point while maintaining the hexagonal topology of the closed flux surface grid. Because the domain topology remains hexagonal, the numbering and ordering method described before can be directly applied to the domain on the open flux surface region (in red). An example of nearest neighbor orientation is also given in Fig. 80 for the SOL region (magenta). However, precisely at that grid node, where the flux surface bends towards the X-point, there might be a numerical issue that is not apparent from this schematic figure, but that can be already explained. Namely, the fact that the three neighbors (0,1,6) all lie in the same flux surface means that the elemental triangle yielded by them can be quite deformed in a realistic shaped flux surfaces, with one of their angles much larger than the other two. This is shown for a realistically shaped flux surfaces in sub-section 9.4.2.

The next step is to devise a test-case whose flux surfaces have the adequate topology. This is the subject covered in the remaining text, first restricted to the closed flux surface region, using a circular disk domain to check that the method is working properly and then generalize it to include the open flux surface region according to the scheme proposed here.

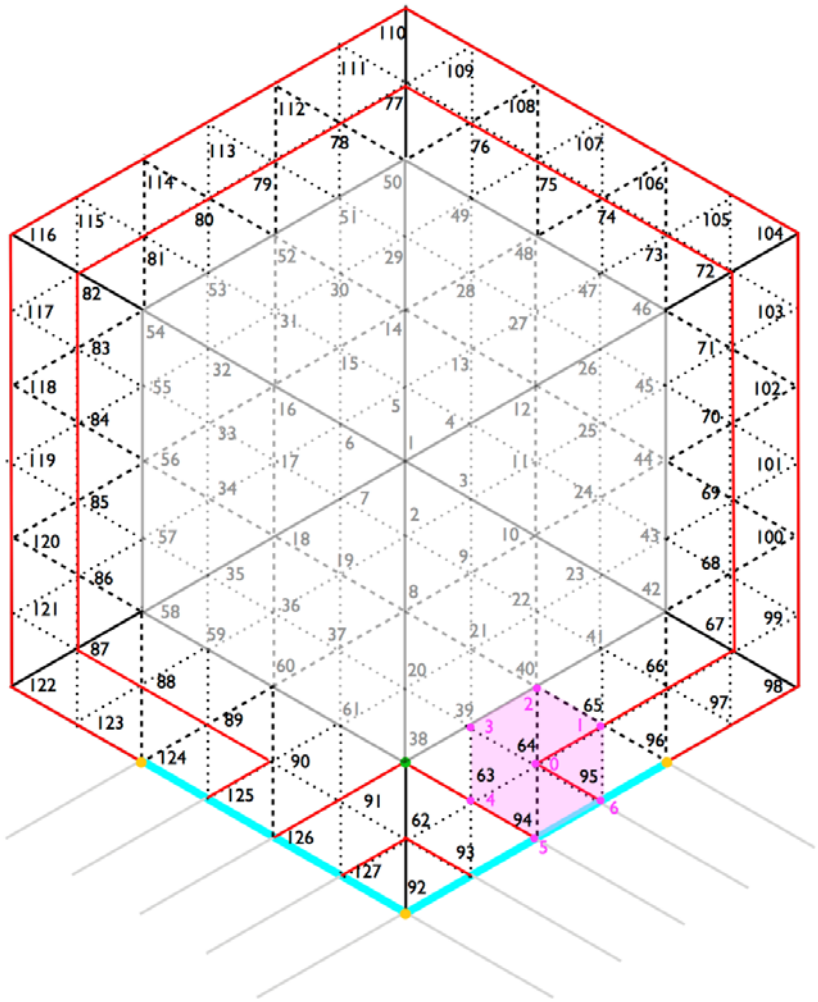

Fig. 80 Proposed extension of the grid indexing to the SOL region (red flux surfaces). The global numbering of the closed flux region (greyed out) is simply continued.

9.4. *Test-case*

In order to proceed with the modifications proposed in the previous section in a structured way, it is advantageous to devise a stand-alone test-case. This will ultimately provide a clean way to check such modifications before they are ready to be implemented in the GKMHD's solver. Additionally, devising a test-case is in practice an appropriate way to become familiarized with the numerical method at hand and its properties, without having to carry the additional overhead of the GKMHD code.

9.4.1. **Closed flux surfaces**

Starting with closed flux surfaces, which we already know how to implement, the test-case chosen consists of solving the Poisson equation on a circular disk, with radial Dirichlet boundary conditions. In polar coordinates (r,θ), it reads

$$\nabla^2 u(r,\theta) = \left(\frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} + \frac{1}{r^2}\frac{\partial^2}{\partial \theta^2} \right) u(r,\theta) = f(r,\theta) \tag{2}$$

with a source term given by [2]

$$f(r,\theta) = -4(p+1)r^p \cos(p\theta) \tag{3}$$

where p is an integer free parameter. This system has an analytical solution given by

$$v(r,\theta) = \left(1 - r^2\right) r^p \cos(p\theta) \tag{4}$$

Setting the free parameter to *p*=10 yields the functions depicted in Fig. 81.

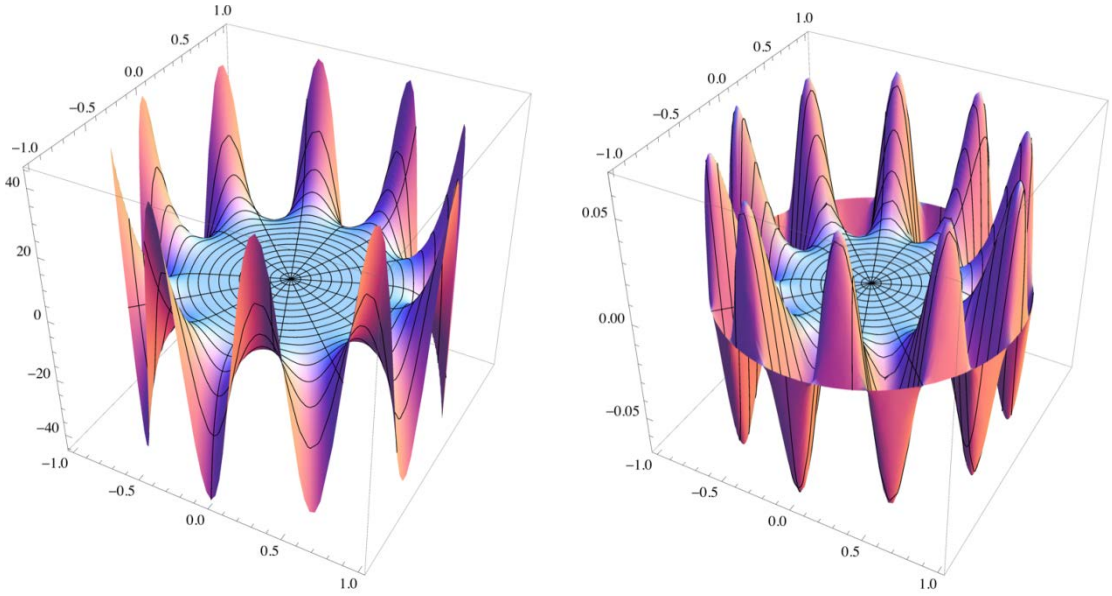

Fig. 81 Source term (3) on the left and corresponding inverse Poisson problem's analytical solution (4) on the right, for *p*=10.

The standard approach to address this problem would be to build a two-dimensional solver in polar coordinates. This was done in a stand-alone test-case code, where a pseudo-spectral method based on the work by Lai [3] was implemented. It uses central second-order finite-differences in the radial direction and truncated Fourier series in the poloidal direction, yielding a standard tridiagonal matrix solver. To address the singularity of the polar coordinates at the radial origin (*r*=0), the equidistant radial grid mesh is shifted half step-size from the origin, which due to an important cancelation property, yields a closed system of equations [3]. It shall be called Lai's method henceforth.

The same test-case Eqs. (2)–(3) was also implemented as another stand-alone code using the solver of interest for the project, namely, the Sardourny's line integral method on hexagonal grids [1] of the GKMHD code, which shall be referred to as the Sadourny's method in the remainder text. The bulk of the effort in implementing Sadourny's method lies in establishing the relations between each grid-node (u_0^j) and its six nearest-neighbors (u_m^j), determined by the choice made in Sec. 9.3 for their local orientation. Further, the corresponding six ratios

$$G_m^j \equiv \frac{\overline{Q_m^j Q_{m+1}^j}}{\overline{P_0^j P_m^j}} \tag{5}$$

and the elemental dual-space hexagon area $s_\star^j$ in Eq. (1) must also be calculated (recall Fig. 78). Here, an alternative way to do so has been developed and checked against the original method implemented in GKMHD. The new method has the advantage that it explicitly calculates the $s_\star^j$ hexagonal element areas of the Voronoi diagram (dual-space) from the Q_m^j's, unlike the original, which approximates them to be 1/3 of the areas of the element hexagon made from the grid-nodes P_m^j. In spite of this, no significant impact on the accuracy was yield, but the exercise served as a validation for the original approximation.

The last step in the process uses the previous information to recast the system of equations (1) into a matrix form. Doing so yields a symmetric and sparse matrix due to the intrinsic conservation properties of the numerical method and its 7-point stencil, respectively. Its structure is depicted in Fig. 82. Following what is done in GKMHD, the IBM-WSMP library is invoked to invert this matrix and find the solution. In practice, for that to happen, the Laplacian sparse matrix at hand must first be converted into the compressed sparse row (CSR) format. This is a non-trivial task due to the involved global grid-node neighboring relations together with the symmetry of the matrix, which together yield a spatially dependent algorithmic stencil. This procedure was checked in GKMHD and implemented in the stand-alone Sadourny test-code. The code was further extended to alternatively allow the use of the PETSc library to solve the linear problem, which although already implemented in GKMHD, was not working there. In this case the task of representing the matrix structure is easier since no explicit format conversion is needed. Of course, other methods could also be used for the purpose of inverting the sparse matrix, and in particular, the popular multi-grid method would be a strong candidate. This in fact has been assessed within the scope of previous HLST projects, e.g. the MGTRIOP of K.S. Kang, so it shall not be addressed here.

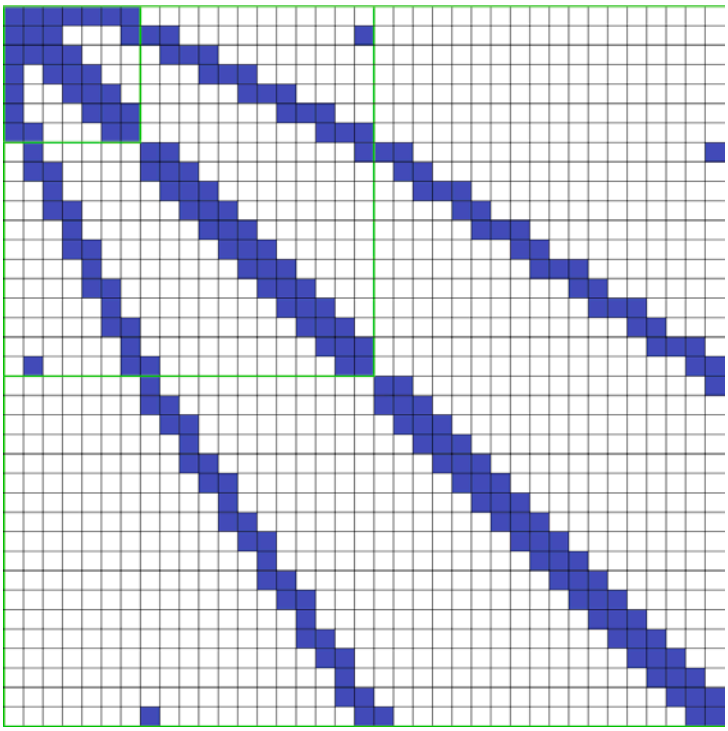

Fig. 82 Representation of the symmetric sparse matrix yield by Sadourny's method on the discrete Poisson problem with four flux surfaces (grid-mesh of Fig. 79) and Dirichlet boundary conditions.

The plots in Fig. 83 show the numerical discretization of the (continuous) source term (3) of Fig. 81, for each of the two solvers methods, using the same poloidal resolution of 96 grid-nodes on the outer boundary contour. The adequacy of the chosen source term to the problem under study is noteworthy. It demands high resolution in the radial periphery of the domain, as is typically the case for tokamak turbulence simulations. This renders the Lai's polar coordinates' equidistant grid quite inefficient, since the grid-resolution increases towards the centre with $1/r^2$. Hence, to obtain the needed mesh resolution on the outer domain, one necessarily wastes resources on the inner region, which becomes over-resolved, as seen in the right plot of Fig. 83. The same does not happen for the Sadourny's hexagonal grid illustrated by the left plot of the same figure. Such property is actually one of the main motivations for using the latter, which keeps the resolution fairly constant across the whole radial domain.

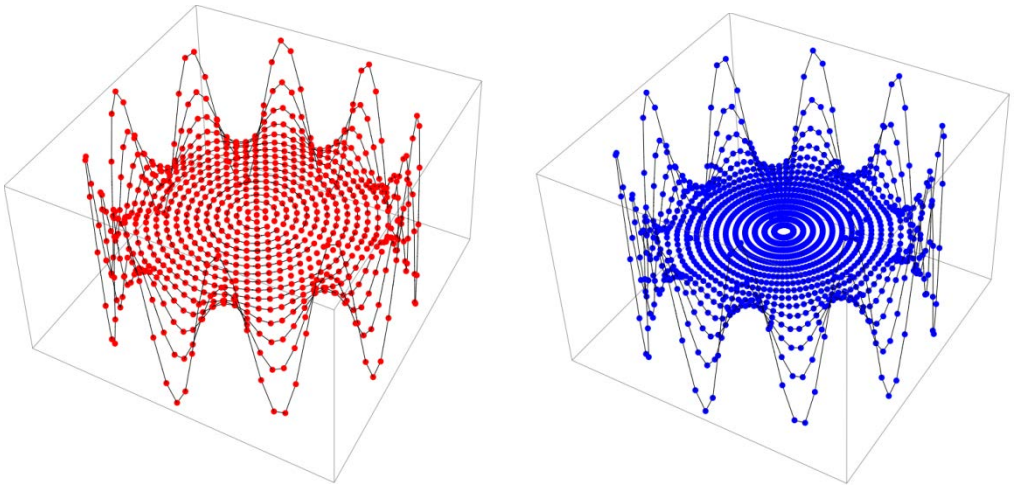

Fig. 83 Discrete representation of the source term (3) for p=10 on Sadourny's hexagonal grid (left, in red) and Lai's polar grid (right, in blue). Both grid-counts yield the same number of grid nodes on the outer radial contour, namely 96, but it is clear that the polar total grid-count (1536) is much higher than the hexagonal counterpart (817), leading to unnecessary resolution on the more internal radial contours.

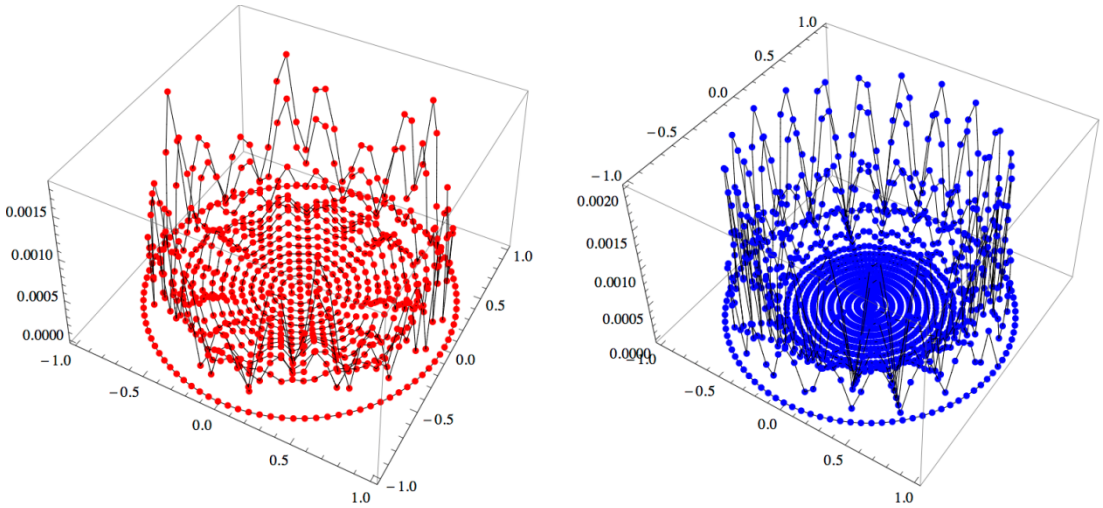

Fig. 84 Absolute difference (truncation error) between the numerical and analytical solutions (4) for Sadourny's method (left) and Lai's method (right), with p=10 on the same grid counts as before, namely, 817 and 1536, respectively. Despite this difference, both methods yield similar precisions, namely, maximum errors of 1.86e-3 (Sadourny) and 2.01e-3 (Lai).

Since the circular test-case possesses a non-trivial analytical solution, the comparison with the numerical solution provides an important direct check on the correctness of the implementation of the methods, in particular of the one employed in GKMHD. This is one important goal of the project TOPOX, as agreed with the project coordinator (Bruce Scott), given that this code is still in a development phase and therefore has not yet been extensively tested. Moreover, the existence of analytical solutions provides also a way to directly inspect the precision of the methods. The plots in Fig. 84 show the maximum global truncation error, defined here as the absolute value of the difference between the analytical and numerical solutions, for the same test case used before. One sees that, despite the factor of two smaller grid-count used for Sadourny's method, it yields a similar precision to Lai's counterpart. Moreover, if one repeats the exercise increasing the grid count in Sadourny's method to a similar figure (1519 grid nodes) to that used before with Lai's method (1536 grid nodes), the maximum absolute error decreases by a factor of two, from 1.86e-3 to 9.49e-4.

9.4.2. **Extension to include the open flux surface region**

After having devised the extended numerical scheme to include the X-point and the open flux surface region using the rules briefly outlined in Sec. 9.3.2., we need to find an appropriate domain for our test-case. It should be simple enough to be treated analytically but at the same time mimic a diverted tokamak magnetic equilibrium, therefore including an X-point. Two such candidates have been devised for that purpose.

The former consists of constructing the simplest possible poloidal magnetic flux function made out of two adjacent regions matched continuously. One with a parabolic shape, to yield circular flux surfaces,

$$\Psi_c = -\frac{B_0}{2}[(R-R_0)^2 + (Z-Z_0)^2] + \Psi_a \tag{6}$$

with an O-point located at (R_0,Z_0), and the other with a hyperbolic shape and a saddle-point, to yield an X-point located (R_X,Z_X) and the corresponding divergent flux surfaces

$$\Psi_h = -\frac{B_0}{2}[(R-R_X)^2 - (Z-Z_X)^2] + \Psi_X \tag{7}$$

where the remaining symbols are simply rescaling constants. The drawback of this approach is that, even though the poloidal function is continuous and continuously differentiable (class C^1), it's second derivative is discontinuous. As a consequence, its Laplacian is not smooth and therefore not well suited for our numerical test-case.

The latter candidate is a bit more involved but exhibits second order continuous differentiability (class C^2), meaning that its Laplacian, i.e. the source term in Poisson's equation (2) is smooth. It further yields more realistic shapes of the flux contours, although at the cost of more effort to adapt for the test-case. It is based on a magnetic island equilibrium description according to

$$\Psi_{\text{island}} = -\frac{B_0}{2}\left\{(R-R_0)^2 - \frac{a_0}{2}\left[1+\cos\left(\pi\frac{Z-Z_0}{ba_0}\right)\right]\right\} \tag{8}$$

where, the meaning of the coordinates is the same as before and the remaining symbols are simply rescaling constants. This function can be further modified by means of multiplying its terms with extra constants such to (i) shift the flux surfaces radially outwards to some degree, to mimic the Shafranov shift, and/or to (ii) yield two separatrices instead of a single one connecting both X-points, which is experimentally denominated a disconnected double null configuration. Here we shall not explore these possibilities. Instead, we are interested in a single-null (single X-point) configuration since it is the simplest configuration that includes all the needed ingredients. For that purpose an extra term was added to Eq. (8) according to

$$\Psi_{\text{island}} = -\frac{B_0}{2}\left\{(R-R_0)^2 - \frac{a_0}{2}\left[1+\cos\left(\pi\frac{Z-Z_0}{ba_0}\right) - c_0\frac{1+\tanh(2Z)}{2}(Z-Z_0)^2\right]\right\} \tag{9}$$

The hyperbolic tangent factor in this term allows for a smooth global function with a single saddle point (X-point), therefore keeping its C^2 differentiability. This ensures the existence of an analytical Laplacian, which is crucial for the comparison with the numerical solution provided by the Poisson solver. The plot of such a poloidal flux function can be seen in Fig. 85.

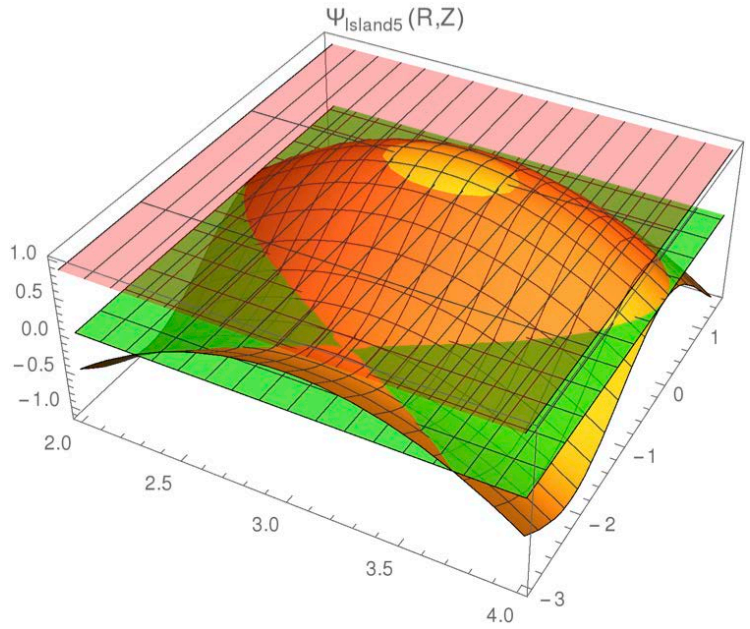

Fig. 85 Poloidal flux function yielded by Eq. (9). The two cuts shown in pink and green illustrate the underlying shape of the flux surfaces in the core region and at the separatrix, respectively.

An example of flux surfaces yielded by such analytical poloidal potential are shown in Fig. 86, on which an appropriate hexagonal mesh was constructed for Sadourny's scheme, i.e. the rules laid out in section 9.3 had to be observed. The closed flux surface region was discretised just like in the test-case of sub-section 9.4.1, where the radial label of the grid contours coincides with the equilibrium flux label up to the discretization resolution, as illustrated by the black dashed lines in Fig. 86. For the open flux surface region in the SOL this is no longer the case, and the radial grid-contours cross different flux surfaces near the X-point.

Using an equidistant arc-length between grid-nodes along the field lines was found to minimize the local deformation of the elemental hexagons (made by each grid-node and its six nearest neighbors) that results from the flux surface shaping. Still, the grid-nodes that lie on the horizontal line that goes through the X-point will most probably pose a challenge to the numerical method. The reason being that those and two of their consecutive nearest neighbors lie in the very same flux surface, which yields a triangle with three very disparate angles, therefore resulting in high local mesh deformation, which might even render the Voronoi diagram unfeasible. This is highlighted in Fig. 86 by the light red hexagons. Whether this issue impacts the numerical solution needs to be checked with numerical tests, for which the analytical solution that our extended test-case possesses is very helpful. In any case, alternatives to the current proposed solution can readily be devised. Namely, to use a different criteria for the grid-node's distribution, for instance based on setting limiting values for the angles of the elemental hexagons. Otherwise, one could resort to use a different kind of approach, whereby the property that each grid-node has six nearest neighbors is locally relaxed, therefore breaking the current global hexagonal topology of the mesh.

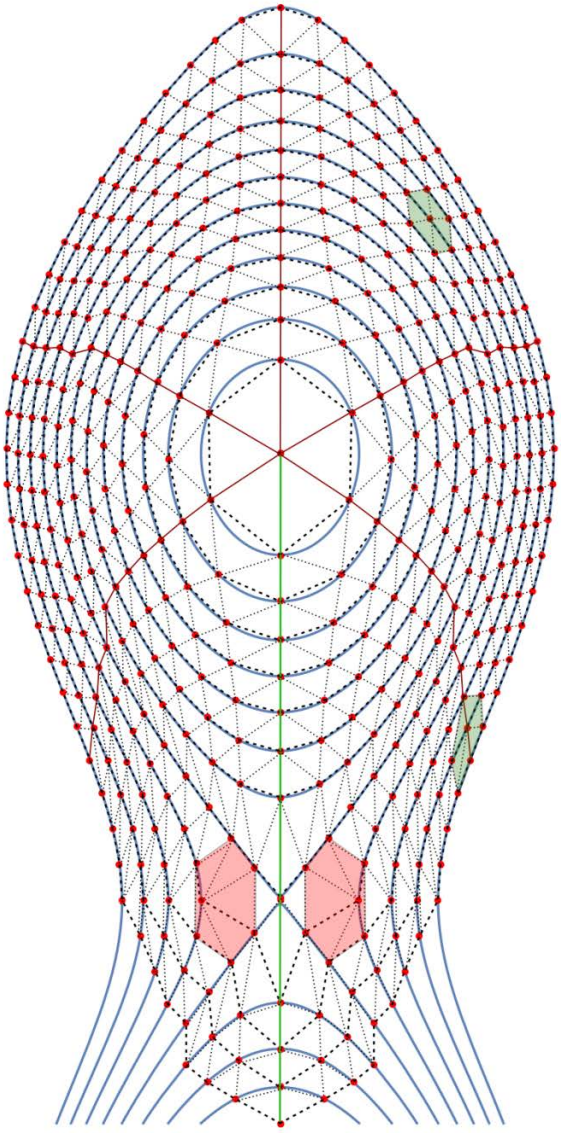

Fig. 86 The flux surfaces yielded by Eq. (9) are show in blue and the six main triangles of the discrete domain are represented by the red lines and the branch-cut by the green line. The radial grid contours are represented by the dashed lines. They coincide with the flux surface contours in the closed flux surface region but not in the open flux surface region, where they cross flux surfaces near the X-point. The grid-nodes appear as red dots, and together with their six nearest neighbors form the discrete hexagonal grid shown by the dotted lines. Four grid-nodes are highlighted with their corresponding elemental hexagons shown in colour. The red ones near the X-point illustrate the deformation of the underlying elemental triangles.

The last step of extending the test-case code with the new mesh and solving for the corresponding Poisson equation numerically constitutes the method's proof of principle. Knowing the analytical solution (9), and therefore also the Dirichlet boundary conditions for the numerical solution, and knowing the source function given by the Laplacian of (9), allows to solve the system (2) numerically using Sadourny's method. Implementation-wise, this task is mostly complete, although a subsequent debugging phase is required as the test-case is not yet working. However, as the end of the project was meanwhile reached, it was not possible to complete the extended test-case, but its current status should allow for the project coordinator to complete it and test the proposed extended Poisson solver method.

9.5. *Conclusions and outlook*

The project TOPOX2 proceeded as planned with a possible extension to the Poisson solver method used in the GKMHD code devised. This task encompassed several stages which we briefly summarise now.

Sadourny's numerical method [1] used in the GKMHD code was implemented in a stand-alone Poisson solver test-case on a circular disk, which was used for validating it against analytical solutions. To invert the sparse matrix yielded one can choose between using either the WSMP or the PETSc library. A standard pseudo-spectral solver was additionally implemented as an alternative way to find the numerical solutions. This further allowed for numerical comparisons between both methods.

In order to implement the extension to include the SOL region, several simplified analytical poloidal flux functions were devised, such to yield a set of nested flux surfaces whose topology mimics that of a diverted tokamak equilibrium. The chosen candidate uses a magnetic island equilibrium description, with the poloidal flux modified with a hyperbolic tangent function such to mimic a single-null magnetic topology while keeping its second order differentiability. This ensures the existence of an analytical Laplacian, which is crucial for the comparison with the numerical solution provided by the Poisson solver. A hexagonal mesh was constructed for this equilibrium according to the needs of the Sadourny's scheme. Using an equidistant arc-length between grid-nodes along the field lines was found to minimize the local deformation of the elemental hexagons (made by each grid-node and its six nearest neighbors) caused by the flux surface shaping. Nevertheless, a potential issue related to local grid deformation was identified near the X-point. It could not be tested numerically since the implementation of the extended test-case could not be completed as the end of the project was meanwhile reached. Even though the code implementation was nearly finished, a debugging phase, which is potentially time consuming is needed.

Additionally, in the case where the mentioned local deformation issue reveals itself unsurmountable, new solutions have to be devised, either by using different poloidal distribution of nodes along the mesh's main hexagons, or by locally relaxing the constraint that each grid node has six nearest neighbors. A significant amount of time would be needed, rendering these activities beyond the scope of the current project. However, we believe that the current status of the test-case should allow the project coordinator to take over and proceed with the testing of the proposed scheme in its current form.

9.6. *References*

[1] R. Sadourny, A. Arakawa and Y. Mintz, *Mon. Weather Rev.* **96** (1968) 351
[2] Trach-Minh Tran, private communication (2013)
[3] M.C. Lai, *Numer. Methods. Partial Differential Equations* **18** (2002) 56

10. Final report on HLST project VIRIATO

10.1. *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.

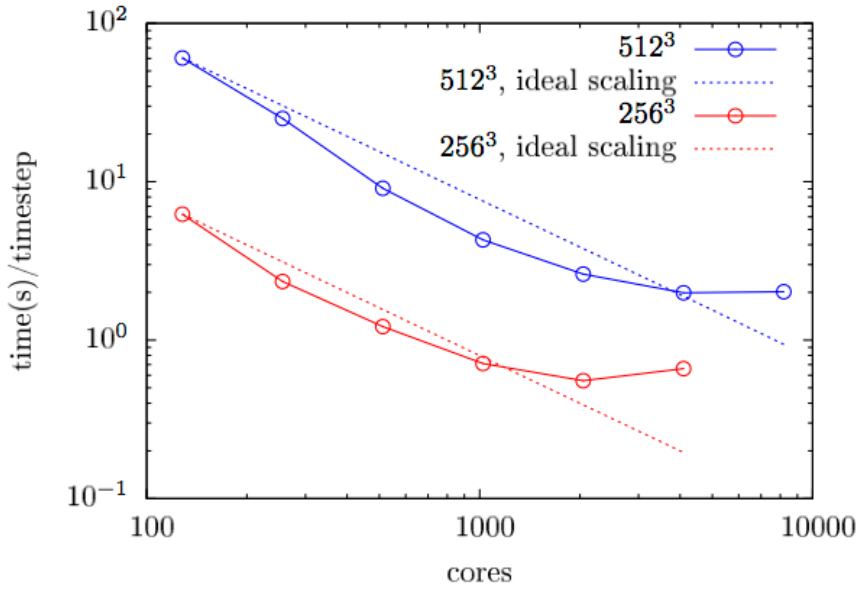

Fig. 87 Strong scaling of the original VIRIATO code with 16 velocity moments and a configuration space grid-count of 256^3 (red) and 512^3 (blue).

10.2. *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, such 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 Forckeck 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.

10.3. *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 (2D FFTs) dominate the costs, representing at least half of the total simulation time for relevant production cases. For instance, Table 14 shows the profiling for a typical medium-sized domain simulation running on 256 cores, where the 2D 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 14 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.

10.4. *2D 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 2D 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 $O(npe)^2$.

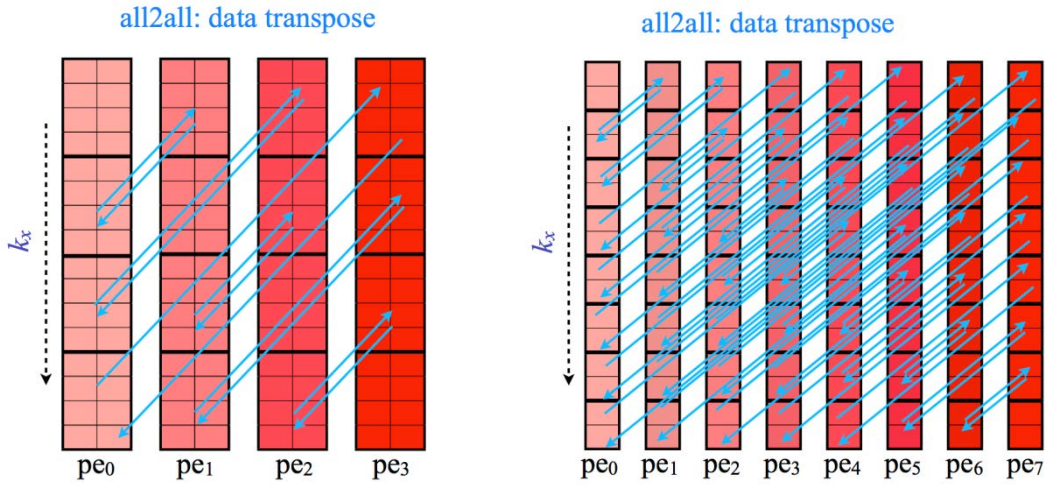

Fig. 88 Illustration of the all-to-all communication pattern for the parallel transposition of a 2D dataset parallelized over four (left) and eight (right) cores.

This is clearly shown in Fig. 88, where the all-to-all communication patterns involved in transposing the same 2D 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.

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

Since a significant part of the work was dedicated to the parallel 2D 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 2D FFT algorithm, therefore allowing to study their behavior 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 2D FFTs both in forward and backward directions on a 4D data-set, such to allow comparing the final doubly transformed data-set with the original untransformed one. A sequence of convolutions is also implemented such 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 2D 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 2D FFT algorithm has been implemented in the test-case code and then extended to minimize the accumulation of network latency for large number 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 2D FFT algorithms 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 also is prone to network latency accumulation. To understand this, we need to recall Fig. 88 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 show in Fig. 87. 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 linearly this cost. To avoid this accumulation, the data in the remaining dimensions can be aggregated such to be carried during one single transposition. The transpose algorithm was extended to do so.

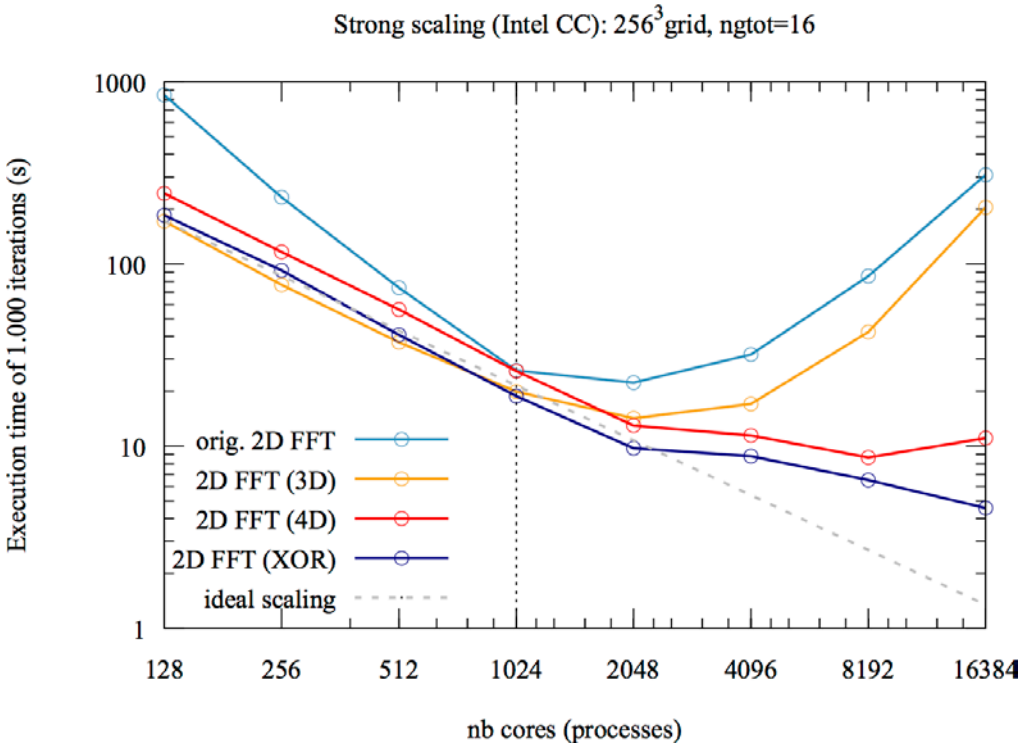

Fig. 89 Strong scaling of the parallel 2D FFT algorithm on the same medium-sized domain of Fig. 87. VIRIATO's original algorithm is show in light-blue. The extended algorithm with the data aggregated over the *z*-dimension and over the *z*- and *ng*-dimensions are shown in yellow and red, respectively. The dark-blue curve shows the behavior of the whole algorithm when the alternative XOR-transpose is employed together with data aggregation over the *z*- and *ng*-dimensions.

The impact of these changes on the parallel performance was measured using the test-case described previously and the results are showed in Fig. 89 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 light-blue line represents the strong scaling of the original 2D FFT algorithm in VIRIATO. It mimics fairly close the scaling of the full VIRIATO code (Fig. 87), 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. 88). 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 yellow 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 computational 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 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 red line uses the same algorithm, except that it it further aggregates the data in the velocity space dimension when the transpose is carried. If on the one hand the MPI-buffer overhead is higher because the messages are (*ng=16*) times bigger compared to the solid yellow 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 (red and yellow) for the lower *npe* numbers.

Beyond 1024 cores, indicated by the dashed vertical, 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 we know that the complexity of the all-to-all communication pattern involved grows with $O(npe)^2$. That the light-blue and yellow curves show a qualitatively similar behavior (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 more 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 costs. The difference between the light-blue or yellow curves and the remaining two, whose strong scaling still show the expected monotonic decreasing shape, comes from the data aggregation over the moment direction 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 yellow curve (*ng*=16), and additionally by the *z*-grid-count divided by *npez* for the light-blue curve (*ng*Nz*/*npez*=32). The difference between red and dark-blue curves 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 that the transpose used originally in VIRIATO. In terms of speedup figures between the latter (fastest) and the original method (light-blue) the figures are approximately 2x, 4x, 15x and 55x for 2048, 4096, 8192 and 16384 cores, respectively.

10.6. ***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 2D algorithms.

We have seen in Sec. 10.5 that the best result was obtained using the data-aggregation technique together with the XOR-transpose (dark-blue curve in Fig. 89). 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. 90, for the case with N_y=4.

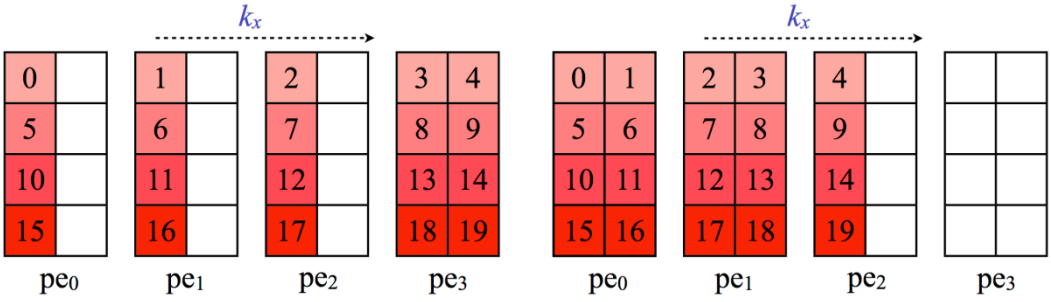

Fig. 90 Illustration of the different storage of the radial Fourier modes over the sub-domains yielded by the original transpose (a) and new XOR-transpose methods (b).

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 2D 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 2D FFT method used in VIRIATO (red curve in Fig. 89) 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, to 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 3^{rd} and 3^{rd}+4^{th} dimensions technique to minimize the latency accumulation, as explained in Sec. 10.5. 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.

10.7. ***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. 89 refer to a single 4D 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 3D 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 2D FFTs of those quantities will degrade much earlier because smaller messages are involved. Second, even for the 4D 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 2D FFT method) and red (extended 2D FFT method with transpose invoking data aggregation in *z* and *ng*-dimension) curves in Fig. 89 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. 91.

The light-blue curve represents the scaling of the original VIRIATO code. It has the same shape of the blue curve depicted in Fig. 87. The yellow 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 3D. The red curve corresponds to the same method except for the coefficients of the Hermite polynomials in velocity space. Being 4D quantities, they allow in principle to further invoke the data aggregation in the *ng*-direction (on top of aggregation the *z*-direction) during the transposes. Unfortunately, inside the *P_Loop* region, data dependencies prevent this. There, the calculation of each moment coefficient *m* depends on the *m*-1 coefficient 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 red curve in Fig. 91 compared to the one in Fig. 89.

In practice, even though more modest than the isolated test case results, being able to extend the region where VIRIATO show perfect linear scaling beyond the point where the degradation used to start (vertical dashed line) is something very positive. Especially if one considers that this tendency is expected to improve when more velocity moments (*ngtot*=256 instead of 16) are used, which is something that the code developers of VIRIATO are aiming for. Such problem sizes demand the usage of higher numbers of cores, so the absolute gain in CPU hours yield by the new version of the code is quite important.

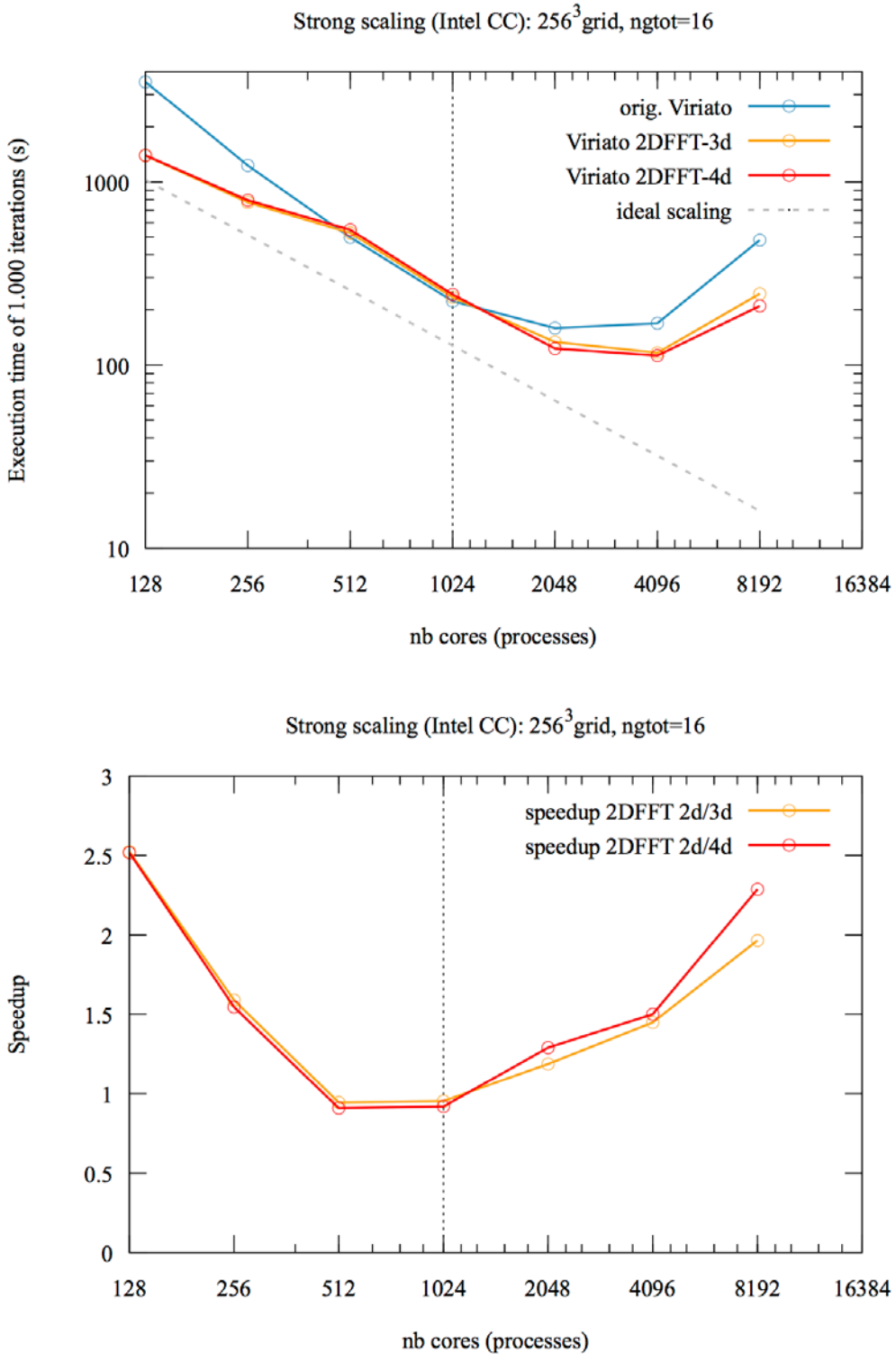

Fig. 91 Strong scaling (top) and corresponding speedup for the full VIRIATO code on the same medium-sized domain of Fig. 88. The same colour coding of Fig. 89 is used here.

10.8. ***Strategy for the hybrid parallelization***

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 are two significant issues with such a model. First, since each thread would need to spawn MPI tasks, the level of MPI thread-safety would need 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.

The solution proposed here avoids the main issues described before by creating 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 required by the data transposition step. Hence, only a level of thread-safety of *MPI_THREAD_FUNNELED* is required to the MPI library.

More importantly, 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 2D FFTs. The reason is that, since only the MPI tasks, not the OpenMP threads, need to be involved in the MPI transpose communication, the number of needed messages can be significantly reduced. Hence, the all-to-all communication complexity would be 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 would reduce substantially complexity of the all-to-all communication, by reducing *npe* by a factor of four, while still having the access to the full inter-node network bandwidth for the MPI communication.

However, this approach implies code modifications in the remaining parts of VIRIATO, to ensure that the extra resources (threads) allocated to the FFT algorithm are also used elsewhere, to avoid have idling processes during runtime. This problem becomes 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. There is no technical impediment in doing this other than the man-power needed to implement such changes.

An alternative to the hybrid scheme just proposed can also be devised using two sub-communicators for the *npe* decomposition. In this scenario, only a subset of *npe* tasks would be involved in the inter-node communication (transpose), while the remaining part would be responsible for intra-node data communication (*MPI_Gather/Scatter*) or would share the data via “shared memory segments” [1] which are now part of the MPI 3.0 standard. All of the *npe* tasks would be in any case available for the floating point operations and this number does not change compared to the original MPI domain decomposition. So, the remainder code lines need not be changed like in the hybrid version.

To explore the possibility of implementing these schemes in the 2D FFT algorithm, another simplified test-case consistent with VIRIATO’s framework has been devised to study its behavior in a controlled manner. Up to now, an effort has been put on calculating the forward 1D FFT part of the algorithm using OpenMP threads. The transformation results obtained are correct but show some degree of irreproducibility in terms of their performance scaling with the number of threads, for reasons not yet clear. More effort needs to be put on this issue, probably by using a more sophisticated performance analyzing tool, like the Intel VTune.

Since the end of the project was reached before these tasks could be completed, the work had to be halted. However, guidelines have been provided to the developers of VIRIATO to allow them to further explore the proposed hybridization of VIRIATO on their own.

10.9. *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 emphasis improving the transpose algorithms at hand, which as reported here, led to significant gains in the overall parallel 2D 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. This will further greatly facilitate the task of handing back the optimized VIRIATO code to them, since they are already familiar to some extent with the changes being introduced.

10.10. *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 2D 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 test-case, mostly by modifying the parallel transpose algorithm implied in these operations. The main idea was to aggregate data before carrying the all-to-all communication patterns to avoid penalizations due to network latency accumulation. Significant gains in performance have been achieved, in some situations reaching figures of the order of more that 50x on sixteen 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 perfect linear strong scaling.

The remaining time of the project was dedicated to the hybridization of VIRIATO. Having a hybrid MPI/OpenMP 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. In practice, this involves implementing a fork-join parallel execution model, where threads are spawned in cost-intensive regions of the code. Within this project we restricted ourselves to the hybridization of the FFT algorithm. In these, the complexity of the underlying all-to-all communication required can be greatly reduced [1], therefore allowing to further extend the scalability of VIRIATO. To that end, another test-case has been devised to exploit this technique. As the project reached its term before the tasks could be finished, a set of guidelines was given to the developers of VIRIATO on how to proceed on this topic.

It is also noteworthy that, once implemented, the hybridization concept has to be extended to the remainder computational intensive parts of VIRIATO (besides the FFTs), otherwise the extra resources allocated for the threads would be idling during those calculations. To avoid this limitation, but still take advantage of the node-level NUMA topology of current HPC machine to reduce the transpose communication complexity, another scheme using a pure MPI implementation can be devised. Using two instead of one MPI communicators for the *y*-domain decomposition (*npe*) can achieve this goal of splitting the intra-node from the inter-node communication. To this end, either the collectives MPI_Gather/Scatter can be used inside the node to collect or distribute the data before it is exchanged across nodes, or shared memory segments can be used to avoid this step, by allowing the tasks involved in the inter-node communication to access all of the data stored in their node. Since *npe* does not change compared to the original pure MPI domain decomposition, no changes

are required in the rest of the code. Further pursuing these activities is left for future work.

10.11. *References*

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