EUROfusion

EUROFUSION WPISA-REP(16) 16107

R Hatzky et al.

HLST Core Team Report 2013

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 2013

Contents

1. Executive Summary

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

In agreement with the HLST coordinator the individual core team members have been/are working on the projects listed in Table 1.

Project acronym	Core team member	Status
BLIGHTHO	Matthieu Haefele	finished
EMPHORB	Tamás Fehér	finished
ITM-ADIOS2	Michele Martone	running
KSOL2D-2	Kab Seok Kang	finished
MGTRIOPT	Kab Seok Kang	finished
MICPORT	Tamás Fehér	finished
PARFS	Michele Martone	finished
REFMULXP	Tiago Ribeiro	finished
TOPOX	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.

Tamás Fehér worked on the EMPHORB and MICPORT projects.

NEMORB is a global nonlinear gyrokinetic particle-in-cell code (PIC) that can be used for turbulence simulations in tokamak geometry. The aim of the EMPHORB project was to implement the phase factor transformation in the NEMORB code. The phase factor transformation can be used to decrease the grid resolution for linear calculations and thereby lower the amount of computational resources needed for the simulation. The implementation of the phase factor involves small changes that extend throughout the whole NEMORB code.

The code structure was analyzed and the phase factor was implemented for the main steps in the PIC algorithm. This includes the matrix building routines, the field evaluation, and the charge and current assignment. As the equation of motion modifies only the weights, the phase factor had to be introduced for pushing them.

With the introduction of the phase factor, several variables had to be changed from real to complex. The call-graph for the code was investigated to check which functions are influenced by the change of the variable types, and the necessary changes were implemented. The diagnostic methods have also been updated for the use of complex variables.

Finally, the modified code was tested by simulating electrostatic ITG modes. Several tests were made with different phase factors, and the results of the modified simulation agree well with the original code.

The aim of the MICPORT project was to port the JOREK code to the Intel MIC architecture in order to gain experience with the new hardware and its software stack. While it was easy to get the code running on the new hardware, the ported code does not perform well on the MIC architecture. Compared to a simulation on the host, the code runs approximately seven times slower in native mode on the Xeon Phi. To make effective use of the MIC hardware, the code has to scale up to hundreds of threads, and it should utilize the vector processing power of the

mathematical units. Both of these points provide a challenge in case of the JOREK code. The vector performance and the scaling of the JOREK code have been assessed on the new hardware.

It has been found that significant improvement in vectorization is possible for the matrix construction part of the code. With a separate example it has been shown, that the vectorization improves the performance both on the host and on the accelerator card, and the vectorized code becomes faster on the Xeon Phi than on the host. Theoretically this could lead to a factor of three speed-up during the matrix construction step if one uses two Xeon Phi cards.

There are two other main stages during a JOREK simulation step: LU factorization of the matrix (as a part of preconditioning) and solving a linear problem with an iterative method. For these steps, the PaStiX library is used. Therefore, the performance of these steps depends on the performance of the library. The PaStiX version that was used in this work is not optimized for the MIC architecture, and therefore, it is slow on the Xeon Phi, not only for the JOREK matrix, but also for a simpler test example. Because of this the overall speed-up without an optimized linear algebra library would be between 3–30%. But the PaStiX library is also evolving, and if a new PaStiX version would use both the host and the Xeon Phi effectively, then the JOREK code could also benefit significantly from the accelerator cards.

Matthieu Haefele supported the LCHD project and worked on the BLIGHTHO project.

Matthieu Haefele has contributed to the LCHD project. He made first contact to the project coordinator, Guillaume Latu, and established the work plan for the project. This was especially helpful as Silvia Espinoza just joined HLST and the time scale for the project was quite tight. The project gave Silvia Espinoza the opportunity to train her skills in managing projects and she could profit from Matthieu Haefele's knowledge in the field of parallel-IO and data compression.

The BLIGHTHO project is explicitly providing support on different levels for the European scientists who use the Helios machine. HLST has access via the trouble ticket system of CSC to most of the tickets submitted by the European users. This gives the flexibility to pick up special concerns of users whenever necessary. In addition, the BLIGHTO project investigates topics which are of general interest such as checking and improving the documentation provided by CSC. Especially this year we helped CSC to reconfigure the whole procedure for threads and task pinning on Helios, and the corresponding documentation was updated accordingly. This is especially of interest when OpenMP is used within a shared memory node.

The main contribution addressed in this report is the continuation of the extensive evaluation of the MPI libraries available on the Helios machine. We found out that both Bull and Intel MPI libraries require a large amount of time when initializing a so-called ALL_TO_ALL operation and we have strongly contributed to reduce the effort in collaboration with the CSC support team and Bull. Some alternative implementations have been thoroughly tested on several supercomputers. These alternatives basically consist of reducing the number of communicating MPI tasks during the communication operation. The outcome of this study was presented at an international conference on parallel computing. The results of this study are interesting as they show that a distributed matrix transposition on 64k cores with one MPI task per core is possible. Runs of simple applications revealed several issues on large numbers of cores which are now fixed or about to be fixed. Consequently, the whole user community of Helios will benefit from this experience.

Kab Seok Kang worked on the KSOL2D-2 and MGTRIOPT projects.

The KSOLD-2 project aims at improving the Poisson solver of the BIT2 code, which is a code for Scrape-Off-Layer (SOL) simulations in 2-dim real space + 3-dim velocity space. It is necessary for the Poisson solver in BIT2 to scale up to very high core numbers in order to maintain the good scaling property of the whole code. To achieve this goal two approaches are pursued. One is based on the multigrid method with gathering of the data at a certain coarser level. The other is based on a domain decomposition method, so that the potential field in each subdomain could be calculated separately and only the boundary conditions of the subdomains have to be exchanged. Particularly the FETI-DP method is a promising candidate for an efficient non-overlapping domain decomposition method for 2D and 3D problems.

The initial implementation of the multigrid method in BIT2 suffered from excessive communication cost. To overcome this obstacle, all data are gathered at a certain level with an all-reduce operation involving each core. We adapted the data structure of BIT2 to make the gathering of the data possible. In parallel we have started to implement the FETI-DP method for the SOL geometry in two different versions: as a finite difference method (FDM) and as a finite element method (FEM).

As the current project came to an end it is planned to continue the work in a follow-up project. There is still the debugging phase to be completed to be able to study the numerical performance of the FDM and FEM within the FETI-DP method. Such tests are scheduled on the Helios machine at IFERC-CSC.

In the previous MGTRI project, we had developed a parallelized multigrid solver for the parallelized GKMHD code. GKMHD is a serial MHD equilibrium solver which evolves the Grad-Shafranov MHD equilibrium. Especially for a very high number of cores the pure MPI implementation seems to be not optimal. Therefore, the parallel multigrid solver has been enhanced to a hybrid OpenMP/MPI model. In general, the turning point where the hybrid model becomes more efficient than the pure MPI model shifts to smaller numbers of cores, the smaller the problem size is. Therefore, an improvement can be achieved with the hybrid model in cases with a small number of degrees of freedom (DoF) per core on a large number of cores. However, if more than eight threads are used per MPI task the efficiency usually starts to degrade.

Michele Martone worked on the PARFS and ITM-ADIOS projects.

The main goal of the PARFS project was to adapt the HYMAGYC code to support simulation resolution up to that needed for ITER-like configurations. HYMAGYC is being used to study linear and nonlinear dynamics of Alfénic type modes in Tokamaks in the presence of energetic particle populations. It is a hybrid simulation code that features both particle-In-cell (PIC) and Magnetohydrodynamic (MHD) components. While the PIC module could scale up to several hundreds of processes, the field solver's computational core (the linear solver) which was bound to be serial, using an own ad hoc implementation. Hence, the serial sparse linear solver was a bottleneck to the scalability of HYMAGYC.

Initial activities included changes for portability and standards' compliance. With our collaboration, project coordinator Gregorio Vlad made the necessary changes in HYMAGYC to allow the use of parallel solvers. In this context, the best solver we have been able to identify was MUMPS-4.10. A strong scaling test based on the MUMPS solver has been performed on the Helios machine from a minimum allowed (for memory reasons) of four nodes/MPI tasks to a maximum of sixteen nodes/256 MPI tasks for the most relevant case (ITER sized, with $1.4 \cdot 10^6$ equations and $5.5 \cdot 10^8$ nonzeroes). The scalability properties of this solution are a mere factor of two for the LU factorization part, and a factor of six for the more time critical "backsolve" part. This showed that the problem under consideration is quite demanding for direct

solvers as they require an excessive amount of memory for the LU factorization. Therefore, iterative solvers might be more appropriate as long as their convergence rate is satisfactory. Nevertheless, the present project has enabled HYMAGYC to handle ITER sized cases. As a consequence of the solver-related code being now parallel, further solvers could be easily tested in the future.

The project's goal of ITM-ADIOS is to evaluate the usage of the ADIOS parallel I/O library on the Helios machine to provide authors of I/O intensive ITM codes with best practices guidelines. During the first ITM-ADIOS project a C based ADIOS-1.3 benchmark program (IABC) was developed, which served as a valuable tool to explore ADIOS performance on the file systems of the HPC-FF and Helios supercomputers. Unfortunately, IABC showed to be incompatible to the newest version of ADIOS-1.5. Moreover, being programmed in C instead of Fortran, IABC may not serve as a good reference for ITM code developers. Therefore, a fairly small Fortran ADIOS-1.5 benchmark program was developed. Still, it required personal communication with the ADIOS developers to be able to get satisfying throughput again (from ~2 GB/s on one node to ≈ 20–25 GB/s with >= 32 nodes). The new results confirm our past experience in that ≈ 32 computing nodes saturate the I/O capacity, on either the `$SCRATCHDIR` or the `$WORKDIR` partition. The new small sized benchmark is simple enough that it should be straightforward for ITM users to adapt it to their needs.

Finally, we are working to provide the GENE team with a test ADIOS based I/O module. Depending on the results, we will study selected aspects of interest to ensure that the integration is seamless and beneficial. Tests with other major codes (GEMZA, NEMORB) are also foreseen. In this context certain questions are of interest e.g. how data layouts (e.g. array dimensions distribution across computing nodes) employed by the user's codes will influence efficiency. Additionally, we are providing the ITM Gateway machine with an ADIOS installation in order to ease development of ADIOS-based solutions by the ITM community.

Tiago Ribeiro worked on the TOPOX and REFMULXP 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. A complete numerical demonstration of this is yet to be made due to the outstanding challenge of resolving all the relevant space scales involved. 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 currently being used in the Grad-Shafranov equilibrium solver GKMHD, built on a triangular grid in *RZ*-space, with the points arranged along flux surfaces, which are topologically treated as hexagons.

The goal of the project was to extend such a scheme beyond the magnetic separatrix into the SOL, including the X-point. The first part of the work consisted in getting acquainted with the tools already available, namely the GKMHD code and its underlying solver algorithm. An appropriate stand-alone test-case was implemented using both Sadourny's method with the IBM WSMP library and an alternative finite-differences pseudo-spectral solver. The comparison of both methods demonstrated both the correctness of the former's implementation, as well as its advantages over the latter in terms of discretization efficiency.

The second part of the work was devoted to devising the extension of the Sadourny's method beyond the X-point, into the SOL. The solution found consists of 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. Everything outside this domain is discarded, meaning that the divertor itself is not included. However, the necessary

magnetic structure, namely the SOL, the private flux region and the X-point are kept together with the hexagonal topology of the closed field line region grid. This means that Sadourny's method can be directly applied to the open field-line region with only minor modifications. The relatively short time duration of the project (six months) for the ambitious tasks proposed meant that the implementation and test of such ideas were left for future work, in the framework of an eventual project extension.

Simulation of x-mode reflectometry using a finite-difference time-domain (FDTD) code is one of the most popular numerical techniques used, as it offers a comprehensive description of the plasma phenomena. This method requires, however, to keep the error to a minimum, a fine spatial grid discretization, which also implies a high-resolution time discretization to comply with CFL stability condition. As a consequence, simulations can become quite demanding on computational cost.

The REFMULXP project was devised to circumvent these questions by implementing a parallel version of the x-mode REFMULx code, developed at Instituto de Plasmas e Fusão Nuclear (IPFN). To guide the optimization strategy, the code was profiled, revealing the numerical kernel to be the hot-spot cost wise, as expected. Then, the serial performance of these code regions was improved by explicitly declaring the non-aliasing property of the pointers used therein, which is not the default in C-language. This aids the compiler cache optimization tasks, and resulted in practice in a 5.6 speed-up when the Intel compiler was used.

Additionally, the roofline model was invoked to quantify the performance bounds of the algorithm on Helios. However, the checks made so far raised doubts about the correctness of the Sandybridge hardware counters, a topic that is still under investigation.

The project proceeded with the implementation of the one-dimensional OpenMP parallelized REFMULx(P). Within a single node of Helios, which has sixteen cores, perfectly linear strong scaling was measured for the GNU C compiled (GCC) code, which with sixteen threads needed less than 1/10 of the time used by the original serial version. The same threaded code produced with the Intel C compiler (ICC) was always much faster, in absolute terms, even though its strong scaling started to degrade beyond four threads. The maximum speed-up, obtained with eight threads, yielded a gain of almost 24 times in this case. Finally, it is noteworthy that the saturation reported here for a modest number of threads constitutes no obstacle to the plan of further increasing the dimensionality of the domain decomposition, since this will not be the case for the planned three-dimensional version of the model.

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

1.2.1. **Dissemination**

Hatzky, R.: The experience of the High Level Support Team (HLST), *IFERC-CSC Report Meeting*, 20th March 2013, Kyoto, Japan.

Kang, K.S.: IPP, EFDA – Fusion research in Europe, *National Institute for Mathematical Sciences (NIMS)*, 24th January 2013, Daejeon, South Korea.

Kang, K.S.: HPC and Parallel programming, *National Institute for Mathematical Sciences (NIMS)*, 25th January 2013, Daejeon, South Korea.

Kang, K.S.: The parallel Multigrid method, *National Institute for Mathematical Sciences (NIMS)*, 29th January 2013, Daejeon, South Korea.

Kang, K.S.: Numerical comparison of domain decomposition methods with the multigrid method, *National Institute for Mathematical Sciences (NIMS)*, 30th January 2013, Daejeon, South Korea.

Kang, K.S.: Fusion research and HPC, *Department of Computational Science and Engineering, Yonsei University*, 31st January 2013, Seoul, South Korea.

Martone, M.: A Sparse BLAS implementation using the "Recursive Sparse Blocks" layout*, Chair of Scientific Computing (SCCS) of TUM*, 23rd April 2013, TUM, Garching, Germany.

1.3. *Training*

Tamás Fehér has given a webinar for Helios users:
Forcheck – A Fortran source code analyzer, 20th June 2013, Garching, Germany.

Ribeiro, T.: NEMOFFT project: Improved Fourier Algorithms for Global Electromagnetic Simulations, IPP report 19/2, Max Planck Society eDoc Server, [online], 2013.

1.3.1. **Internal training**

The HLST core team has attended:
The HLST meetings at IPP, 11th April and 16th October 2013, Garching, Germany.

Tamás Fehér has attended:
Intel Xeon Phi Coprocessor – MIC-Programmierung, Intel Software-Tools & Programmiermodelle, Internationales Congresszentrum München (ICM), 23rd January 2013, München, Germany.

Tamás Fehér, Matthieu Haefele and Michele Martone have attended:
GPU programming workshop, RZG, 4th March 2013, Garching, Germany.

Roman Hatzky, Matthieu Haefele, Kab Seok Kang, and Tiago Ribeiro have attended the workshop:

- Numerical Methods for the Kinetic Equations of Plasma Physics, 2th – 6th September 2013, IPP, Garching, Germany.

Tamas Fehér, Matthieu Haefele, Michele Martone, and Tiago Ribeiro have attended the workshop:

- Advanced Fortran Topics, 16th – 20th September 2013, LRZ, Garching, Germany.

Matthieu Haefele has attended:
PRACE PATC course: Node-level performance engineering, 3rd – 4th December 2013, LRZ, Garching, Germany.

1.3.2. **Workshops & conferences**

Roman Hatzky and Tiago Ribeiro have attended:
IPP Theory Meeting, 18th – 22th November 2013, Damerow, Germany.

Fehér, T.: Introduction to the Intel Xeon Phi coprocessor, *IPP Theory Meeting*, 18th – 22th November 2013, Damerow, Germany.

Hatzky, R.: Evaluating the potential for improvement of SOLPS parallelization, *SOLPS Workshop*, 29th – 31st July 2013, IPP, Garching, Germany.

Kang, K.S.: Parallel multigrid solvers using OpenMP/MPI hybridization, *EU-Korea Conference on Science and Technology (EKC2013)*, 24th – 26th July 2013, Brighton, UK.

Kang, K.S.: A fast parallel Poisson solver for the scrape-off-layer, *Int. Conference on Domain Decomposition Methods*, 16th – 20th September 2013 Lugarno, Switzerland.

Kang, K.S.: Fast parallel solvers, *IPP Theory Meeting*, 18th – 22th November 2013, Damerow, Germany.

Kang, K.S.: A fast parallel multigrid solver, *New Algorithms for Exascale Computing Workshop*, 4th – 6th December 2013, Cologne, Germany.

Martone, M.: A Shared Memory Parallel Sparse BLAS Implementation using the Recursive Sparse Blocks format, *Sparse Days Meeting 2013*, 17th – 18th June, Toulouse, France.

Martone, M.: Accessing librsb's Shared Memory Parallel Sparse BLAS Implementation from GNU/Octave, *OctConf 2013*, 24th – 26th June, Milan, Italy.

Martone, M.: Cache and Energy Efficiency of Sparse Matrix-Vector Multiplication for different BLAS Numerical Types with the RSB Format, *Int. Conference on Parallel Computing*, 10th – 13th September 2013, Garching, Germany.

Ribeiro, T. and Haefele, M.: NEMORB'S Fourier filter and distributed matrix transposition on Peta flop systems, *Int. Conference on Parallel Computing*, 10th – 13th September 2013, Garching Germany.

1.3.3. **Meetings**

Roman Hatzky attended on a regular basis:

- CSC management meeting
- CSC European ticket meeting
- IFERC-HLST-Bull research & development meeting

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

2.1. *Introduction*

NEMORB is a global nonlinear gyrokinetic particle-in-cell code that can be used for turbulence simulations in tokamak geometry. The aim of the EMPHORB project was to implement the phase factor transformation in the NEMORB code. Phase factor transformations can only be used for linear simulations, but for such calculations it effectively decreases the amount of computational resources needed for the simulation.

The structure of this report is the following: first the concept of phase factor will be introduced, and then we discuss the changes in the discretized equations. We will report on how these equations are implemented numerically, and in the end the modified code will be tested, and the results summarized.

2.2. *Phase factor*

The potential functions are represented with splines in the original NEMORB code

$$\Phi(\vec{r}) = \Phi(s,\theta,\phi) = \sum_{klm} \Phi_{klm}\Lambda_k(s)\Lambda_l(\theta)\Lambda_m(\phi),$$

where the Λ functions are the elementary B-spline basis functions. NEMORB uses a magnetic coordinate system: a position in space $\vec{r}(s,\theta,\phi)$ is described by the flux surface label *s*, the poloidal angle θ and the toroidal angle ϕ. The potential can have rapid oscillations in the angular directions. To remove these oscillations, we introduce the phase factor function

$$S(\vec{r}) = S(\theta,\phi) = \exp\left[i\left(m\theta - n\phi\right)\right],$$

and redefine the potential in the following form

$$\Phi(s,\theta,\phi) = \tilde{\Phi}(s,\theta,\phi)S(\theta,\phi) = \sum_{klm} \tilde{\Phi}_{klm}\Lambda_k(s)\Lambda_l(\theta)\Lambda_m(\phi)S(\theta,\phi).$$

Here $\tilde{\Phi}_{klm}$ are the spline coefficients of the phase factor extracted potential $\tilde{\Phi}(\vec{r})$. The idea behind the phase factor transformation is that the oscillating component (the *S* function*)* can be removed analytically from the linear equations, and $\tilde{\Phi}$ can be represented using significantly fewer discretization points than the original function. The phase factor is extracted from the magnetic potential $(A_{\|})$ in a similar way.

The plasma particles are described by the distribution function *f,* and the distribution function is represented by marker particles in the code

$$f(\vec{r},\vec{v},t) = \sum_k w_k \delta(\vec{r}-\vec{r}_k)\delta(\vec{v}-\vec{v}_k).$$

Here *k* is the index of the particle, $\vec{r}_k$ and $\vec{v}_k$ are the particle coordinates. The phase factor is also extracted from the distribution function

$$f(\vec{r},\vec{v},t) = \tilde{f}(\vec{r},\vec{v},t)S(\vec{r}) = \sum_k \tilde{w}_k S(\vec{r}_k)\delta(\vec{r}-\vec{r}_k)\delta(\vec{v}-\vec{v}_k).$$

In the phase factor version of the NEMORB code, we solve the gyrokinetic equations to find the solution in the phase factor extracted variables. When we introduce the phase factor function in the governing equations, then these equations become slightly modified. In the next section, we discuss the changes in the equations.

2.3. *Modified equations*

In NEMORB, the equations are discretized using the PIC method, therefore, we will have to discuss the four basic steps of the general PIC algorithm:

1. integrate the equations of motion,
2. assign charge and current to the mesh,
3. solve the field equations on the mesh,
4. interpolate the fields from the mesh to the particle locations.

Fig. 1 shows a simplified picture of these steps. We have to consider the phase factor in all these steps. In this section we will summarize how the equations change. The actual implementation is discussed later.

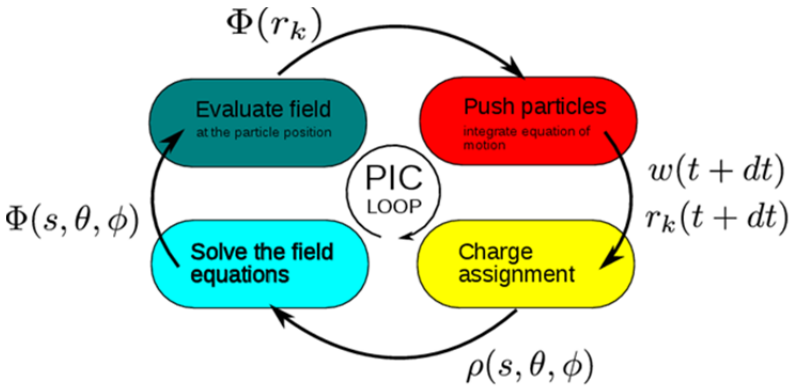


Fig. 1 Four steps of the PIC loop

2.3.1. **Weight equation**

The equations of motion are integrated to push the particles into a new position for the next time step. For the space and velocity coordinates, the equations do not change after introducing the phase factors. The equation for the weight evolution does change. The original weight evolution equation can be written in the following form

$$\frac{\mathrm{d}w_k}{\mathrm{d}t} = G(\vec{r}_k, v_{\parallel k}, \mu_k, t),$$

where *G* is a source term, and *k* is the particle index. As the weights are dependent on the phase factor, this equation has to be changed to the following form

$$\frac{\mathrm{d}\tilde{w}_k}{\mathrm{d}t} = \frac{1}{S}\frac{\mathrm{d}w_k}{\mathrm{d}t} - \tilde{w}_k\frac{1}{S}\frac{\mathrm{d}S}{\mathrm{d}t} = \tilde{G}(\vec{r}_k, v_{\parallel k}, \mu_k, t) - \tilde{w}_k\frac{\mathrm{d}\vec{r}}{\mathrm{d}t}\cdot(im\nabla\theta - in\nabla\phi).$$

2.3.2. **Charge assignment**

Charge assignment is the process of assigning the charge or current represented by the particles onto the grid, so these values can be later used as a source term in the field equations. In principle the following integral is calculated

$$\varrho(\vec{x}) = \sum_\nu q_\nu \int \mathrm{d}^6 z f_\nu(z,t)\delta(\vec{r}+\vec{\rho}-\vec{x}).$$

Here ϱ is the charge density, the 0[th] moment of the distribution function in velocity space. Because of the spline discretization, we have to project the density to the elementary spline functions, which means that the following integral has to be evaluated

$$\tilde{\varrho}_k = \int \varrho(\vec{x})\Lambda_k(\vec{x})S^*(\vec{x})\mathrm{d}^3x = \sum_\nu q_\nu \int \mathrm{d}^3x \int \mathrm{d}^6z f_\nu(z,t)\delta(\vec{r}+\vec{\rho}-\vec{x})\Lambda_k S^*.$$

Here the star denotes complex conjugation. The integral over the angular coordinate in velocity space is the gyroaverage, and it is approximated with an *N* point average over the phase angle α. As the distribution function is discretized with markers, the integral corresponds to the following sum

$$\tilde{\varrho}_k = \sum_\nu \sum_j \sum_a q_\nu \tilde{w}_j \Lambda_k(\vec{r}_j + \vec{\rho}_j(\alpha_a))S(\vec{r}_j)S^*(\vec{r}_j + \vec{\rho}(\alpha_a)).$$

Since $S^* = S^{-1}$, the phase factor almost entirely drops out, there is only a small phase shift remaining.

2.3.3. **Field solver**

The electric and magnetic fields are determined by the Poisson equation and Ampère's law. In NEMORB, these field equations are discretized with a finite element method using B-splines as finite elements. Matrices are constructed that represent the differential equations as linear algebraic equation systems. The matrices are built only once in the initialization phase and the field equations are solved always with the same matrix, because the coefficients of the differential equation do not depend on time.

In NEMORB, we use a long-wavelength approximation, assume quasineutrality and (in the simplest case) assume adiabatic electrons, so we can write the equation for the electric potential into the following form

$$\frac{en_0}{T}\Phi - \frac{n_0}{B\Omega_i}\nabla_\perp^2\Phi = \rho.$$

The discretization is done using the Galerkin method, using the splines as weight functions. The differential equation is transformed to the following weak form

$$\int \mathrm{d}^3x \left(\frac{en_0}{T}\Lambda_j\Lambda_k + \frac{n_0}{B\Omega_i}\nabla_\perp\Lambda_j\cdot\nabla_\perp\Lambda_k\right)\Phi_k = \int \mathrm{d}^3x\Lambda_j\varrho.$$

This is how the discretization is done in the original code, but when we introduce the phase factors, then the weak form changes. To preserve the Hermitian property of the matrix, the weight function is chosen as $\Lambda_j S^*$

$$\int \mathrm{d}^3x \left(\frac{en_0}{T}[\Lambda_j S^*][\Lambda_k S] + \frac{n_0}{B\Omega_i}\nabla_\perp[\Lambda_j S^*]\cdot\nabla_\perp[\Lambda_k S]\right)\tilde{\Phi}_k = \int \mathrm{d}^3x\Lambda_j S^*\varrho\,.$$

This corresponds to the following algebraic equation

$$\tilde{A}_{jk}\tilde{\Phi}_k = \tilde{\varrho}_j,$$

where matrix $\tilde{A}_{jk}$ represents the discretized equation, $\tilde{\Phi}_k$ are the unknown coefficients of the field, and $\tilde{\rho}_j$ is the source term introduced in the charge assignment section. The expression for $\tilde{A}_{jk}$ contains the derivative of the splines; for these, we have to include the phase factors. The derivative in perpendicular direction is approximated with the derivatives in the poloidal plane

$$\nabla_\perp[\Lambda S] \approx \nabla s\frac{\partial[\Lambda S]}{\partial s} + \nabla\theta\frac{\partial[\Lambda S]}{\partial\theta} = \left(\nabla s\frac{\partial\Lambda}{\partial s} + \nabla\theta\left[\frac{\partial\Lambda}{\partial\theta} + im\Lambda\right]\right)S.$$

Therefore, we only need to take into account the poloidal phase factor *m.* So, after the phase factor is introduced, the matrix element is calculated in the following way

$$\tilde{A}_{jk} = \int \mathrm{d}^3x \left(\frac{en_0}{T}\Lambda_j\Lambda_k + \frac{n_0}{B\Omega_i}\left[\nabla_\perp\Lambda_j - i\Lambda_j m\nabla\theta\right]\cdot\left[\nabla_\perp\Lambda_k + i\Lambda_k m\nabla\theta\right]\right).$$

The weak form of the equation contains only first order derivatives.

2.3.4. **Field evaluation**

After solving the equations, the fields have to be evaluated at the particle positions. To evaluate the electric field, the code computes first the gradient of the potential. This gradient acquires an extra contribution from the phase factor, according to the following formula

$$\nabla\Phi = \nabla(\tilde{\Phi}S) = S\nabla\tilde{\Phi} + \tilde{\Phi}\nabla S = S\nabla\tilde{\Phi} + i\tilde{\Phi}(m\nabla\theta - n\nabla\phi)S.$$

We have to consider that we need the gyro-averaged field in the code

$$\left\langle \vec{E} \right\rangle (\vec{r}) = \langle \tilde{\vec{E}} \rangle (\vec{r})S(\vec{r}) = \sum_{a=1}^{N_{avg}} \nabla\Phi(\vec{r} + \vec{\rho}_a).$$

Using the relation for the gradient of the potential, we can see that the gyro-averaged electric field becomes

$$\langle \tilde{\vec{E}} \rangle = \sum_{a=1}^{N_{avg}} \left[\nabla\tilde{\Phi}(\vec{r} + \vec{\rho}_a) + i\tilde{\Phi}(\vec{r} + \vec{\rho}_a)(m\nabla\theta - n\nabla\phi)\right] S(\vec{r} + \vec{\rho}_a)S^{-1}(\vec{r}).$$

2.3.5. **Diagnostics**

Apart from the four steps in the PIC loop, there is an additional step when we collect information about the result. After introducing the phase factors, we represent the physical quantities with complex functions, but only the real part has physical meaning. We have to consider this when we calculate the diagnostics. The most important diagnostics are the field energy and the energy transfer.

The field energy is calculated by the following integral

$$\mathcal{E}_{\mathrm{field}} = \int \mathrm{d}^3x \mathrm{Re}(\Phi)\mathrm{Re}(\rho) = 1/4 \int \mathrm{d}^3x(\Phi + \Phi^*)(\rho + \rho^*).$$

We introduce the phase factors and use the fact that $S^* = S^{-1}$

$$\mathcal{E}_{\mathrm{field}} = 1/4 \int \mathrm{d}^3x[\tilde{\Phi}\tilde{\rho}S^2 + \tilde{\Phi}^*\tilde{\rho}^*S^{-2}] + 1/2 \int \mathrm{d}^3x \mathrm{Re}(\tilde{\Phi}^*\tilde{\rho}).$$

The *S* function is periodic, therefore, the first term disappears if the mode that we are simulating does not couple to the (*m*=0, *n*=0) mode. If the mode of interest couples to the zeroth mode, then we should not use the phase factor. Otherwise we can define the energy integral in the following way

$$\mathcal{E}_{\mathrm{field}} = 1/2 \int \mathrm{d}^3x \mathrm{Re}(\tilde{\Phi}^*\tilde{\rho}).$$

The other important diagnostic is the energy transfer between the particles and the field. This is calculated as the work done by the electric field on the particles

$$\vec{j}\cdot\vec{E} = \sum_i q_i \int \mathrm{d}^6z\vec{v}_{gc}f_i(z,t)\cdot\langle\vec{E}(x,t)\rangle = \sum_{i,k} q_i \mathrm{Re}(\tilde{w}_k^i S)\vec{v}_{gc}\cdot\langle\mathrm{Re}(\tilde{\vec{E}}S)\rangle.$$

Here *i* is the particle species index and v_{gc} is the guiding center velocity. The phase factor is restored explicitly before the sum is calculated.

2.4. *Implementation*

In this section we discuss how the changes were implemented in the code. A preprocessor switch was introduced to turn the phase factor mode on/off at compilation time, and set the type of the influenced variables accordingly. The modifications that are necessary for the equations are located in performance critical places in the code. Care has been taken to implement the changes in a way that does not affect the performance for normal (non-phase factor extracted) operation of the code.

To ensure that the proper linear form of the equations is used when we include the phase factors, a new function has been added to check whether the correct switches are set for a linear calculation. We also have to switch off zonal flows, since by introducing the phase factors we practically shift the mode of interest into the (0,0) position, which is treated differently from the other modes if zonal flow calculation is turned on.

The implementation of the PIC loop is actually more complicated than Fig. 1 suggests. Fig. 2 shows the subroutines called during one step of the PIC loop. The places where most of the changes were implemented are highlighted with red circles. In the next subsection the modifications are discussed in more detail.

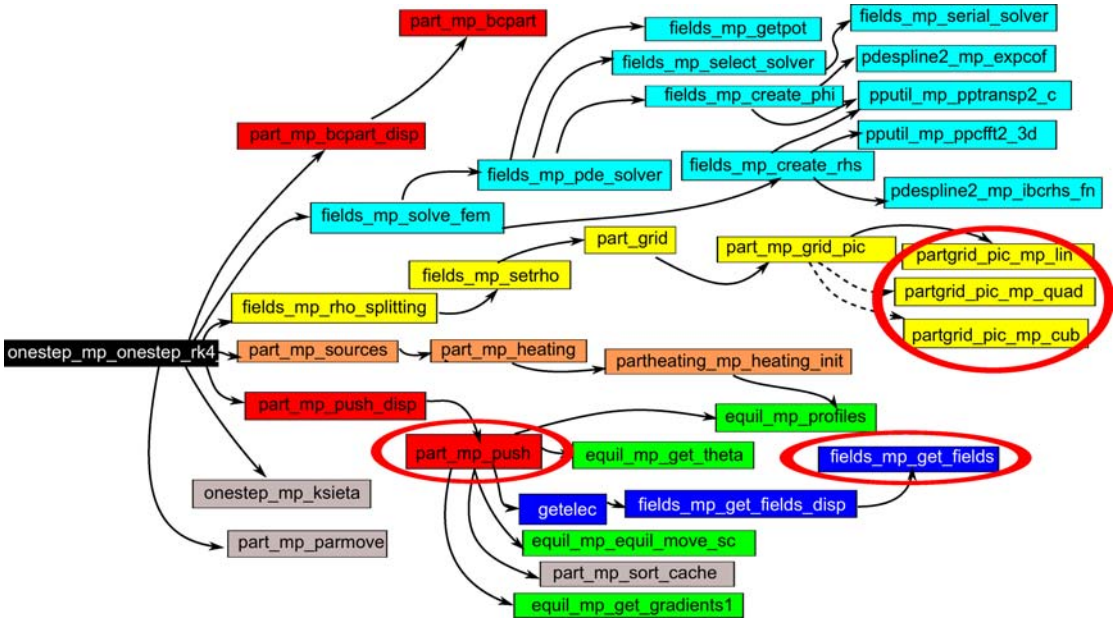


Fig. 2 Subroutines called in the PIC loop: the red circles highlight subroutines where most of the modifications were necessary**.**

2.4.1. **Weight equation**

The modified weight equation has been implemented in the PUSH subroutine that calculates the time derivatives for the particle motion.

In NEMORB, the magnetic coordinates are used to describe the particle position almost everywhere, except close to the magnetic axis, where this could lead to numerical instability. Close to the axis (s<s_push) a pseudo Cartesian grid is introduced in the code, and the particles are pushed using these Cartesian coordinates [1].

In the modified weight equation, the extra *dS/dt* term causes a problem around the magnetic axis: even a small change in the position can cause a large change in the phase. This is independent of the coordinate system that we use to describe the particles; it is an inherent problem caused by the definition of the phase factor function. It can lead to numerical instability during time integration.

We have to note that it is not necessary to resolve the tiny structures around *s*=0, the modes of interest are generally located outside s_push. Moreover, the gyro-averaging would prevent too small structures evolving around *s*=0. That is why we can switch back to using the original weight, if the particle is inside s_push. For these particles, we restore the phase factor, and solve the original weight equation. This way the weight equation is numerically more stable. Fig. 3 shows the two coordinate systems.

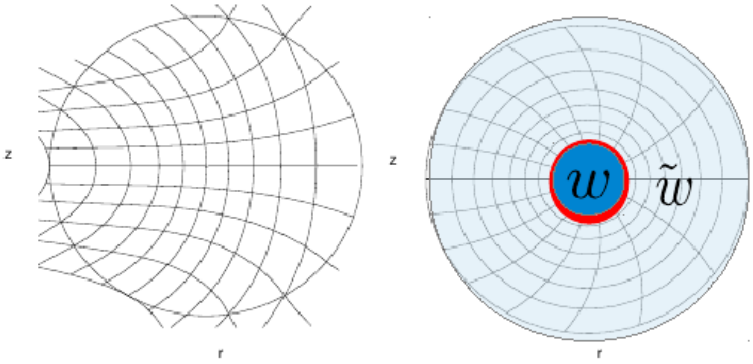


Fig. 3 Left: pseudo Cartesian coordinates, right: magnetic coordinate system. The red circle denotes the s_push, inside this circle the pseudo Cartesian coordinates are used together with the original weight *w*, outside s_push the magnetic coordinates are used and the weight is represented by $\tilde{w}$**.**

The KSIETA subroutine in the ONESTEP module is responsible for changing the coordinate system if the particle is inside s_push. This subroutine was modified to change the weight too for the affected particles.

One other possible solution would be to change the time integrator, and use a simpler second order scheme. In that case the phase factor part can be analytically integrated using the operator splitting method. This possibility was not explored in detail, since the Runge-Kutta scheme provided adequate results.

2.4.2. **Charge assignment and field evaluation**

The modifications both for the charge assignment and for the field evaluation are straightforward to implement. The spline values are interpolated from a three dimensional grid. The field is represented in the code by B-splines which can have linear, quadratic or cubic polynomial order. The spline interpolation is explicitly coded for all three orders of B-splines, the corrections had to be implemented for all orders separately. The charge assignment summation formula is evaluated in the PART_GRID subroutine, the phase factor modifications were introduced there. The modified field evaluation has been implemented in the GET_FIELDS function.

2.4.3. **Matrix building**

The code implements different methods to build the matrix and to solve the linear equations. The field is represented with splines, but the matrix can be built in a mixed Fourier-spline space. Let us denote the Fourier transformation operator with $\mathcal{F}$ to rewrite the matrix equation with the Fourier transformed variables

$$\underbrace{\mathcal{F}A\mathcal{F}^{-1}}_{A_{\mathcal{F}}} \cdot \underbrace{\mathcal{F}\Phi}_{\Phi_{\mathcal{F}}} = \underbrace{\mathcal{F}\varrho}_{\varrho_{\mathcal{F}}} .$$

In the most recent version of the field solver, a Fourier transformation is used in both toroidal and poloidal direction, and the $A_{\mathcal{F}}$ matrix is built. This allows filtering out irrelevant modes and reducing the problem size. The method is described by McMillan *et al.* [2]. The phase factors contributions have been implemented for this solver.

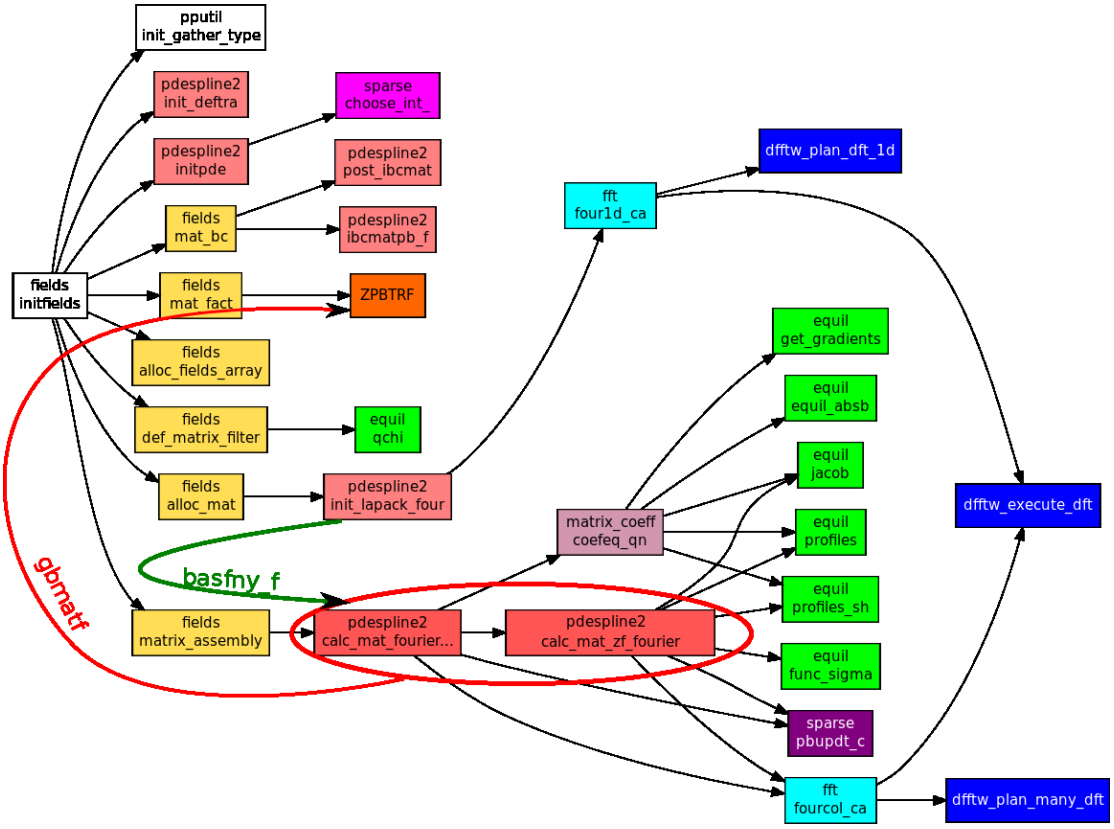


Fig. 4 Subroutines called during field initialization. The subroutines where the matrix elements are calculated are highlighted with the red circle. The matrix building uses precalculated spline values that are stored in the BASFNY_F array. The resulting matrix GBMATF is passed to the linear solver routines.

Fig. 4 shows the functions called during the initialization of the field solver. The matrix elements are calculated in the subroutines CALC_MAT_FOURIER. This function uses the values of the splines and their derivatives that are evaluated in advance by the subroutine INIT_LAPACK_FOURIER. The precalculated values are stored in the BASFNY_F array. The phase factors are introduced for the derivatives of the splines, inside the INIT_LAPACK_FOURIER subroutine.

Fortunately, the matrix building and the equation solver already use complex variables. Because the inclusion of the phase factors does not change the Hermitian property of the matrix, no change is necessary in the solver. The same way as in the original code version, the Cholesky decomposition of the matrix is calculated during the initialization phase, and the solution for the field is determined by back substitution during the simulation.

2.4.4. **Complex numbers**

The phase factor *S* is a complex quantity, therefore, the equations become complex, and it is necessary to use complex variables in the code. As we introduce the phase factor, the weight and the field quantities have to be stored using complex variables. There are several other variables that also need to be redeclared as complex, like different terms in the weight equation and variables that store the charge or the current. The whole call-graph of the matrix building, of the field solver and of the field evaluation was investigated and the variable types were changed to complex where it was necessary. The modifications affect several function arguments, global variables in the field module and temporary variables within the subroutines.

2.4.5. **Diagnostics**

Before writing the diagnostic output, the phase factors are restored inside the Fortran code. This way there is no need to change the Matlab diagnostic routines.

The changes in the definition of the field energy and energy transfer diagnostics were discussed earlier. The fluid moments are calculated by a similar sum as the $j \cdot E$ diagnostic. The phase of the weight is restored before doing this diagnostic, this way only a minimal change is required in the diagnostic routines

The values of the potentials are also saved on a grid. The phase factor is restored, and the real part is taken before writing the output.

The spectral diagnostics is not implemented for the phase factor version of the code. This diagnostic makes heavy use of the Hermitian properties of the Fourier transform of the real data, which is lost because of the phase factor. After discussion with the principal investigator, we decided not to implement the spectral diagnostics for the phase factor version of NEMORB, since the spectral information can be recovered from other 3D diagnostic information.

2.4.6. **Adjusting the filter**

In the original version of NEMORB, the electric field is a real quantity, and its Fourier spectrum is symmetric (Fig. 5). The code actually solves only for the positive part of the spectrum, and the negative part is restored using the Hermitian symmetry.

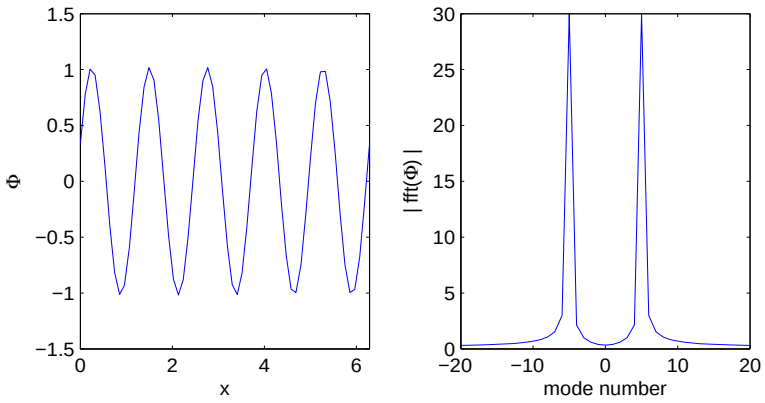


Fig. 5 Oscillating real signal and its Fourier transform.

After introducing the phase factor, the mode spectrum is shifted. This is illustrated in Fig. 6. The $\tilde{\Phi}$ signal is complex, therefore, we cannot restore the negative part of the spectrum, as it was done in the real case. Instead, we apply a Fourier filter and simulate only the positive part of the spectrum.

The filters in NEMORB were modified to include the shift caused by the phase factor. The negative part of the spectrum is not restored when we use a phase factor. This way there is a qualitative difference between the complex and the real simulation: we simulate only half of the mode spectrum. So the total field energy is less in the complex (phase factor extracted) simulation than in the original simulation. Accordingly, the energy transfer rate is also lower. Still, the quantitative result of the simulation, the growth rate of the instability does not change, as it depends only on the ratio of the energy transfer and the field energy.

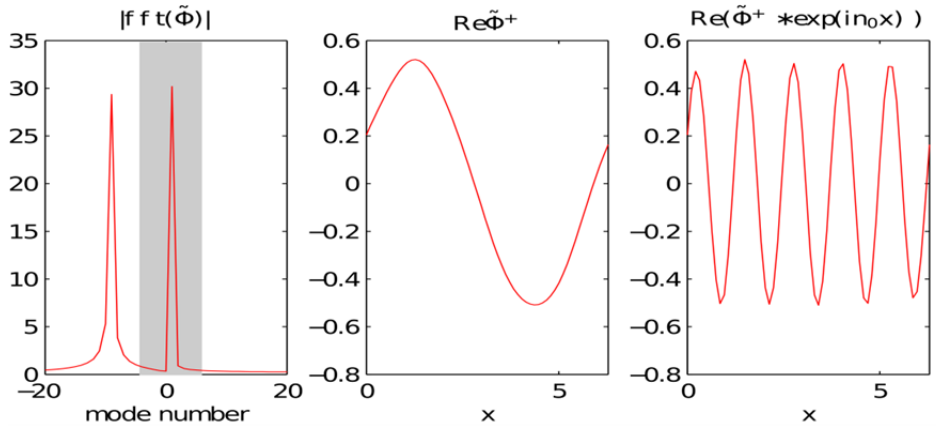


Fig. 6 After the phase factor is introduced, the mode spectrum will be shifted (left figure). The shaded area shows the filter, we keep only half of the spectrum. After the filtering the phase factor extracted signal (middle) is smoother then the original signal (shown in Fig. 5). When we restore the phase factor, we get back the oscillating signal (right). Note that the amplitude is only half of the original signal (compare to Fig. 5) because we have filtered out half of the mode.

2.5. *Using NEMORB with phase factor*

To compile the modified code one has to define the compilation switch USE_PHASEFACTORS in the makefile. The modified NEMORB code has new input parameters: phasefactor_m and phasefactor_n, which can be defined in the Fields namelist in the input file. By default these have zero value.

Since the toroidal modes in linear tokamak calculations do not couple to each other, usually only one toroidal mode is simulated by setting *nfilt1=nfilt2*. It is recommended to set the Phasefactor_n variable to the same value as *nfilt1* in this case.

The simulation can be run exactly the same way as in the old code, and the results can be processed with the same diagnostics tools.

The phase factor that was actually used in the simulation is also saved in the outptut HDF5 file.

2.6. *Tests*

After all the changes were implemented, the testing of the code could start. The modified code was compiled and tested on the HPC-FF and Helios computers.

The test case used in this section is a simple electrostatic ITG (Ion Temperature Gradient driven) test case in an analytical tokamak equilibrium. The toroidal filter was set to choose a single toroidal mode *nfilt1=nfilt2*=6. The results from the original code were compared to the results of the modified code.

In the first test, the phase factor was switched off with the compiler switch. The results agree up to machine precision with the original code.

In the next test, the compiler switch was set to compile the phase factor modifications into the code. Tests were done with setting both the toroidal and poloidal components of the phase factor to zero. This way we would test the changes of using complex variables instead of reals. The comparison is not completely trivial. As we have discussed in the previous sections, we are simulating only half of the mode, so the total energy will be different (the amplitude is half, the energy is one fourth of the original). But the growth rate calculated from the ratio of the total energy transfer and the field energy agrees well with the original code, as it is shown in Fig. 7. In the initial phase we can see a difference, but after the mode starts to grow, then the results of the two simulations converge to the same value.

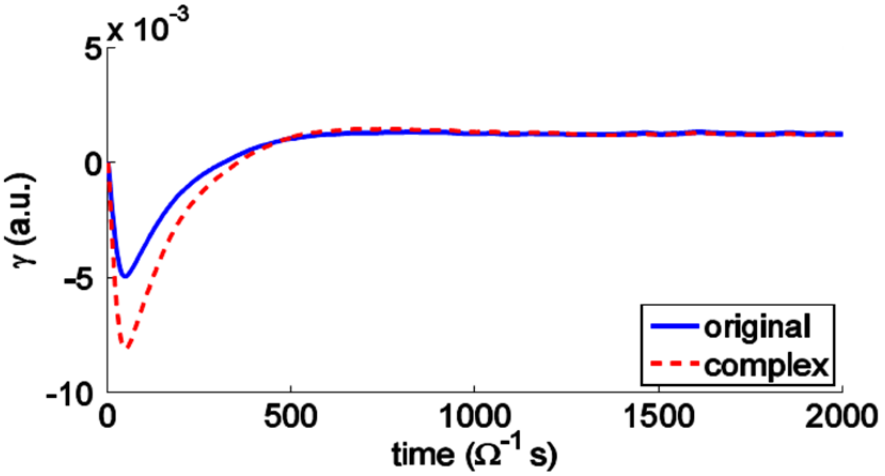


Fig. 7 Growth rate from the original code and from the modified code that uses complex numbers. The phase factor is set to zero.

In the next step, tests were made using nonzero value for the toroidal phase factor *n*. This way we can test the modifications in the weight equation and the field evaluation separately, since the matrix building and the charge assignment steps are not influenced by the toroidal phase factor. The result is shown in Fig. 8, we can see that the modified code converges to the same growth rate as the original.

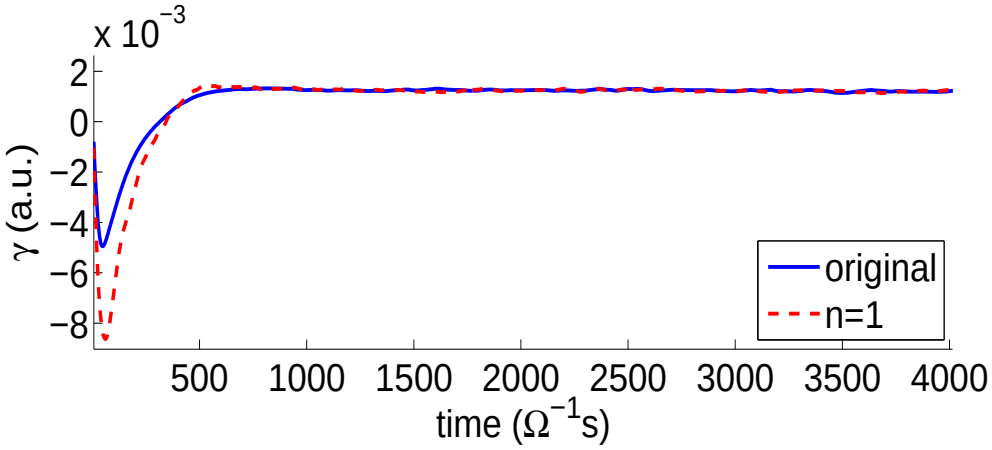


Fig. 8 Growth rate comparison between the original and the modified code, using toroidal phase factor *n*=1.

Finally, tests were made with nonzero poloidal phase factor (*m*), and different combinations of the two phase factor components (see Fig. 9). From the results we can see that the equations were modified correctly and the modified code delivers the same results as the original.

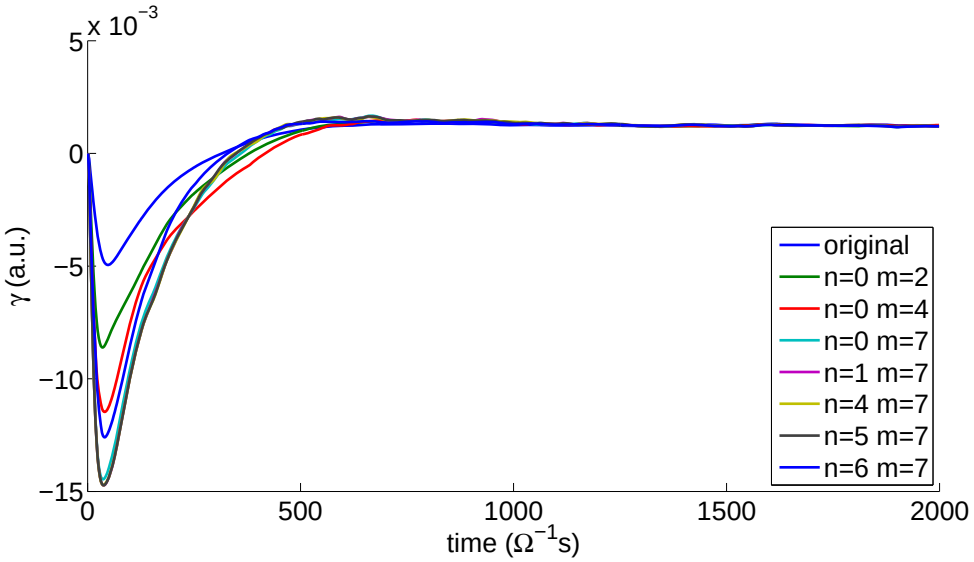


Fig. 9 Tests different phase factor values.

The poloidal structure of the simulated ITG mode is shown in Fig. 10. We can observe an *m*=7 mode in the left figure. The simulation using *m*=7 as poloidal phase factor delivers the same spatial structure (after we restore the phase factor function). On the right side we can see the structure of the $\tilde{\Phi}$ function, that is actually simulated. It is a much smoother function than Φ, which allows us to use smaller resolution for the simulation, both in the number of B-splines and in the number of Monte-Carlo particles.

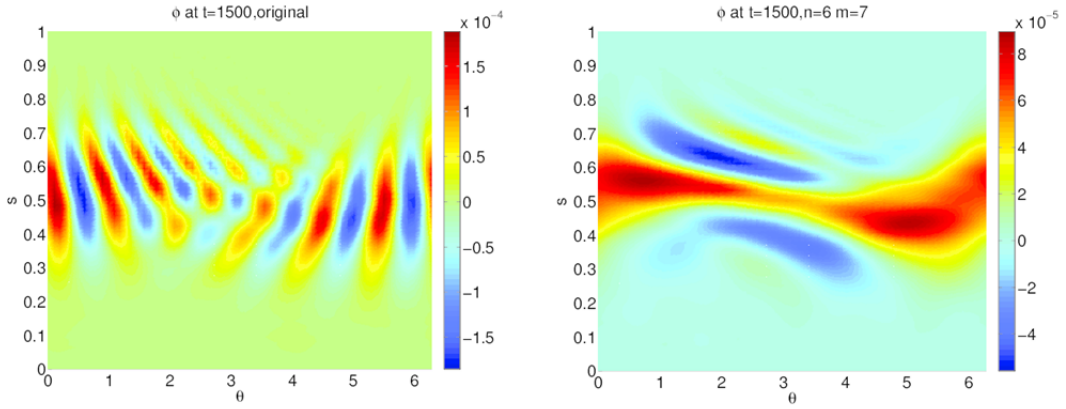


Fig. 10 Poloidal cut of the mode structure in magnetic coordinates, the fluctuations of the electrostatic potential is shown. The horizontal axis is the poloidal angle, the vertical axis is the flux surface label. Left: Φ from the original code, Right: $\tilde{\Phi}$ using phase factor *m*=7 in the modified code.

2.7. ***Resolution***

In the original NEMORB code, the spatial resolution for the field discretization has to be chosen according to the mode numbers that we intend to simulate. In Fig. 11, we can see how the results from the original code depend on the resolution. For the simulation shown in the previous figures 64x64 points were used to resolve the angular directions. The results are already converged with 32x32 points. If we decrease the resolution, then first the noise increases (using 32x16 points), then with 16x16 points, the mode will not be properly resolved.

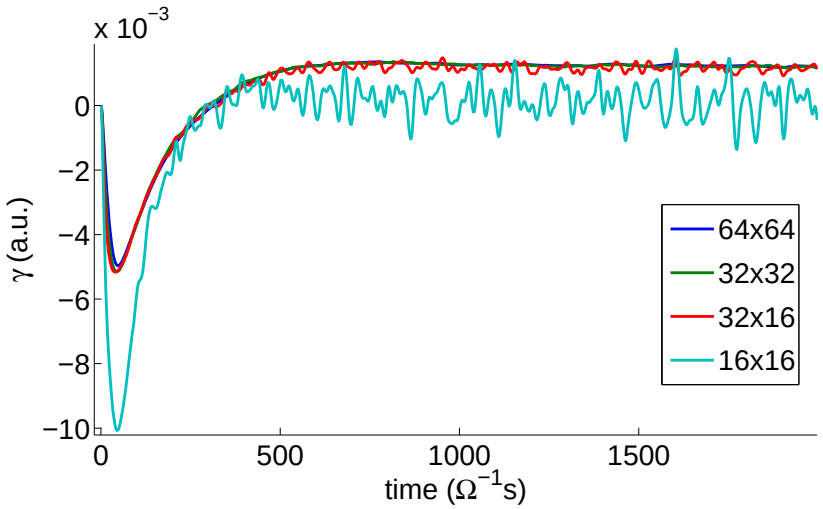


Fig. 11 Simulations with the original code using different resolutions (poloidal x toroidal).

In the phase factor version of the code, the resolution requirements are relaxed for linear simulations. The resolution test was repeated using phase factors n=6, m=7, and the results are shown in Fig. 12. We can see that we need much smaller spatial resolution to resolve the mode.

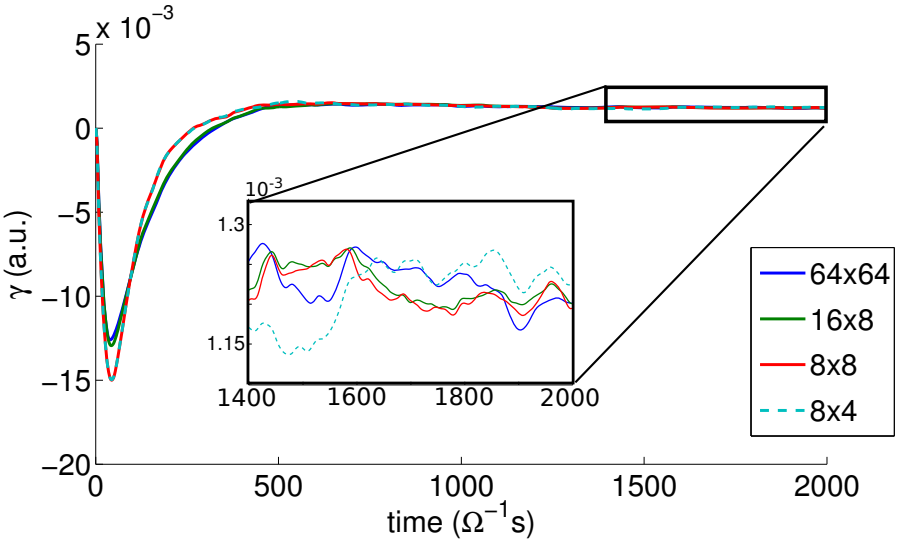


Fig. 12 Simulations using phase factor n=6, m=7, with different resolution.

For the original code, higher mode numbers require higher resolution in general. If we use a phase factor, then the resolution depends only on the number of modes that we have to keep inside the filter, which is usually a low number. This way we can save memory and decrease communication time which can lead to faster calculations.

2.8. *Summary*

The aim of the EMPHORB project was to implement the phase factor transformation in the NEMORB code. The phase factor transformation can be used to decrease the grid resolution for linear calculations and thereby lower the amount of computational resources needed for the simulation. The implementation of the phase factor involves small changes that extend throughout the whole NEMORB code.

The phase factor was successfully implemented in the code. The code structure was analyzed and the phase factor was implemented for the main steps in the particle-in-cell (PIC) algorithm. This includes the matrix building routines, the field evaluation, and the charge and current assignment. As the equation of motion modifies only the weights, the phase factor had to be introduced for pushing the weights.

With the introduction of the phase factor, several variables had to be changed from real to complex. The call-graph for the code was investigated to check which functions are influenced by the change of the variable types, and the necessary changes were implemented. The diagnostic methods have also been updated for the use of complex variables.

Finally, the modified code was tested simulating electrostatic ITG modes. Several tests were made with different phase factor values, and the results of the modified simulation agree well with the original code.

The modifications introduced during the EMPHORB project have been analyzed with the Forcheck tool, to ensure consistency of the modified code.

2.9. *References*

[1] S. Jolliet, *Gyrokinetic Particle-In-Cell global simulations of Ion-Temperature-Gradient and Collisionless-Trapped-Electron-Mode turbulence in tokamaks*, PhD thesis, EPFL (2009)

[2] B.F. McMillan, S. Jolliet, A. Bottino *et al.*, *Rapid Fourier space solution of linear partial integro-differenial equations in toroidal magnetic confinement geometries*, Computer Physics Communications **181**, p. 715–719 (2010)

3. Final report on HLST project MICPORT

3.1. *Introduction*

Accelerator cards, like GPU systems from NVIDIA or the Intel MIC architecture, are gaining more and more recognition in the field of high performance computing. They offer high peak performance combined with high energy efficiency. Many computing facilities, including IFERC-CSC are already planning to extend their computational resources by accelerator cards. To exploit the capabilities of such systems is not a trivial task, the applications have to be adapted to the new hardware. The aim of the MICPORT project was to port the JOREK code to the Intel MIC architecture in order to gain experience with the new hardware and its software stack.

While it was easy to get the code running on the new hardware, the ported code does not perform well on the MIC architecture. To make effective use of the MIC hardware, the code has to scale up to hundreds of threads, and it should utilize the vector processing power of the mathematical units. Both of these points provide a challenge in case of the JOREK code. The vector performance and the scaling of the JOREK code is assessed on the new hardware. It is identified that significant improvement in vectorization is possible. The linear algebra library used in the code is also tested on the MIC architecture. For this library further improvement is needed before we can utilize the full performance of the new hardware. In the following, after introducing the hardware and the basic steps of porting the code, we discuss the different main parts of the code, their scaling on the MIC architecture and steps that are needed to improve their execution time.

3.2. *The hardware*

Access was granted to the MIC test platform at Rechenzentrum Garching (RZG). The system consists of two hosts with Intel Xeon E5-2670 processors (Sandy Bridge architecture) and two Xeon Phi 7120 accelerator cards (MIC architecture) in each host. Markus Rampp has explained how to access the test platform, provided training material for the new hardware, and helped in the various problems that occurred while using this system.

The MIC abbreviation refers to the Intel Many Integrated Core Architecture. Intel's first product with MIC architecture is the Xeon Phi coprocessor. In this report, these terms are used interchangeably: when we mention MIC (or accelerator, or coprocessor) then we specifically refer to the Xeon Phi 7120 coprocessor that was used for testing. To describe codes running on such an accelerated system it becomes necessary to introduce some terminology. The computer with the accelerator cards is referred as host. Host calculation means that only the main CPUs of the host are utilized, and the accelerator cards are inactive. Such calculations will serve as baseline in this study. The so called native execution means that the program runs only on the cards and the host CPUs are not utilized. There can be several models which utilizes both the host and the card CPUs. In this report, we focus on the comparison of host and native calculations. If the performance would be similar, then it would be easy to combine the two modes into a symmetric host-accelerator calculation mode.

In this report we will compare the performance of the host CPUs to the Xeon Phi accelerator card. To make such a comparison it is important to know about the differences between the two architectures and also about their theoretical peak performance. In Table 2, we compare some of the key characteristics. The host has two processors with eight cores each; one Xeon Phi coprocessor contains 61 cores and is capable of executing 244 threads simultaneously. The host processor has more than double clock frequency, while the Xeon Phi has more cores and wider vector registers. Ideally, each core can execute two floating point operations (one multiply and one add) at every clock cycle. The theoretical peak performance is calculated by multiplying the values in columns 2–5 together.

	Freq	cores	Size of vector registers	Vector operations / core / cycle	Theoretical peak perf.
Xeon E5-260 (Host)	2.6 GHz	8	4 doubles	2 (execution units)	166.4 GFlop/sec
Xeon Phi 7120 (MIC)	1.238 GHz	61	8 doubles	2 (fused multiply-add)	1208.3 GFlop/sec

Table 2 Host CPU and Coprocessor comparison

The Xeon Phi has higher theoretical peak performance, but it is more difficult to attain it. Each core has to execute at least two threads to keep the instruction pipeline busy, which means that the code should scale at least up to 120 threads. The Xeon Phi has in-order execution, while the host CPU has the additional freedom to execute instruction out of order to maximize performance.

There is also a difference of the memory size and bandwidth that is available on the two systems. A single Xeon Phi 7120 coprocessor has 16 GB of memory with 160 GB/s peak memory bandwidth. On the host, each CPU is connected directly to a memory bank with 32 GB memory and 40 GB/sec transfer rate. Both CPUs are able to access both memory banks, and together they form a shared memory node with 64 GB total memory and 80 GB/sec peak bandwidth.

It is important to keep these differences in mind, when we compare the performance of the two architectures. If we compare the performance of single cores, then the cores in the host are more powerful. One could decide to compare a single processor (8 cores) with one accelerator card. This is considered as a fair and common comparison, because the test system has two CPUs and two Xeon Phi accelerator cards. A different comparison can be made on the basis that the two CPUs of the host have access to the same shared memory and thus, form a node. The Xeon Phi coprocessors can be also considered as two separate nodes, and we could compare these to the original host node. In this work, we will make such node to node comparisons: unless otherwise noted, we will compare the performance of the host with 2 CPUs to the performance of a single Xeon Phi card. This means also, that the results could be a factor of two better for the Xeon Phi if we would compare it to only a single multicore CPU.

3.3. *Porting JOREK*

3.3.1. **Installation of libraries**

The JOREK code uses the PaStiX (Parallel Sparse Matrix) library to solve linear systems. PaStiX is used to LU factorize the matrix corresponding to the linearized equations of JOREK. The PaStiX library depends on the Scotch package to calculate the matrix ordering (permutation of the matrix to reduce fill-ins). Both libraries were installed on the host machine. The libraries are compiled twice, once in normal mode for execution on the host and once in native mode for the cards. To compile the code for native mode execution, one has to do cross compiling on the host, because there are no compilers available on the card. One has to give the option `-mmic` to the compiler and linker for native mode compilation. The only problem is that both libraries use a small program to generate header files with size information for different variable types. This program is compiled and automatically invoked by the make process. But the program compiled with the `-mmic` option cannot run on the host, so the build process stops at this point. One can copy the program to the card and run it there, or recompile it for the host. This way we can generate the necessary headers manually. This little inconvenience might be avoided if the host system is set up in a way that the native code is automatically transferred to the card for execution.

After installing the libraries, the compilation of the JOREK code was relatively simple: the `-mmic` option had to be inserted into the makefile. One has to use a recent version of the compiler to create executable for the Xeon Phi architecture, here version 14.0.1 of the Intel Fortran and C compilers was used. A slight modification of the JOREK code was necessary, because the compiler reported an error at some of the interface definitions for the PaStiX library. This problem was fixed using Fortran iso_C_bindings.

JOREK is a hybrid MPI-OpenMP code, and it requires a thread safe MPI library. The code requires at least MPI_THREAD_FUNNELED thread support. This means that every MPI process has a master thread. Each thread can spawn other threads, but only the master is allowed to make MPI calls.

3.3.2. **Thread pinning**

Before we run the code, it is important to specify how the threads/processes are assigned to the physical cores. The thread and process pinning is controlled by environment variables. The environment variable `I_MPI_PIN_DOMAIN` gives the choice of how the MPI processes are distributed. A domain is a set that contains a certain number of CPU cores. Using this variable, one can define non overlapping domains, and each MPI task will be assigned to a separate domain. The threads that are spawned by an MPI task are assigned to CPUs within the same domain as the master thread.

With two MPI process per host, `I_MPI_PIN_DOMAIN=socket` was used, this way each MPI task is assigned to a separate socket. For larger number of MPI processes, the same variable is set to `omp`, this way the number of logical CPUs in a domain is equal to `OMP_NUM_THREADS`. It is possible to further specify whether the members within the domain are allocated close to each other (`omp:compact`, which is the default value) or whether they are allocated far from each other (`omp:scatter`). Once the domains are defined, the `KMP_AFFINITY` variable can be used to control thread pinning within the domain. For calculations on the host, this value was set to `compact`.

On the Xeon Phi card, several combinations of thread/process pinning were tested. For the process pinning, on average the best option was to leave the variable `I_MPI_PIN_DOMAIN` unset. If we set `I_MPI_PIN_DOMAIN=omp:compact`, then the execution time increases by around 20%. If we choose `omp:scatter`, then the code can run up to 35% longer. For thread pinning, `KMP_AFFINITY=scatter` is used. Using compact affinity leads to slightly longer execution time. There is a third option on the MIC architecture which is called `balanced`, and it gives similar results as the `scatter` option. The numbers presented here correspond to four MPI processes, and eight threads per process, and they can vary if we change the number of threads.

3.3.3. **JOREK shell scripts**

Test runs with JOREK consist of at least two parts. In the first run, the magnetic equilibrium is calculated. In the second phase, a time dependent simulation is performed. There are shell scripts to automate this process. However, the scripts did not run on the Xeon Phi card. The card has a minimal Linux distribution and the shell does not recognize the function keyword used in the test scripts; moreover it does not invoke other scripts unless called by `/bin/sh` with the scriptname as argument. Small modifications to the scripts solved these problems.

3.3.4. **Shared libraries**

The Xeon Phi card has a virtual file system located in the memory of the card, and it has no access to the host file system. Therefore, the shared libraries that we have on the host computer are not accessible from the card. One has to manually copy (using

`scp`) the shared library files for MPI, MKL, OpenMP and for the Fortran runtime environment to the card. In the future, most probably we can avoid this inconvenience by mounting a network file system on the card through which we could access the libraries. But this has to be done by the system administrators.

3.4. *JOREK Tests*

The first test case is defined by the `util/launch_test.sh` script in JOREK, where model199 was selected. The test ran successfully on the card and on the host and gave correct results. It was also seen immediately, that the compiled code is slower on the Xeon Phi.

3.4.1. **Profiling**

The JOREK code has built in timers to measure the execution time for different stages of the computation using `SYSTEM_CLOCK` calls. A finite element matrix is constructed at every time step. This represents a set of linear equations, which is solved using an iterative method. At certain time steps, sub-matrices are LU factorized to create a preconditioner. The execution time for these three steps is reported at every time step. In Fig. 13 we can see how much time is spent in these three stages (averaged over many iterations). The data is from the benchmark folder of the JOREK repository and represents simulations running on the Helios computer using four MPI processes and eight or sixteen ranks per process. The execution time depends on the complexity of the physical model. Model199 has a simpler MHD model therefore, it is faster to construct the matrix, and relatively more time is spent in solving the linear equation system, while in the case of model303, the matrix construction is the most expensive execution stage. These ratios depend also on the resolution of the grid in the poloidal plane, as well as the number of Fourier harmonics used in the code. Since all of these three stages of computation can take a significant share of the runtime, we will study the performance for all of them.

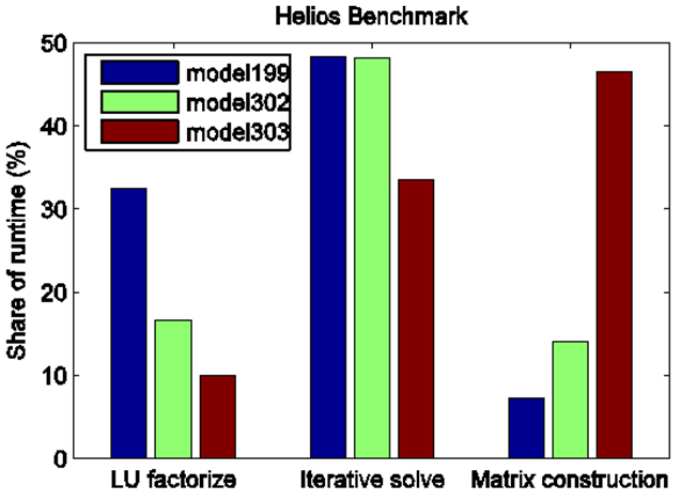


Fig. 13 Fraction of time spent in different parts of the code, for three different physical models running on Helios CPUs.

We have repeated the same test on the Xeon Phi coprocessor in native mode (four MPI processes and eight thread/process). In Fig. 14, we can see the relative execution time of these three stages. The ratio of the matrix construction and the LU factorization is different on the MIC than on the host CPU (compare to the blue bars on Fig. 13). In the next sections we will study the three stages individually.

To perform a test with physically relevant results takes a long time. For further tests, shorter benchmarks were used with only five time steps, and the average values over these five steps will be presented. All other parameters are kept the same as defined

by the `launch_test.sh` script. In the following, we will focus only on model199, because it is the fastest to execute.

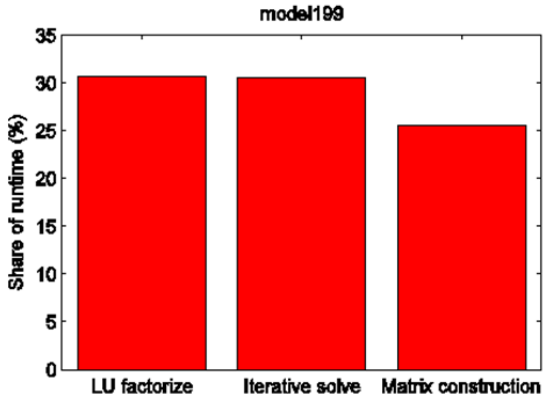


Fig. 14 Simulation on Xeon Phi in native mode, relative execution time

3.5. *Matrix building*

The code was executed first on the host, and later separately on a single Xeon Phi card using different numbers of MPI processes and OpenMP threads. The problem size was kept constant. The execution time for the matrix construction is shown in Fig. 15. The horizontal axis is the total number of threads, which is given by the number of MPI processes multiplied by the number of OpenMP threads per process (`np * OMP_NUM_THREADS`). The matrix construction scales relatively well on the host. For the test problem (model199 with three Fourier modes), at least two MPI processes are required. Using more than two MPI processes on a host increases the overhead and is less efficient.

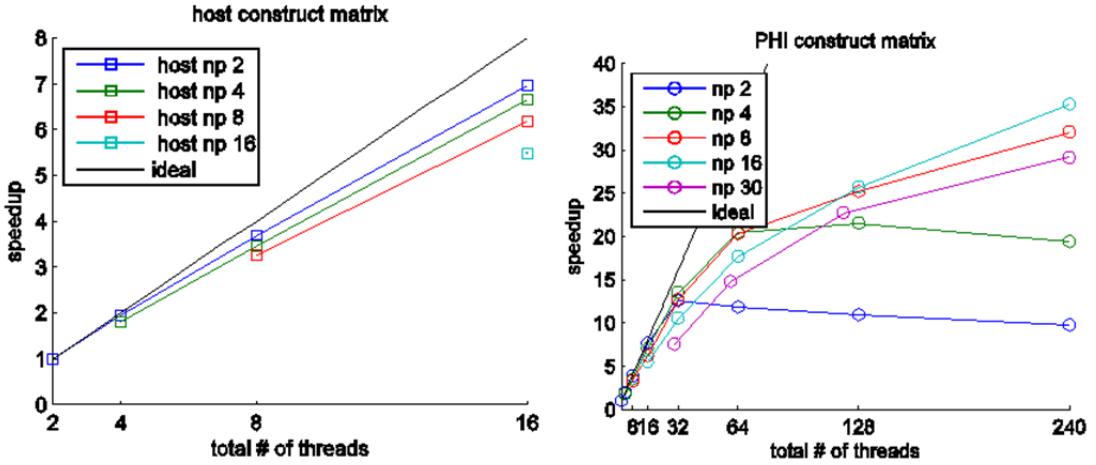


Fig. 15 Matrix construction on the host (left) and on the Xeon Phi card (right), using different numbers of MPI processes (np). The speed-up is measured relative to a calculation using two MPI processes and one thread/process, and it is normalized separately for the host and the card. The speed-up is shown as a function of the total number of threads (`np * OMP_NUM_THREADS`)

On the right side we can see the speed-up factor on the Xeon Phi card, which is normalized to the base case on the card (two MPI process, one thread each). The scaling of the matrix construction is reasonable, even though it is not so good above 32 threads. Using the maximum number of available threads the best execution time was achieved with 16 MPI tasks and 15 threads per task.

In Fig. 16, the host and native execution time is compared for the matrix construction. Using the same number of threads as on the host, the native execution is much slower. Qualitatively this is not a surprise, since the CPUs on the Xeon Phi card are

weaker then the host CPUs. But the quantitative difference is too large: using the same number of threads the card is approximately 20 times slower than the host. The key to decrease this difference is vectorization, which will be discussed in the next subsection. The best execution time using 240 threads was 1.5 times faster than the reference case (two threads on the host), but it is still 4.5 times slower compared to the fastest host simulation.

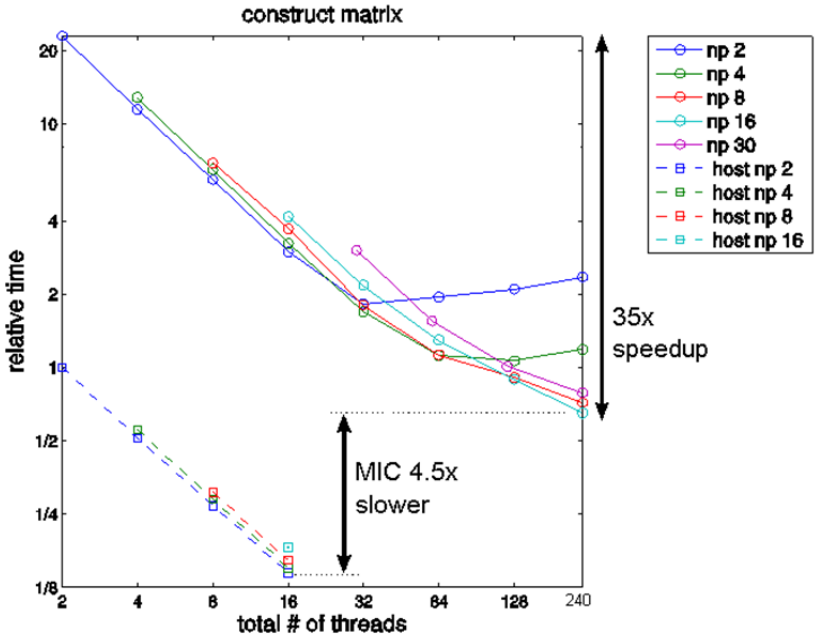


Fig. 16 Relative execution time of the matrix construction on the Xeon Phi card (solid lines) and on the host (dashed lines) using different numbers of threads and MPI processes.

The scaling on the Xean Phi is close to ideal below 32 threads, but becomes worse above this point. Using 240 threads means that one thread has around 100 rows of the global matrix. To check whether the degradation in scaling is due to the small problem size or not, the weak scaling of the matrix building is also studied. In these tests, the problem size was successively increased. The radial resolution (`n_tht * n_flux`) was increased from (8 * 21) to (128 * 81) in order to keep the problem size approximately constant for each thread. The weak scaling of the matrix construction is shown in Fig. 17. The host has a small number of threads (not larger than 16), and it scales reasonably well using two MPI processes, and it degrades with more.

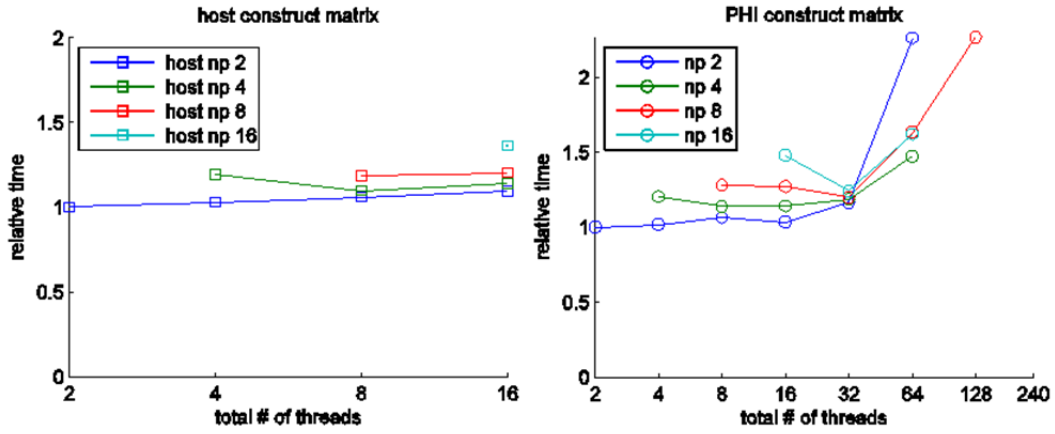


Fig. 17 Weak scaling of matrix construction. The execution time is normalized separately for the host and the card.

The scaling on the Xeon Phi card is good until 32 threads, below this number the fastest execution is achieved using two MPI processes. Similarly to the host, increasing the number of MPI processes leads to longer execution times. Above 32 threads, the execution time increases, but the exact behavior depends on the number of MPI processes. On the Xeon Phi, we need at least two threads per core to be able to feed the instruction pipeline at every cycle. Even more threads are required to mask memory latencies. On the host we have out of order execution, therefore, the instruction pipeline can be kept busy without additional virtual threads.

From the results of the weak scaling, we can see that degradation of scaling above 32 threads is not caused by the small matrix size. Another reason for the problem could be due to communication delays. But the matrix building has only minimal communication between the threads: there is only a lock when the matrix elements are updated. Instead of using a lock, it has been tried to use atomic operations to update the matrix elements. These operations are in principle cheaper than a lock, but the simulation time did not improve. Another cause for the problem in scaling could be connected to the access to the memory or the cache usage. The access to the local shared memory on the Xeon Phi is in principle faster than on the host, so it would be interesting to study whether the memory access can be improved.

Due to the limited timeframe of the project, instead of trying to improve the scaling of the matrix construction, we focused on the improvement of the single core performance. To achieve good performance on the Xeon Phi, the code should be properly vectorized, therefore, the vectorization is analyzed in the following section.

3.5.1. **Vectorization**

A hotspot analysis was performed with the VTune tool, and the `element_matrix_fft` subroutine was identified as the hotspot during the matrix construction. The compiler vectorization log showed that almost no vectorization is performed here; moreover enabling or disabling the vectorization did not have a significant effect on the execution time. A separate test case was created to be able to study and improve the vectorization of this subroutine. In this test case, we initialized the basic data structures and called the subroutine several times with slightly different input parameters.

To improve the vectorization it is necessary to change the code. In Fig. 18 we can compare the execution time for the original and vectorized code on the host and on the Xeon Phi in native mode. Here we will compare one host CPU with one accelerator card. On the host one thread/core was used (eight threads), and on the Xeon Phi the maximal number of 244 threads was used.

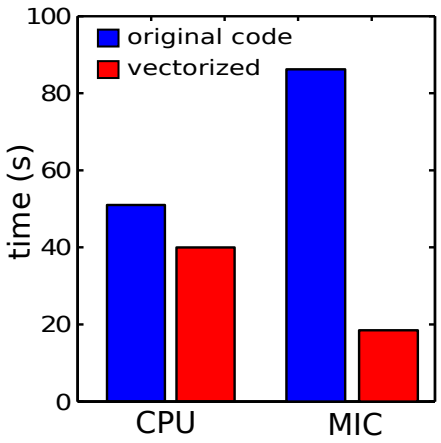


Fig. 18 Execution time for the vectorization test case

The vectorization leads to an improvement of a factor of 4.3 on the MIC architecture. The vectorized code is also faster on the host, but there it is a more modest gain (around 20%). Altogether the vectorized code is 2.16 times faster on the MIC in native mode than on a single CPU.

To understand what kind of changes are necessary for such an improvement, we have to look into the details of what the function calculates. In JOREK, the poloidal cross section of the tokamak is discretized by a quadrilateral elements. The `element_matrix` subroutine calculates the following integral for each of these elements

$$\int_V \mathrm{d}\,V a(R,Z,\phi) = \int_0^1 \mathrm{d}\,s \int_0^1 \mathrm{d}\,t \int_0^{2\pi} \mathrm{d}\,\phi J a(s,t,\phi) = \sum_{\mu=1}^{4}\sum_{\nu=1}^{4}\sum_{p=1}^{N_{plane}} C w_\mu w_\nu a(s=x_\mu, t=x_\nu, \phi_p),$$

where *s* and *t* are the local coordinates of the volume element, ϕ is the toroidal angle and *J* is the Jacobian. The integral is discretized using Gaussian quadrature, using 4x4 points in the poloidal plane. The code has actually two different versions, one where the integration is performed explicitly, as shown in the formula, and another one, which uses Fourier representation in ϕ direction, and explicit Fourier transformations. In an overly simplified picture, the integration kernel is

$$a(s,t,\phi) = X(s,t,\phi)Y(s,t,\phi) = \sum_{i=1}^{N_{vert}}\sum_{j=1}^{N_{ord}}\sum_{k=1}^{N_{vert}}\sum_{l=1}^{N_{ord}} X_{i,j}Y_{k,l},$$

where *X* and *Y* are trial and test functions in the weak formulation of the differential operator, and they are defined by a summation that runs trough the four vertices and the different polynomial orders. Altogether there are seven summations in the two formulas which are represented with seven nested loops in the code. The *XY* product hides 78 lines of complicated arithmetic. Due to this complexity, the compiler was not able to determine that the iterations are actually independent, and therefore, it did not generate a vectorized code.

To improve this situation, the loops of the Gaussian integration points were selected, and three main steps were performed to aid the compiler in automatically vectorizing these loops. First, the arrays were permuted, so that the values that correspond to different integration points (*s,t*) are stored contiguously in memory

H(i, j, s, t) → H(s, t, i, j).

Second, instead of looping explicitly through all array elements, the Fortran array notation was used to simplify the code:

A(s,t) H(s, t, i, j) → A*H(:, :, i, j).*

This way the compiler automatically generated the loops, and could easily recognize that the loop iterations are independent. There was still a problem, since we had only four Gaussian integration points in each direction. The compiler tried to generate two nested loops for the above code line and tried to vectorize only the inner one. Since four elements are not enough to fill the vector register of the Xeon Phi, the vectorization was not effective. Because of this, in a third step, the arrays had to be reshaped, by fusing the first two dimensions*: A(:) * H(:, i, j)*. The resulting code was now vectorized by the compiler.

Some of the frequently used variables are precalculated into work arrays that are allocated for every thread separately. These arrays are handled with pointers in the code. This leads to difficulties in optimization, since the compiler cannot be sure whether the arrays are contiguous, and cannot initialize with simply `memset` operations. In the vectorization example, changing to allocated arrays did not improve the initialization time. The arrays are allocated in different compilation units, so the compiler does not know whether they are aligned. Nevertheless, the size of these work arrays is known at compilation time, so they can be declared as static

arrays (actually static array components of a dynamically allocated array of derived type). This had also a large influence on the execution time.

After all these modifications, an improvement of the factor 4.3 could be achieved for the vectorized code on the Xeon Phi. In Fig. 19 we can see that the vectorization test case scales well if we increase the number of threads. There is a small degradation in scaling above 61 threads where the number of threads/core increased from one to two, and similarly above 122 threads. Using the maximum number of threads, one Xeon Phi card is 8% faster than the host with 2 CPUs. Comparing to a single processor with eight cores, the Xeon Phi is more than a factor of two faster. If the speed-up that we have gained in the separate test case applies to the full matrix construction in the JOREK code, then the matrix construction on a node with two Xeon Phi coprocessors could be three times faster then without the coprocessors. This could be achieved by treating the host and the two accelerators as three separate nodes, and distributing the work equally among them.

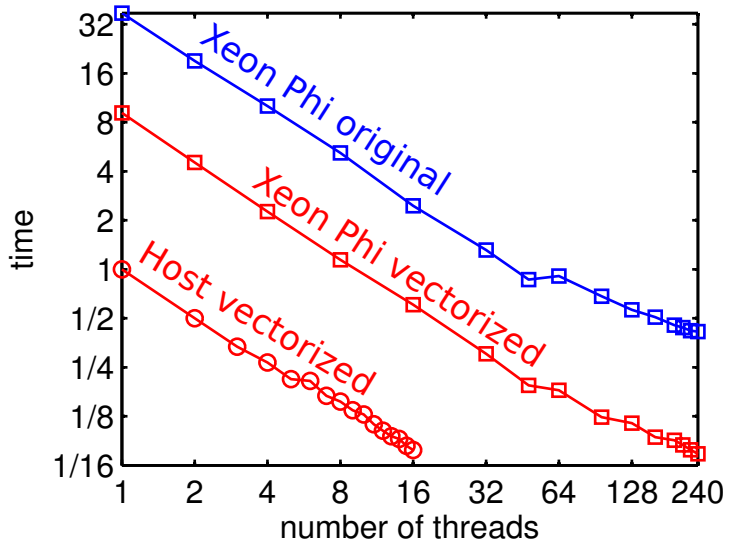


Fig. 19 Scaling of the vectorization test case

The vectorization improved the performance of the test case. To incorporate the same changes in the JOREK code is a longer task. The basic arrays that store the Gaussian integration point and weights have been permuted, therefore, it becomes necessary to change all the matrix building routines. Changes would be required in 31 files concerning around 2500 lines of code (300–400 lines per model plus 600 lines in the rest of the code). The work is more or less straightforward, but error prone, and it should be performed with careful testing. It is not possible to incorporate all these changes into the codebase during the MICPORT project. So we continue with the investigation of the factorization of the matrix. An additional report about the possibility of the vectorization will be submitted to the developers of JOREK.

3.6. *LU factorization*

After the matrix is constructed, the PaStiX library is used to LU factorize it. In Fig. 20, we can see the execution time of the factorization. It does not scale well on the host. On the Xeon Phi we have a problem, because the calculation time increases significantly when we use a large number of threads. It turned out later that it is due to a linking problem.

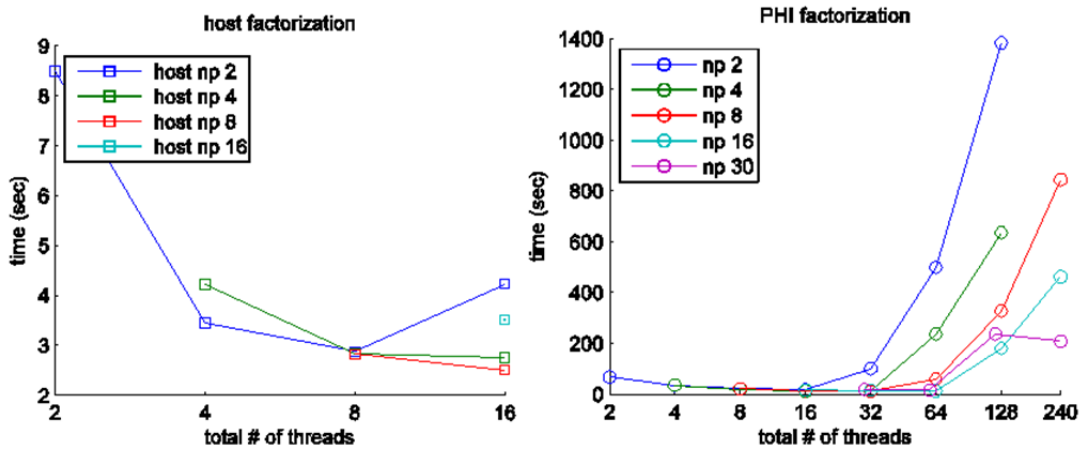


Fig. 20 Execution time of LU factorization as a function of the number of working threads

3.6.1. **PaStiX test**

To find the cause of the problem, the scaling of the PaStiX library was investigated separately using one of the example test programs that comes with the library. The "simple" example was called using the following parameters:

```
mpirun –np $nproc ./simple –lap 200000 –t $nthread
```

This example generates a Laplacian matrix of size 200,000, and calculates the LU decomposition of it. The number of fill-ins during factorization is 4.33, and the memory size of the factorized matrix is around 14 Mbytes. For some reason, increasing the matrix size resulted in a segmentation fault, but the problem size was already large enough to test the strong scaling of the factorization.

First, the "simple" example showed the same behavior as the factorization in JOREK. Further investigation revealed that linking to the wrong version of the MKL library was the cause. By mistake, it was linked to the threaded version of the library, and as a result, both PaStiX and MKL were spawning new threads, which caused a performance drop. This problem was corrected by linking to the sequential MKL library. This way the example scales well as we can see in Fig. 21, where the LU factorization time of the "simple" example is compared between the host and the Xeon Phi. Still, the example runs slower on the Xeon Phi than on the host. It seems that the PaStiX library (version 4030 is used here) is not yet completely adapted to the Xeon Phi architecture. A quick profiling with VTune shows high cycles per instruction rate, and also indicates that the cache usage is not optimal. It is also not clear whether the data alignment is properly set up within PaStiX, which is essential for good performance on the MIC architecture.

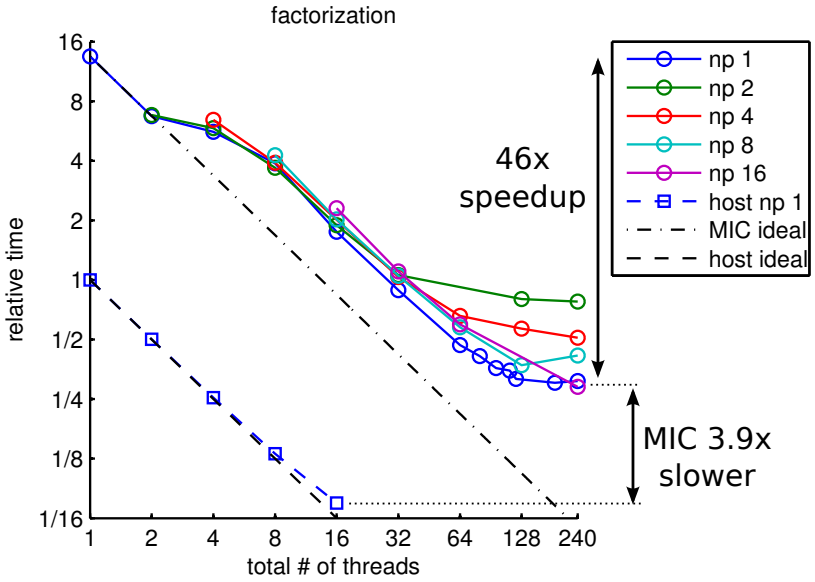


Fig. 21 Strong scaling of the "simple" PaStiX example

In principle, it could be possible to achieve better performance for the factorization. The LU factorization is implemented by other libraries too, and the examples from MKL or from the MAGMA libraries show that it is possible to achieve a good speed-up using the Xeon Phi accelerator. The PaStiX developers are also working on an improved version of the library which could work effectively on the new hardware.

3.6.2. **JOREK LU factorization**

After correcting the linking problem with the MKL library, the scaling test of the matrix factorization was repeated (Fig. 22). The factorization of the matrix used in JOREK does not scale ideally, both on the host and on the accelerator. We can see a similar factor between the MIC and the host performance as for the simple test: the Xeon Phi is a factor of three slower.

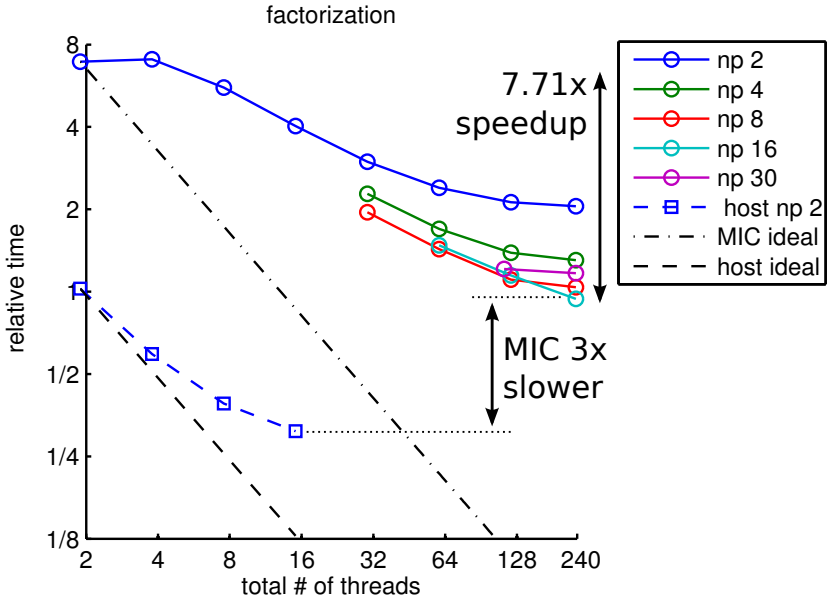


Fig. 22 Strong scaling of LU factorization in JOREK

The JOREK matrix is much smaller (2609 x 2609) than the “simple” example, but the number of nonzeros is comparable to the “simple” example. Investigating the weak scaling of the factorization we can see that the deviation from the ideal scaling is not only due to the small problem size. In fact, the weak scaling of the LU factorization (see Fig. 23) is also far from ideal. But once the PaStiX library is optimized for the MIC architecture, we could expect that the factorization time on the Xeon Phi would be competitive with the host.

During the weak scaling, we have reached a physical limit of the Xeon Phi. This happened at 64 threads. There are only 16 GB of memory available on the Xeon Phi coprocessor. We have to store the result of the LU factorization in memory, and it would require more space than what is available. The program gets killed due to memory allocation error. One would need to utilize more than one card to handle such large problems.

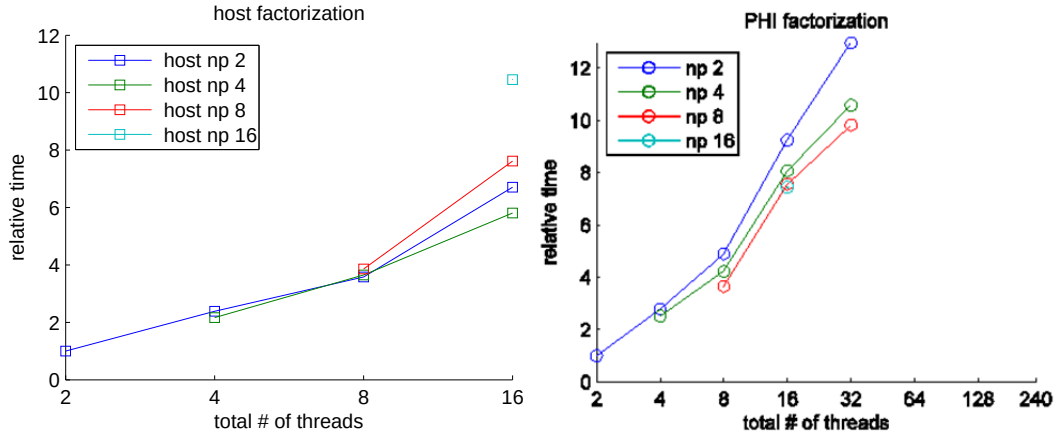


Fig. 23 Weak scaling of LU factorization in JOREK

3.7. *The solver*

In the solving step, a linear system is solved using GMRES method. JOREK uses the PaStiX library for this step too. The comparison between the host and native execution time for the solver step shows a similar picture as the comparison of the LU factorization. Fig. 24 shows the strong scaling for the solver. We can see that the Xeon Phi is 4.7 times slower in the best case.

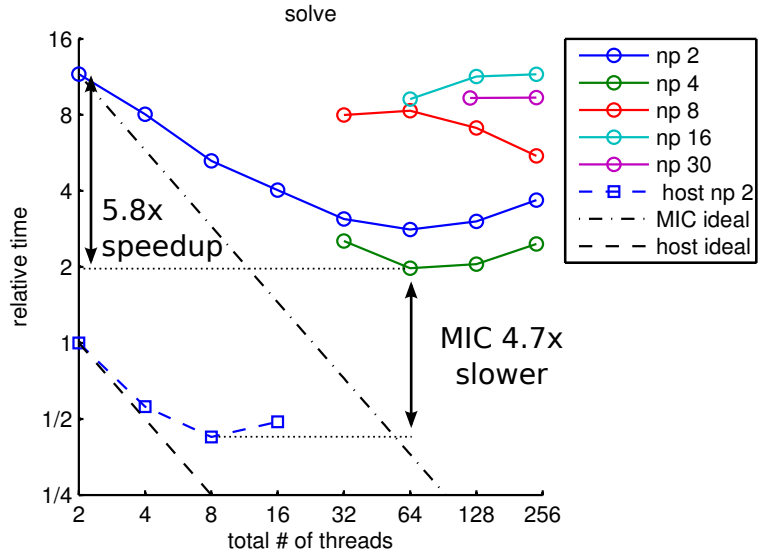


Fig. 24 Strong scaling of the iterative solver

The weak scaling is shown in Fig. 25. We can see that both the LU factorization and the solving step are behaving far from ideal. These steps can take a significant portion of the execution time if we consider Fig. 13. If only the matrix construction part is utilizing the accelerator cards, then the overall speed-up is not expected to be more than 3–30% (depending on the physical model). Therefore, improvements of the PaStiX library are necessary, before the JOREK code can be efficiently used on the MIC hardware.

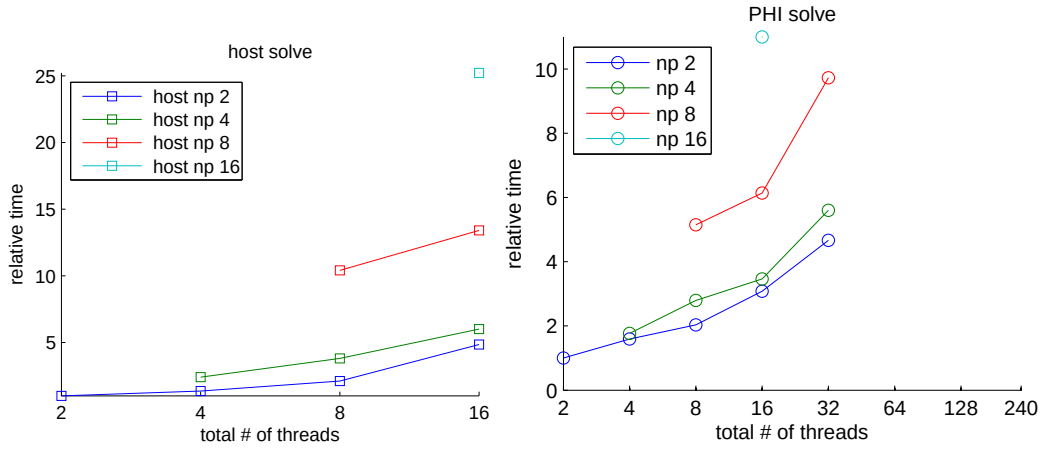


Fig. 25 Weak scaling of the iterative solver

3.8. *Conclusions*

The JOREK code was ported to the Intel Xeon Phi accelerator in native mode. Compiling the code on the new architecture was relatively simple, but the code is slower on the Xeon Phi. The performance was compared with simulations on the host. In its present form, the code runs around seven times slower in native mode on the Xeon Phi, than on the host. To understand the reason for this, the scaling of the three main stages of the simulation has been studied. These are the construction of the finite element matrix, its LU factorization, and the solving of a linear problem with an iterative method.

The matrix construction scales reasonably well on the Xeon Phi as we increase the number of threads, but the execution time is slow because of the lack of vectorization. With a separate example it has been shown, that the matrix construction can be vectorized. The vectorization improves the performance both on the host and on the accelerator card, and the vectorized code becomes faster on the Xeon Phi than on the host. Theoretically this could lead to a factor of three speed-up during the matrix construction step if we use two Xeon Phi cards.

For the LU factorization and the iterative solver, the PaStiX library is used in JOREK. The version of the library that was used in this work is not optimized for the MIC architecture, and therefore, it is slow on the Xeon Phi, not only for the JOREK matrix, but also for a simpler test example. Because of this, currently the JOREK code is not able to gain much using the Xeon Phi accelerator, the overall speed-up without an optimized linear algebra library would be between 3–30%. But the PaStiX library is also evolving, and if a new PaStiX version would use both the host and the Xeon Phi effectively, then the JOREK code could also benefit significantly from the accelerator cards.

4. Final Report on HLST project BLIGHTHO

4.1. *Introduction*

The BLIGHTHO project is explicitly providing support for the European scientists who use the Helios machine. At the beginning, the project went into operation by supporting selected European projects during the lighthouse phase. Starting in April 2012, after the Helios machine went into production phase, the support activities have been enlarged to all approved European projects.

The BLIGHTHO project gives support on different levels. HLST has access via the trouble ticket system of CSC to most of the tickets submitted by the European users. This gives the flexibility to pick up special concerns of users whenever necessary. In addition, the BLIGHTHO project investigates topics which are of general interest such as checking and improving the documentation provided by CSC. Especially this year we helped CSC to rebuild the whole documentation concerning threads and task pinning.

The main contribution addressed in this report concerns the extensive evaluation of the MPI libraries available on the Helios machine. Based on our experience with the HPC-FF machine we assessed that both the Bull and Intel MPI library require a large amount of time when initializing a so-called ALL_TO_ALL operation. In collaboration with the CSC support team we have strongly contributed to reduce the initialization time. Some alternative implementations have been thoroughly tested on several supercomputers and the outcome of this study was presented at an international conference on parallel computing.

4.2. *MPI libraries evaluation status*

Since Helios entered in operation, we performed three different tests on the available MPI libraries, namely: a barrier, a gather and a distributed matrix transpose. The first two tests represent basic MPI calls whereas the third one is more complex and stresses extensively the network of the whole system.

	MPI_Barrier	MPI_Gather	Transpose
Bullxmpi	64k	64k	64k
Intelmpi	32k	16k	8k
MVAPICH2	32k	32k	32k

Table 3 MPI libraries evaluation status

Table 3 shows the number of cores on which each test has been successful for each MPI library. One can see that Bullxmpi is able to run all the cases up to the full system, whereas Intelmpi encounters problems on 16k or more cores. These problems are under investigation by the CSC support. Even if the MVAPICH2 MPI library is not officially supported on Helios, it is still interesting to observe that it can scale up to half of the system for all three tests. The issues encountered on 64k cores are currently under investigation by the CSC support.

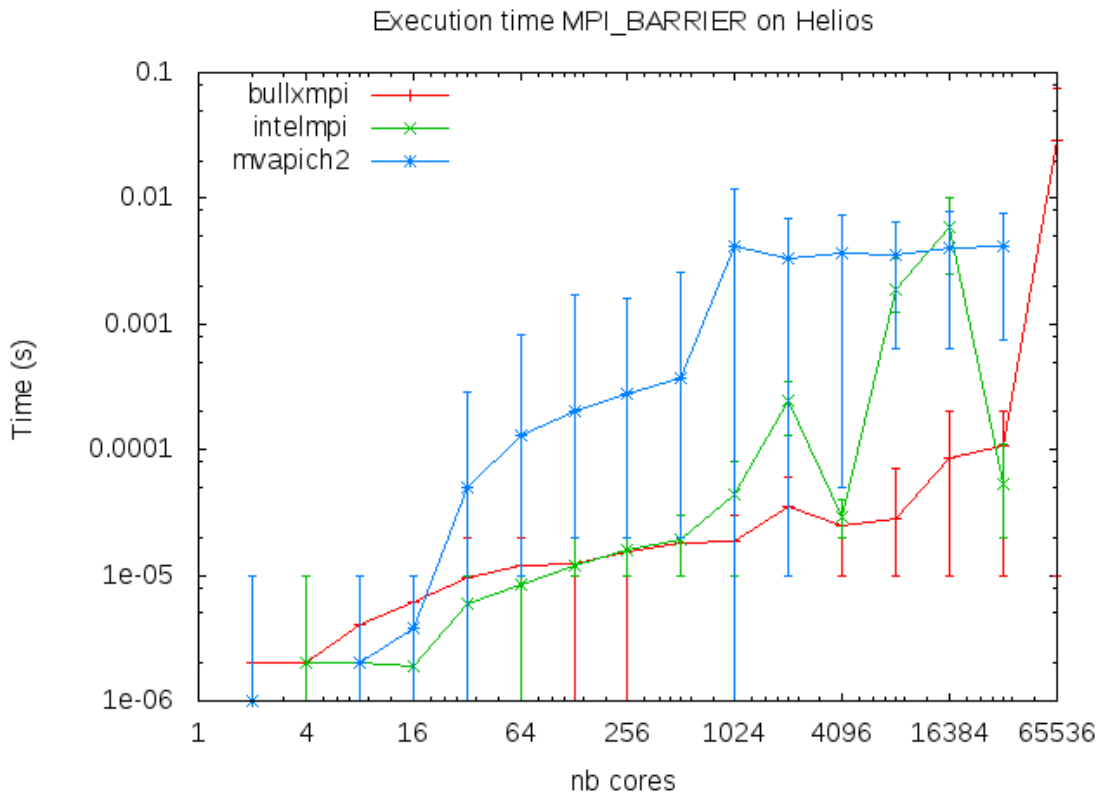


Fig. 26 Execution on Helios for MPI_BARRIER for the three available MPI libraries

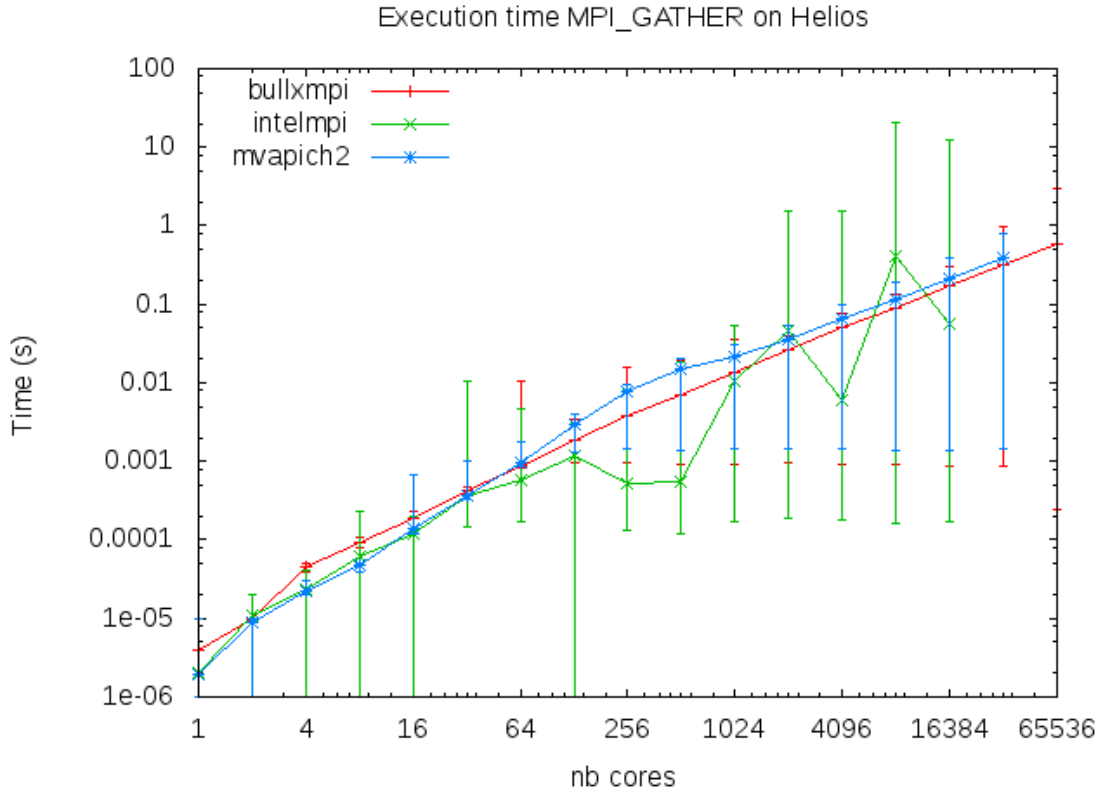


Fig. 27 Execution on Helios for MPI_GATHER for the three available MPI libraries

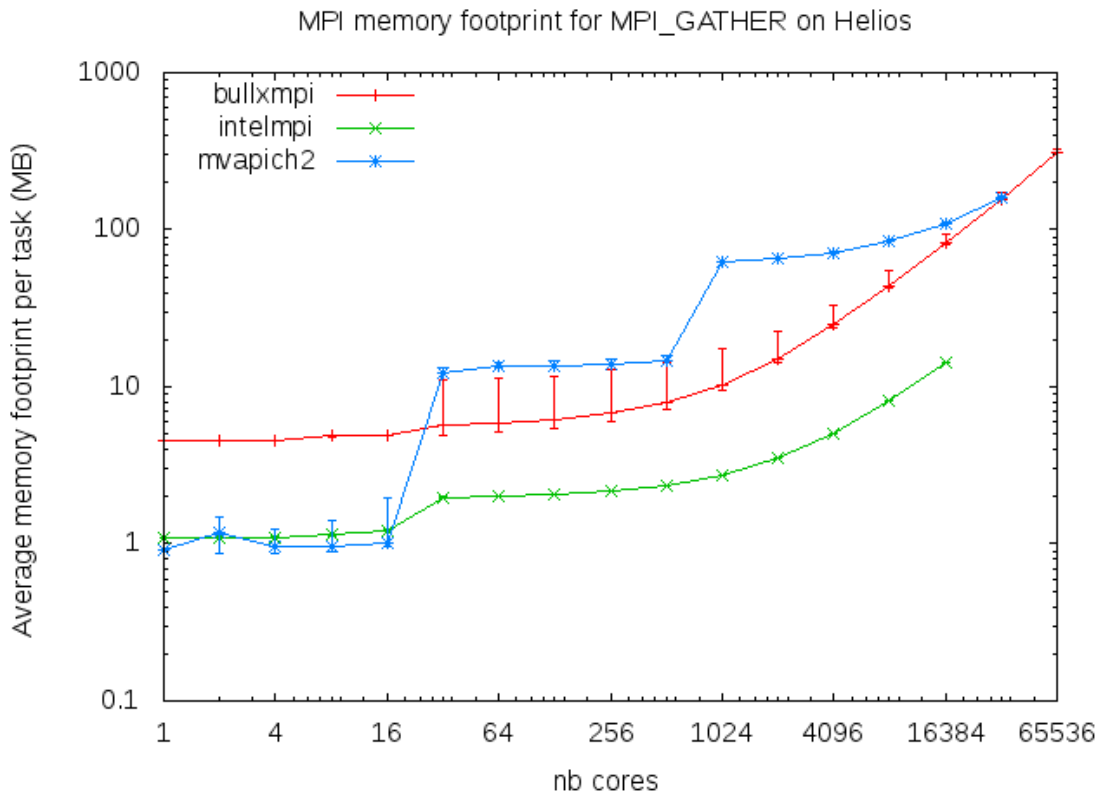


Fig. 28 MPI memory footprint on Helios for MPI_GATHER for the three available MPI libraries

For all three MPI libraries, we show respectively in Fig. 26 and Fig. 27 the execution time of the barrier and the gather and in Fig. 28 the memory consumption of the gather. The barrier is slower for MVAPICH2 than for the two others MPI libraries but all three libraries behave similarly for the gather operation. Quantitative details on the transpose test case can be found in section 4.4. We will see that MVAPICH2 behaves much better than Bullxmpi when it comes to the time needed for the setup of the appropriate environment to perform an all-to-all communication pattern.

4.3. *MPI non-blocking call evaluation*

We tested the MPI non blocking calls for the different MPI libraries available on Helios in order to check how efficiently they perform in terms of overlapping communication with computation. In practice, one has to initiate first the communication by calling the non blocking calls, then start the computation and finally, wait for the communication to end by calling the *MPI_Wait* procedure. If the overlap is efficient, the time spent waiting should correspond to the difference between the communication time and the computation time or should be zero if the computation time is greater than the communication time. We measured these timings by running two tasks on two different nodes exchanging a 1 GB message in two different contexts: with and without performing computations between the non blocking MPI call and the *MPI_Wait*.

Timings (s)	MPI call	Comp.	Wait	Total
Helios	10^{-5}	0	0.332	0.332
Hydra	10^{-5}	0	0.258	0.258

Table 4 MPI non blocking call evaluation without computation

Timings (s)	MPI call	Comp.	Wait	Total
Helios	10^{-5}	0.2	0.378	0.578
Hydra	10^{-5}	0.2	0.059	0.259

Table 5 MPI non blocking call evaluation with 0.2 s computations

Table 4 and Table 5 show the results obtained on both the Bull system Helios at CSC and the IBM iDataplex system called Hydra at RZG. One can notice that the time needed to perform the calls to MPI_Isend and MPI_Irecv is neglectable in all cases, i.e. these MPI calls are indeed non blocking calls. In Table 4, when no computations are performed, one can see the communication time that would appear in a blocking communication. Hydra is a bit faster as a result of a more recent Infiniband FDR interconnect, whereas the Helios interconnect is based on QDR.

In Table 5, we can clearly see that Hydra behaves as expected, i.e. the message is actually sent asynchronously and is overlapped by computation, as the wait time is the difference between the communication (as measured in Table 4) and the computation time. On Helios, the communication is simply postponed to the wait call. This suboptimal behavior has been observed for all three available MPI libraries Bullxmpi, Intelmpi and MVAPICH2 and has been reported to the CSC support.

4.4. *MPI library initialization time*

4.4.1. **Running the initial transpose**

On Helios, in collaboration with BULL and the CSC support, we managed to run the distributed matrix transposition application on the full system on 64K cores using the Bull MPI library. In order to reduce the initialization time, we had to launch our application with the so-called *srun* command instead of the *mpirun* one. The problem of excessive initialization times has been drastically reduced thereafter. So now the initialization time for a 64k run is approximately 1.25 hours and thus, sufficiently small to allow us to run our benchmark case within the wall clock time of the 24 hours queue which was not the case before. Additionally, the Bull MPI library was trying to allocate more memory than being available on a single node. By setting a low memory configuration for the Bull MPI library, we finally managed to run our benchmark case on the full system.

In addition, we ran our benchmark case on other computer architectures in order to compare the initialization and execution times and memory consumption with the Helios results. Initialization time (Fig. 29), MPI memory footprint (Fig. 30) and execution time (Fig. 31) are shown for the following computer systems:

- Cray XE6 (90 112 cores) at Edinburgh Parallel Computing Centre (EPCC) in UK
- IBM BlueGene/Q (458,752 cores) at Jülich Supercomputing Centre (JSC) in Germany
- IBM System x iDataPlex (147,456 cores) at Leibniz-Rechenzentrum (LRZ) in Germany
- Bull B510 blade system (70,560 cores) from the Computational Simulation Centre of the International Fusion Energy Research Centre (IFERC-CSC) in Japan

The experimental setup consists of a test case performing ten 2D matrix transposes successively using the XOR method with one MPI task per core. These kind of runs with one task per core will be called “flat” hereafter. Weak scalings going up to 64k cores are performed on four architectures but CPU time constraints imply that each run is performed only once. Nevertheless, every transpose is measured separately on each task, and the average, maximum and minimum are considered. As inter-tasks connections are actually established during the first MPI communication, the initialization time t_{init} is defined as follows:

$$t_{init} = t_{MPI\_Init} + t_1 - < t_5 - t_{10} >,$$

where t_{MPI_Init} is the duration of the call to MPI_Init, t_1 is the execution time of the first transpose and $< t_5 - t_{10} >$ the average execution time of the last five ones.

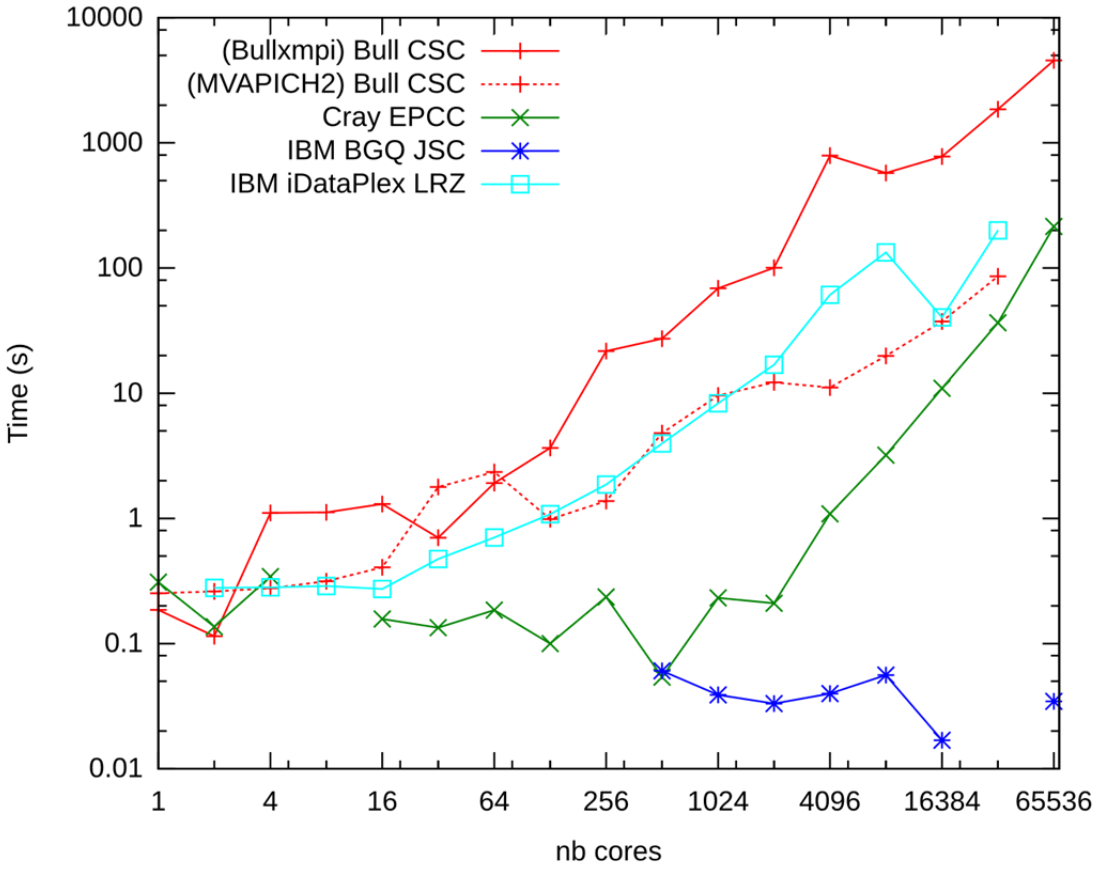


Fig. 29 MPI initialization time

Fig. 29 shows the time needed to setup the MPI environment for different architectures and for different MPI libraries. The results obtained on the IBM BlueGene/Q are really impressive: initialization time does not increase with the number of cores. At the other extreme, the worst behavior has been observed on the Bull system with Bullxmpi which still needs 1.25 hours to get initialized on 64K cores. However, on the same Bull system, the MVAPICH2 MPI library exhibits an initialization time one order of magnitude lower than Bullxmpi which is close to the one obtained on the IBM iDataplex system. This has been reported to the CSC and improvements of Bullxmpi on this topic are awaited in the next release which should be available early 2014. The initialization cost on the Cray system is negligible up to 2K cores and starts to rise up to 214 s on 64K cores. This seems to be reasonable as 64k cores already represent two thirds of the machine.

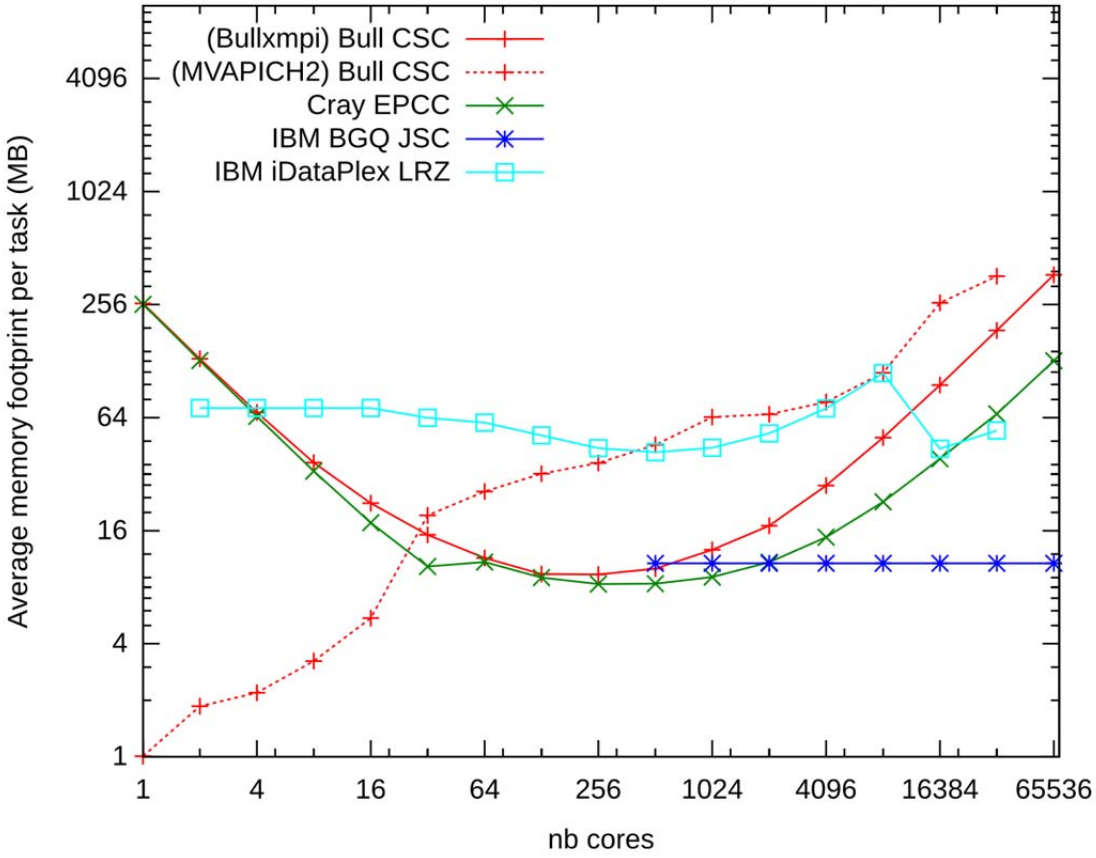


Fig. 30 Total memory footprint of the MPI library

Concerning the MPI memory footprint on Fig. 30, again, results on the BlueGene/Q are impressive. The footprint stays at 11 MB whatever the number of tasks used. The three other machines behave differently in the sense that the memory consumption increases with the number of cores. The MPI library allocates a reasonable amount of memory in any case but it is still within the resources of a node. However, for even larger test cases than 64k cores, a further increase in memory could become a problem.

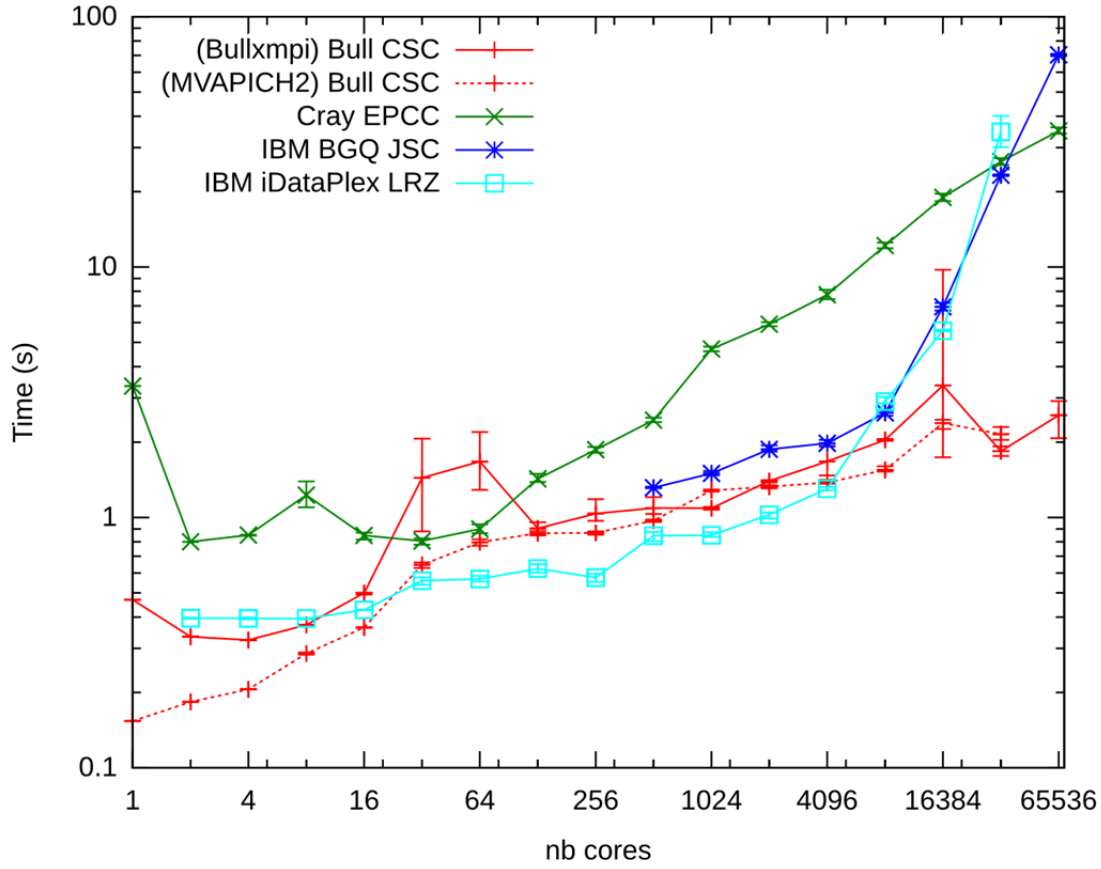


Fig. 31 Execution time of a single transpose with the XOR method

Fig. 31 shows the time needed to perform a single transpose. We can see that it gets larger for larger number of cores, just as expected. Bull and Cray systems show a similar smooth increase even if the slope is steeper for the Cray machine. However, the two IBM machines, the BlueGene/Q and the IBM iDataplex, show a large degradation in execution time for runs with 8k cores or more. The Bull system shows the best performance on 64K cores with around 2 s execution time to perform the transposition of the distributed matrix. The BlueGene/Q shows a similar smooth trend up to 8K cores but then degrades down to 70 s on 64K cores. We will see in the next section that the results are much better on IBM machines with alternative implementations of the transpose algorithm.

4.4.2. **Alternative transpose implementations**

As already explained in the 2012 annual report, we tested different alternative implementations in order to reduce the MPI initialization time and memory consumption. These alternatives consist of reducing the number of communicating MPI tasks during the communication operation by using different techniques. In all the cases, a single task per node is involved in the global all-to-all communication pattern:

• MPI+OpenMP, so-called “hybrid’’ model: The idea is to allocate one task per computing node and to use threads at the computing level on each core. The price to pay here is to implement the OpenMP pragmas and the thread synchronization to avoid race conditions.
• Pure MPI with “two communication levels’’: One MPI task per core is allocated but thanks to MPI communicators, a single task per node is responsible for the global communication. The price to pay here is to scatter and gather data from and to the communicating task within each node. An additional memory buffer is also required on the communicating task.

• MPI + “shared memory segment’’: As with the “two communication levels’’ solution, one MPI task per core is allocated. But instead of working on their own private memory chunk, all the tasks within a node work on a “shared memory segment”. The price to pay is the allocation of the “shared memory segment” and the introduction of explicit synchronizations among the tasks belonging to a same node in order to avoid race conditions.

All these alternative implementations perform pretty well on the four supercomputers used for this study. Initialization time is reduced by an order of magnitude for Cray and IBM iDataPlex, even more for the Bull CSC system. The MPI memory footprint is also reduced by an order of magnitude on all systems when the “hybrid” or the “shared memory segment”' implementations are used. However, the “two communication levels” implementation exhibits similar MPI memory consumption or even a larger one than the “flat” implementation due to the required additional memory buffer. From the execution time perspective, the situation is different on each system. Results for the fastest methods for each system are shown on Fig. 32. For Bull CSC, the fastest method remains the “flat” implementation. The others do not perform badly but they are simply slower by 1 s approximately, which nevertheless, represents a degradation of 50%. For Cray, the picture is roughly the same except that the performance drops for large numbers of cores even for the alternative implementations. This situation was not expected and is still under study at EPCC. The actual best performance on the Cray system was obtained by using the MPI_Alltoall routine instead of the XOR method. For the IBM iDataPlex, all three alternatives present very similar performance and do not degrade as much as the “flat” run on four islands. Finally, the IBM BGQ shows very different behavior for all these alternatives, the best one being by far the “two communications levels”.

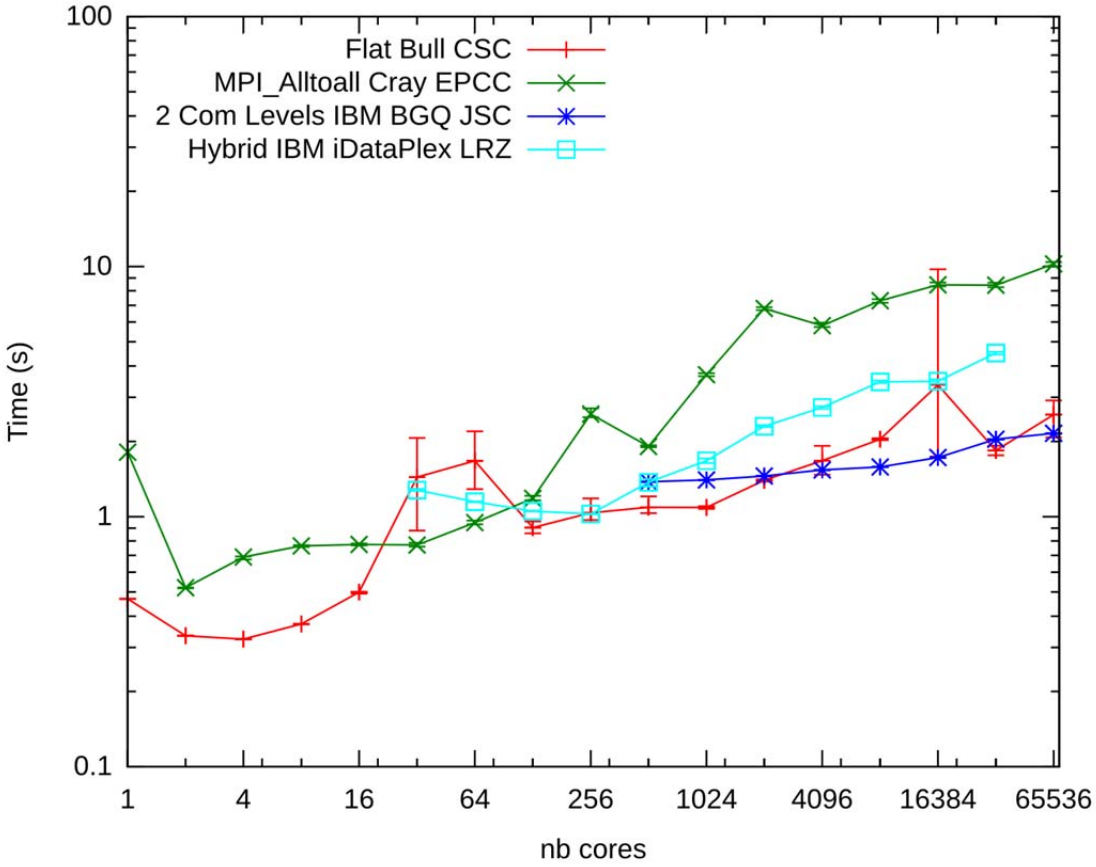


Fig. 32 Best single transpose execution time among all implementations obtained for the same weak scaling

On the way to exascale, the foreseen programming model is still an open question. Our study shows that up to current petaflop systems, pure MPI implementations are still relevant for such a distributed matrix transpose problem, which is a standard problem in scientific applications invoking multidimensional Fourier transforms. The question is now how far will such a pure MPI approach hold and whether or not these alternative implementations will be mandatory on next generation machines. For now, the only conclusion we can formulate is that they perform already pretty well on current petaflop systems even if they behave quite differently on different architectures.

From a software engineering point of view, the “two communication levels” alternative is the one that requires the least modifications in the code, but its memory overhead may be not affordable for some applications. The “shared memory segment” alternative is not the recommended way to follow for two reasons. First, it requires the largest changes in the code. Second, as the allocation of such a “shared memory segment” is system dependent, this reduces the portability of the code as well. The latter argument might disappear in the future as MPI 3 standard supports the usage of “shared memory segments”. Finally, the “hybrid” approach requires skills in OpenMP, but the impact on the source code is not extended as with the shared memory segment. However, this alternative never showed the best performance on any architecture.

4.5. *Task/thread pinning*

Since the beginning of years 2000, modern computer architectures have the trend to integrate more and more computing cores per node. As a side effect, memory access becomes less and less uniform. The name for such computer memory architecture is “cache coherent Non-Uniform Memory Access” (ccNUMA). This means that the available memory throughput to access a specific memory location may be different from one core to another within a single node. This is the case on Helios nodes as depicted in Fig. 33. Cores 0–7 have a better access to memory bank 0 than the cores 8–15 as data have to cross the QPI interconnect. This has consequences at the application level. Indeed, with uniform memory accesses, MPI processes or underlying threads could run with the same performance anywhere within the node. They all would have the same kind of access to the shared memory. In detail the situation is even more complicated as the different cores within one processor have private L1 and L2 caches but share the L3 one. Hence, one has to care about where processes and threads actually run within the node by pinning them explicitly on the different cores.

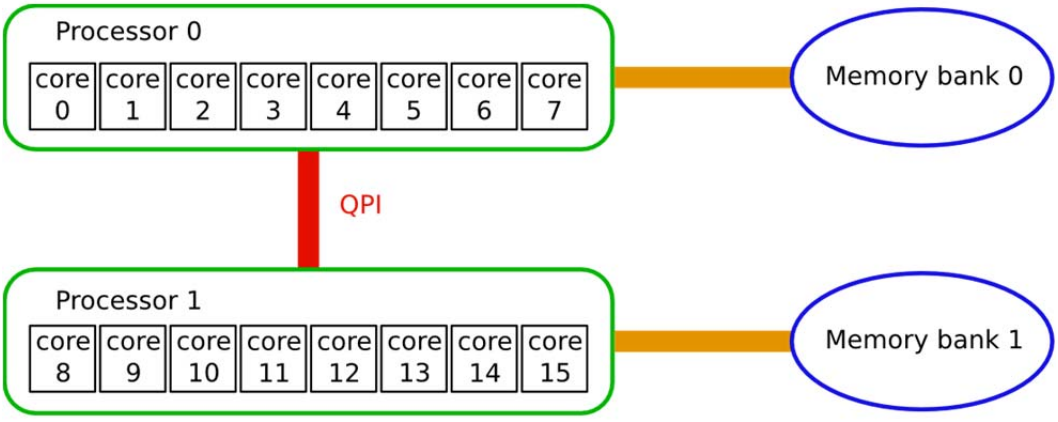


Fig. 33 Sketch of the Intel Sandy Bridge architecture

4.5.1. **How threads are actually pinned**

Whatever the method used, in the end the actual pinning of an MPI process or a thread is performed by a system kernel call. So, from the user perspective, the lowest level of implementation is to include these kernel calls directly into the application. However, such a procedure is quite technical and several other tools exist which perform the pinning task in a more convenient way. These tools operate at two different levels.

The first one is the MPI task pinning which is supported by several tools:

- MPI launcher: it is the application which actually creates the MPI tasks at initialization time:
  - *mpirun/mpiexec*: each MPI implementation provides such a launcher and each has different options to pin MPI processes
  - *srun*: SLURM scheduling system available on Helios is able to create itself the MPI tasks whatever the MPI implementation used afterwards
- *taskset*: application which launches the MPI launcher
- *numactl*: same as *taskset* but with additional options for memory pages allocation on different NUMA domains

The second level is the thread level. It is not possible to pin threads with the previously listed tools because at the time they operate, the threads do not exist yet. Accordingly, threads can be pinned only after their creation:

- *hwloc/likwidpin*: tool packages that wrap a whole MPI implementation and so provide additional pinning options for *mpirun*. It is difficult to set the proper environment for these tools. The technology seems to be not mature yet.
- OpenMP: if the threads are created by OpenMP, there are two ways to specify how threads should be pinned:
  - By setting the environment variable OMP_PROC_BIND=TRUE, the user tells OpenMP to pin threads in the simplest way. For example, for a MPI process scheduled to run on four cores and which in addition create four threads, OpenMP will pin the first thread on the first core, the second on the second … and so on.
  - KMP_AFFINITY and GOMP_CPU_AFFINITY environment variables can be used to describe more complex pinning patterns. These kinds of options are specific to each OpenMP implementation.
- POSIX: if the threads are created with the POSIX interface, the only way to specify how thread should be pinned is to use direct calls to the system kernel.

In the case of a pure MPI application (which represents still the majority of applications running on Helios), all these options are not necessary. By default all MPI processes are pinned on a specific core and are distributed alternatingly among the two processors of a single node. So the default behavior is the most efficient in most cases. But for general hybrid applications, the default pinning behavior of the system is not optimal. Hence, it is up to the user to take advantage of the above mentioned options for pinning threads of his application. The point of this investigation is to identify the simplest and most general strategy. Tests have to be performed to finally document the according procedures for the user.

4.5.2. **The proposed pinning strategy**

The initial Helios documentation was minimal on this topic and suggested the users to use *mpirun* launchers in combination with *taskset* or *numactl*. After some tests, it turned out that this was not the most convenient way and that hybrid applications were not handled properly. Currently, the advised strategy consists of using *srun* as MPI launcher in the first place for mainly four reasons. First, the same pinning options remain whatever Bullx or Intel MPI implementation is used to run the application. Second, with *srun* all the pinning options can be given simply inside the job script instead of having an additional intermediate script, as is the case when

numactl or *taskset* are used. Third, we noticed that *srun* enables to reduce drastically the cost of the MPI initialization with large number of tasks. In this case, each MPI process is given a set of cores to run on. If none of these sets overlap, the simplest thread pinning strategy offered by OpenMP is enough to get a properly pinned application. The fourth and final reason is that we can avoid the usage of variables dedicated to some OpenMP implementations.

The remaining difficulty at the user level is to define the non overlapping sets of cores. Being trivial when the application requires only one MPI process per node or per processor, the situation becomes more complicated if many MPI processes are required. The main difficulty is then to ensure that the pinning options are correct because no error/warning messages are displayed in the case of a disordered pinning. In such case, several threads might simply run on the same core, leaving other cores idling. Therefore, a performance drop is a possible sign for such bad pinning but a real diagnostic of the issue can only be performed by getting information from the system kernel. Such a diagnostic application can be downloaded from the Helios documentation pages and can be used in place of the target application to test the pinning options.

Following our findings CSC has revised the section 5.3 of the documentation dedicated to CPU binding.

4.5.3. **Pinning's impact on performance**

Process and thread pinning can have a large impact on the available main memory bandwidth. The theoretical maximum bandwidth is the product of the number of memory channels, the number of bytes per channel, the number of transfers per clock tick and the bus frequency (GHz):
Theoretical bandwidth for Intel Sandy Bridge = 8 x 8 x 2 x 1.6 = 204.8 GB/s

This value is never reached in practice for various reasons. However, it is possible to measure the sustained main memory bandwidth instead. The stream benchmark[1] has become a standard for measuring this quantity. Fig. 34 shows the results obtained when the benchmark is run on a single processor on Helios. One can see that the thread pinning has no influence on the result. This means that within a processor, main memory access is uniform. In principle, if threads move from one core to another within the same processor, this may invalidate some cache lines and may cause performance loss in some cases. However, this could not be confirmed with this benchmark.

Fig. 35 shows the same benchmark, but now on both processors of the node. One can clearly see the influence of the pinning process by looking at the error bars which show the minimal and maximal bandwidth achieved for different number of threads. In Fig. 34 these two values typically coincide but not in Fig. 35 when pinning is switched off (red curve). Here the minimal bandwidth can be up to 40% smaller than the maximal value reached for the same number of threads in the case with pinning (green curve). Thus, there can be a significant degradation of the bandwidth when threads are moved from one processor to the other by the system scheduler. In this case, data have to cross the QPI interconnect each time they are accessed in order to bring them on the processor where the thread is currently running. Even if this mechanism is transparent to the user, it can have a large impact on performance, as we can see.

The current trend which consists of an expressed growth of computing cores per node will likely increase further the non-uniformity of memory accesses. So process

[1] http://www.cs.virginia.edu/stream/

and thread pinning will become more and more mandatory to exploit efficiently the available resources on next generation computers.

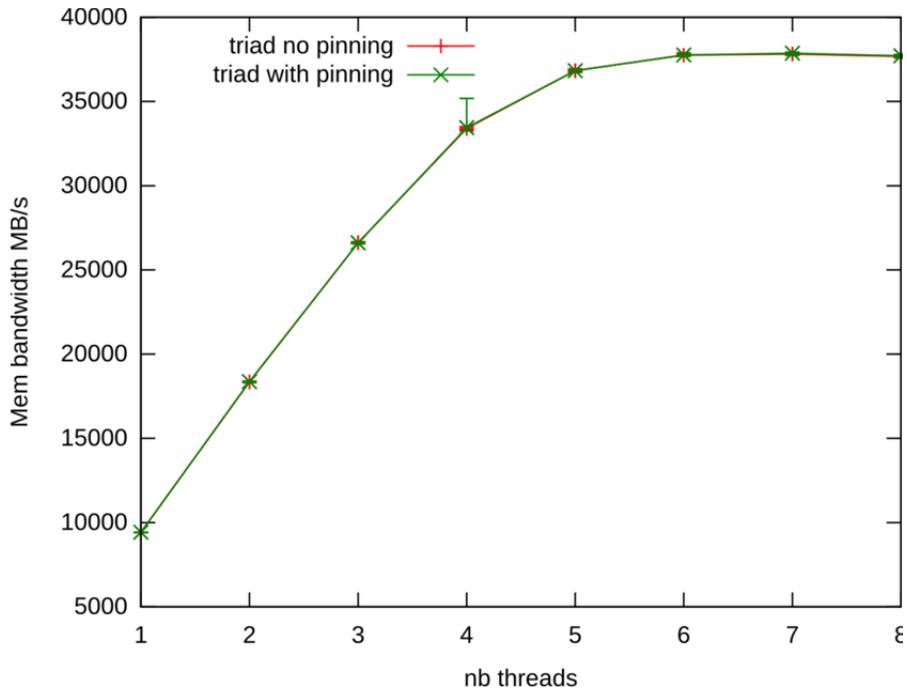


Fig. 34 Triad result of the stream benchmark on a single processor.

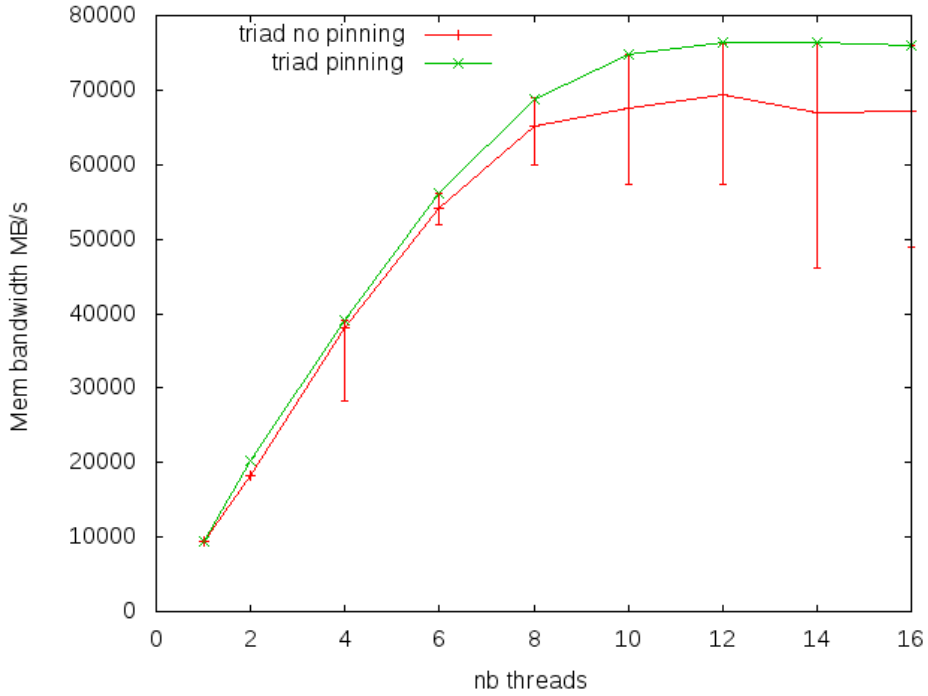


Fig. 35 Triad result of the stream benchmark on the whole node.

4.5.4. **Non temporal stores or streaming stores**

The concept of non “temporal stores” (Cray terminology) or “streaming stores” (Intel terminology) is about which machine language commands the compiler selects to write data into memory. The purpose of steaming stores is to reduce the data traffic on the main memory bus and thus, to improve performance of memory bound computing kernels. To illustrate this concept, let us consider the computing kernel of the simple triad example of the “stream benchmark”:

```
For i=1, N
  A(i) = B(i) + scalar*C(i)
```

In order to execute a single iteration of this loop, two elements have to be moved from memory to registers (*B(i)* and *C(i)*) and one element from register back to memory (*A(i)*). So theoretically, three elements per loop iteration have to travel via the main memory bus the whole cache hierarchy L1, L2 and L3. However, an element present in a register cannot be written directly to memory, it has to be written in L1 cache first. And it has to be written into a cache line that is associated with the memory address of *A(i)*. Such a cache line cannot be “allocated” directly in L1 as cache lines are only transferred from one cache hierarchy to another. So, in order to write back *A(i)*, the associated cache line has to be present in L1. If it does not, it has to be moved via L2 and L3 from main memory. As a consequence, writing back *A(i)* into memory costs the transfer of two elements: one to bring *A(i)* from memory into the cache even if the value of *A(i)* is not used and one to write the value back into memory. This procedure is called “write/allocate” and is the procedure the compilers select by default in binaries at compile time. “Streaming stores” on modern architectures avoid this “cache allocation”. When this feature is switched on at compile time, *A(i)* is written in a specific L1 cache line which does not require to bring the data from main memory. So in this case, writing back *A(i)* into memory costs the transfer of a single element as theory predicts. But this comes with a little overhead illustrated in Fig. 36.

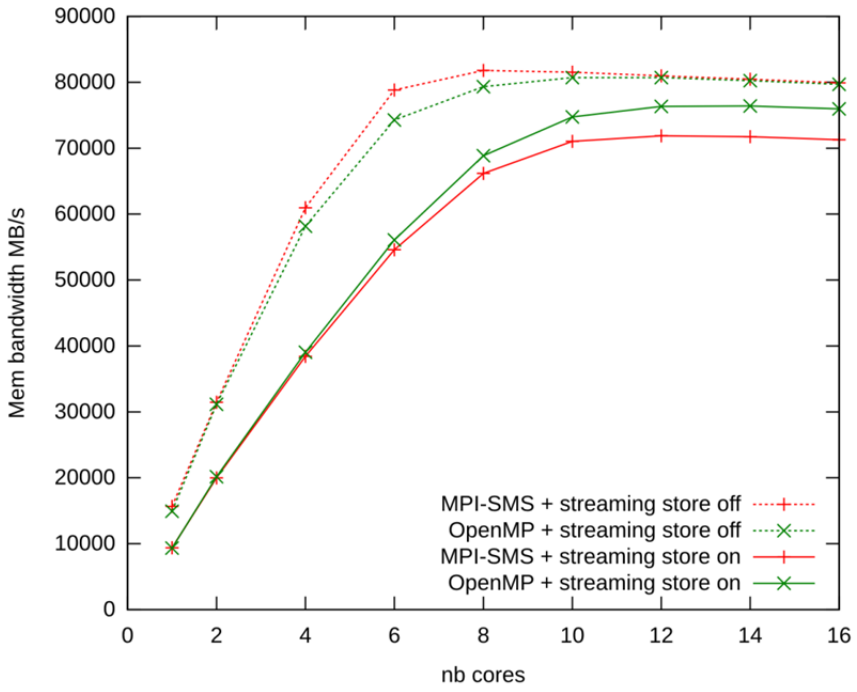


Fig. 36 Triad result of the stream benchmark on the whole node with pinning for four different configurations.

Let us first consider the original “stream benchmark” implementation, i.e. the results for an OpenMP implementation.

Table 6 shows the results for the triad of the stream benchmark on 16 threads with pinning for a vector length of $N = 10^7$. As such a computing kernel is clearly memory bound, the usage of “streaming stores” improves the performance, thus, it reduces the execution time. As with “steaming stores” three elements per iteration are travelling over the memory bus instead of four, one could expect a reduction of 25% resulting in an execution time of 0.003015 s. But the measured execution time is slightly above this value by 5%, meaning that the usage of “streaming stores” comes with a small overhead. A reason might be that “streaming stores” are not able to saturate completely the memory bus, achieving only 95% of the maximum sustained value.

Triad OpenMP	Time (s)	Size (MB)	Bandwidth (MB/s)
Streaming store on	0.003168	3xNx8=240	75757
Streaming store off	0.004020	4xNx8=320	79601

Table 6 Triad result of the stream benchmark on 16 threads with pinning for $N = 10^7$ and for “steaming stores” switched on or off.

In addition we have evaluated a modified version of the stream benchmark which makes use of “shared memory segments” (SMS). Instead of having one process that spawns several threads, the application is made of several processes that share an SMS. As one can see in Fig. 36, using threads or processes sharing an SMS does not lead to the same maximum memory bandwidth. When “streaming stores” are switched off, processes sharing an SMS can reach a higher memory bandwidth for small numbers of cores, i.e. when the bus is not yet saturated. At saturation level, both methods reach the maximum value of 80 GB/s bandwidth. However, when “streaming stores” are switched on, the saturation level is different for the two methods: processes sharing an SMS reach 71 GB/s compared to 76 GB/s for threads. The reasons for this discrepancy seem to be deeply buried in specific interactions between the operating system and the hardware. They are not completely understood yet.

The activation or deactivation of “streaming stores” has to be done at compile time. For the Intel compiler, there are two ways of doing this. First, the compile option *-opt-streaming-stores always|never|auto* applies respectively always, never and when appropriate “streaming stores”. The default is *auto* and this results usually in a deactivation of “streaming stores”. The second way is to add a pragma statement before the loop that should make use of “streaming stores”: *#pragma vector nontemporal* in C language and *!DEC$ VECTOR NONTEMPORAL* in Fortran.

In principle we recommend the usage of “streaming stores” in applications. In the context of memory bound computing kernels, depending on the number of arrays accessed, this feature can improve performance by up to 30%. The coding effort to enable “streaming stores” is low as a single pragma line is already sufficient.

5. **Final report on the KSOL2D-2 project**

5.1. ***Introduction***

The project aims at improving the Poisson solver of the BIT2 code, which 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. The new BIT2 code incorporates a number of techniques, which were originally developed for its predecessor, the BIT1 code, such as, optimized memory management, fast and highly scalable nonlinear collision operators. It is mandatory that the Poisson solver used in BIT2 scales to very high core numbers in order to maintain the good scaling property of the whole code. So the work plan is to further improve the scaling of the Poisson solver in 2D.

In the former project KinSOL2D, we developed a discretization of the Poisson problem and implemented a solver which is based on the multigrid method. The development of such a solver was a challenging task, especially to reach a good scaling property (~100 cores) for the complex SOL geometry (a hollow rectangular with $>10^7$ cells). Although the outcome was an efficient all-purpose Poisson solver, the project coordinator requested further speed-up. Namely, the minimum time required for the solution in a realistic geometry should be less than 0.1 s to afford simulations consisting of at least 10^7 time steps.

As further improvement we pursue the following two approaches. One is based on the multigrid method with gathering of the data at a certain coarser level. This approach was used successfully within the MGTRI project for a structured triangulation for a hexagonal domain. The other approach is to start from the "physics-based" domain decomposition developed in BIT1. The main idea is to decompose the domain into subdomains, so that the potential field in each subdomain can be calculated separately using homogeneous boundary conditions and to couple (and update) each subdomain boundary via physical boundary conditions. The advantage of such an approach is that the calculation of the field (with homogeneous boundary conditions) in each subdomain can be performed nearly independently. The update due to boundary conditions requires only the exchange of small messages between the processors. As a result, the scalability of the solver should increase significantly.

Among the many domain decomposition methods, the dual-primal finite element tearing and interconnection (FETI-DP) and balanced domain decomposition with constraints (BDDC) methods are successful two-level non-overlapping domain decomposition methods for 2D and 3D problems. In the MGTRI project, we tested these two methods for a structured triangulation for a hexagonal domain. Even if these two methods turn out to be slower than the multigrid method for the SOL geometry, such an approach might still give valuable experience.

5.2. ***Model problem***

We consider the second order elliptic partial differential equation with the coefficient $\varepsilon(x,y)$ being defined on a rectangular domain with an internal conducting structure as shown in Fig. 37. As boundary conditions, we have to consider a Dirichlet zero boundary condition for the outer wall, $\Phi_W(x,y,t) = 0$, a Neumann zero boundary condition for the inner empty surface, $E_n(x,y,t) = 0$, and a Dirichlet boundary condition for the internal conductor, $\Phi_c(x,y,t) = \Phi_c(t)$.

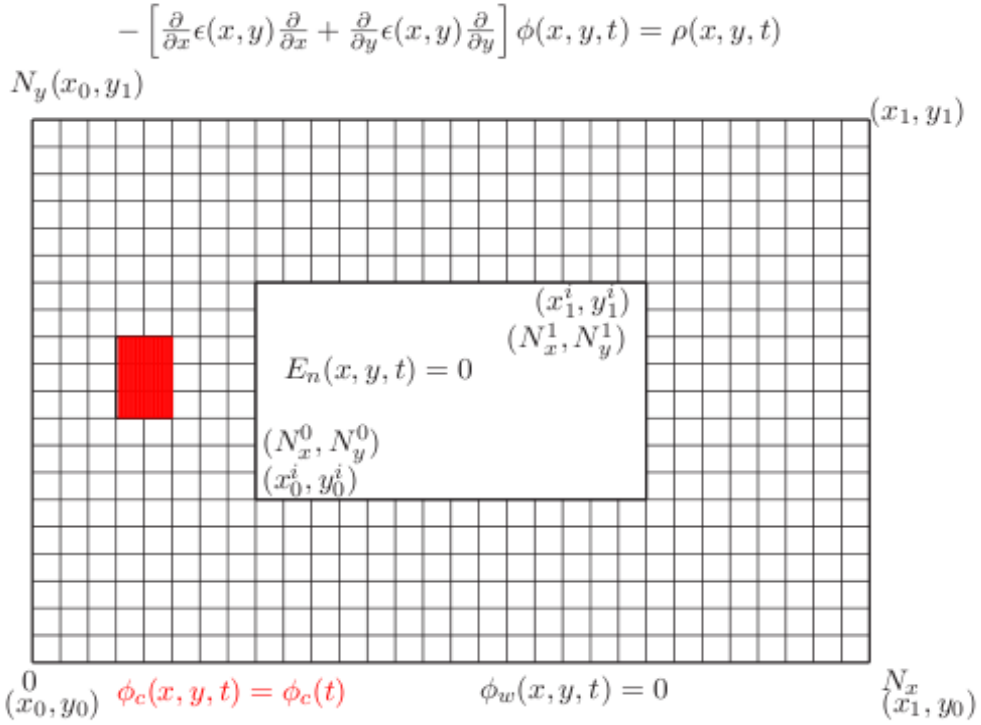


Fig. 37 Rectangular domain with internal conducting structure (■)

As discretization method we start with the finite difference method (FDM). Its stencil has at most five nonzero elements and is intuitively understandable. We use a structured mesh with *ε(x,y)* being defined on each cell boundary. The coefficient *ε(x,y)* is time dependent and will change for each time step of the simulation. For the SOL geometry, the generated matrix is a sparse nonsymmetric matrix due to the inner Neumann boundary condition. In contrast the stencil for the finite element method (FEM) results in at most nine nonzero elements. Thus, it is more complex than the FDM stencil and the according matrix-vector multiplication which is the basic operation of the iterative solver becomes more costly. However, the matrix of the Poisson problem is symmetric which reduces the number of iterations compared to a nonsymmetric matrix problem. Therefore, a performance comparison between the FDM and FEM is worth to be investigated. In addition, the FETI-DP method is mathematically well analized for the FEM which is not the case for the FDM. Hence, the FEFI-DT method is founded on more solid theoretical grounds for the FEM than for the FDM.

5.3. *Multigrid method with reduced number of cores*

The multigrid method is a well-known, fast and efficient algorithm to solve many classes of problems. In general, the ratio of the communication costs to computation costs increases when the grid level is decreased, i.e., the communication costs are high on the coarser levels in comparison to the computation costs. Since the multigrid algorithm is applied on each level, the bottleneck of the parallel multigrid lies on the coarser levels, including the exact solver at the coarsest level.

The number of degrees of freedom (DoF) of the coarsest level problem scales with the number of cores (parallel coarsest level limitation). To improve the performance of the parallel multigrid method, we consider to reduce the number of executing cores to a single core (the simplest case) after gathering data from all cores on a certain level (gathering level) as shown in Fig. 38. After gathering the data on a single core further serial multigrid levels are possible. Thus, the algorithm avoids the parallel coarsest level limitation. The optimal data gathering level for large numbers of cores is the coarsest possible parallel multigrid level. It guarantees a minimum of data transfer in the gathering procedure. The numerical results of the multigrid algorithm

showed that it has a very good semi-weak scaling property up to 24,576 cores for the large problem sizes of the MGTRI project.

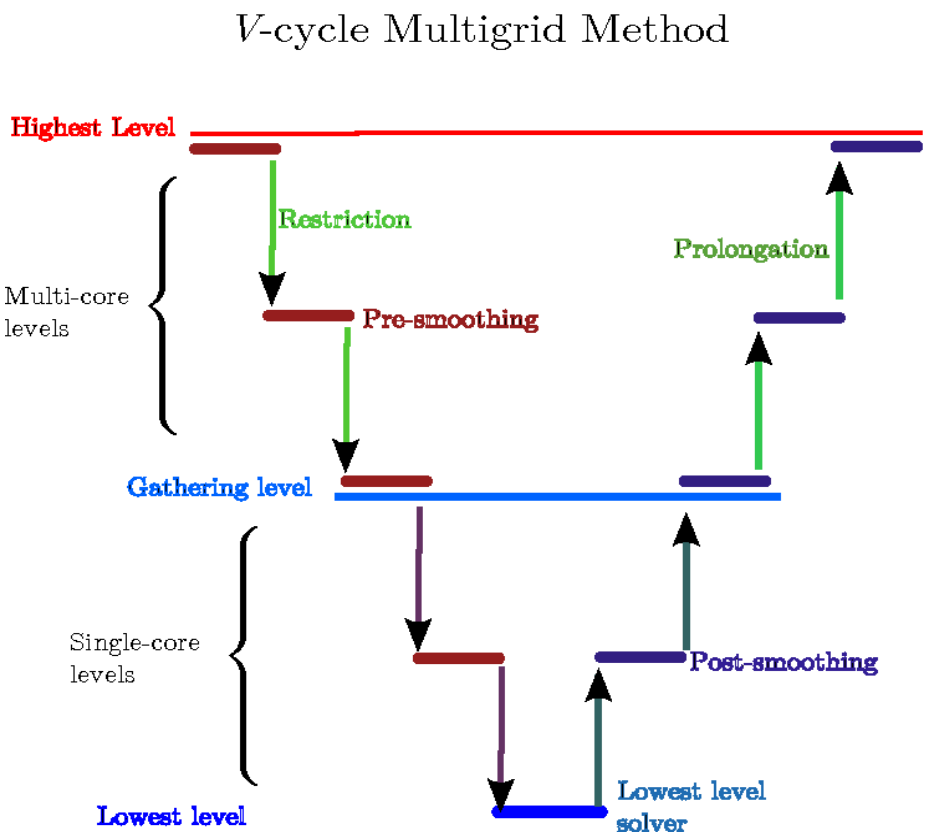


Fig. 38 Multigrid algorithm with single core levels

For the problem at hand the coarsest level has to resolve the inner structures unlike for the standard problems such as the one from the MGTRI project. This leaves the coarsest level with a couple of thousands DoF which for the multigrid method are more efficiently solved on the multiple core-level than on a single-core level. Unfortunately, this concept comes to its limit when the total number of cores is significantly large, as it is the case on the Helios machine with up to 64k cores. In such a case the DoF on the coarsest level is not sufficiently large to be distributed among all the cores. To overcome this obstacle, all data should be gathered at a certain level with an all-reduce operation for each core. As a first step, the data structure of BIT2 has to be adapted to make the gathering of the data feasible. This is in principle possible but nevertheless, the communication costs will cause an additional overhead. Instead, it seems to be more efficient to switch at the coarsest levels to a domain decomposition method which has the potential to achieve reasonable efficiency even with a small number of DoF per core.

5.4. *FETI-DP method*

We focus on the non-overlapping two-level domain decomposition method (DDM) because it is intrinsically parallel and therefore, achieves very good parallel efficiency. In addition, its required number of iterations does not depend on the number of sub-domains (cores). To define the non-overlapping DDM, we suppose that the domain is partitioned into *N* non-overlapping sub-domains as shown in Fig. 39.

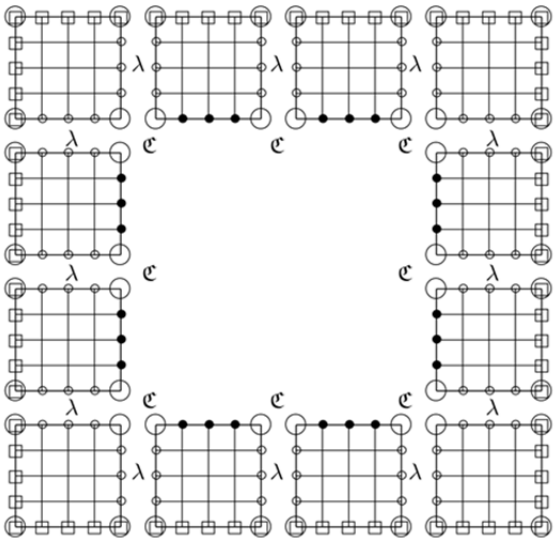


Fig. 39 Non-overlapping subdomains of the SOL domain

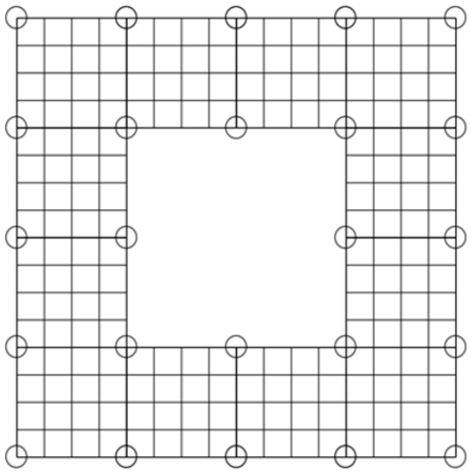


Fig. 40 Coarse level domains of the FETI-DP method for SOL domain

The non-overlapping DDM consists of two steps: one to solve the local problem on each subdomain, which do not overlap with any other subdomain and the other to solve a modified local problem with imposed continuity at the boundaries of the sumdomains. Depending on how the continuity is imposed on the boundaries for the non-overlapping DDM, we distinguish between the Dirichlet-Dirichlet and Neumann-Neumann DDM cases. In particular, the BDD method belongs to the Neumann-Neumann DDM and the FETI method is a Dirichlet-Dirichlet DDM. In particular, the FETI method uses Lagrange multipliers λ to impose the continuity on the inner boundary nodes. The local matrix of FETI which comes from the Neumann boundary problem is usually a singular matrix. To avoid this singularity, the FETI-DP method imposes continuity on the corner nodes which are part of more than one subdomain as shown in Fig. 39. This leads to a coarser level domain for only the corner nodes as shown in Fig. 40.

The whole procedure can be expressed in the following system of equations:

$$\begin{pmatrix} A_{\Lambda\Lambda} & A_{\Lambda\mathfrak{C}} & B^T \\ A_{\mathfrak{C}\Lambda} & A_{\mathfrak{C}\mathfrak{C}} & 0 \\ B & 0 & 0 \end{pmatrix} \begin{pmatrix} u_\Lambda \\ u_\mathfrak{C} \\ \lambda \end{pmatrix} = \begin{pmatrix} f_\Lambda \\ f_\mathfrak{C} \\ 0 \end{pmatrix}.$$

Applying one step of the block Gaussian elimination, we obtain the reduced system

$$\begin{pmatrix} S_{\mathfrak{C}\mathfrak{C}} & -A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}B^T \\ -BA_{\Lambda\Lambda}^{-1}A_{\Lambda\mathfrak{C}}^T & -BA_{\Lambda\Lambda}^{-1}B^T \end{pmatrix} \begin{pmatrix} u_{\mathfrak{C}} \\ \lambda \end{pmatrix} = \begin{pmatrix} f_{\mathfrak{C}} - A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}f_\Lambda \\ -BA_{\Lambda\Lambda}^{-1}f_\Lambda \end{pmatrix}$$

where

$$S_{\mathfrak{C}\mathfrak{C}} = A_{\mathfrak{C}\mathfrak{C}} - A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}A_{\Lambda\mathfrak{C}}.$$

By eliminating again, we obtain the reduced system for λ

$$F\lambda = d$$

where

$$F = BA_{\Lambda\Lambda}^{-1}B^T + BA_{\Lambda\Lambda}^{-1}A_{\Lambda\mathfrak{C}}S_{\mathfrak{C}\mathfrak{C}}^{-1}A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}B^T = B\tilde{A}^{-1}B^T$$
$$d = BA_{\Lambda\Lambda}^{-1}f_\Lambda - BA_{\Lambda\Lambda}^{-1}A_{\Lambda\mathfrak{C}}S_{\mathfrak{C}\mathfrak{C}}^{-1}(f_{\mathfrak{C}} - A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}f_\Lambda) = B\tilde{A}^{-1}\tilde{f}.$$

F has less numbers of DoF than the original problem because the number of DoF is the number of nodes on the inner-boundary. But, these matrices are denser than the original one and globally defined. From the construction of these matrices, we can perform the matrix-vector multiplication locally. So we can solve the system by using an iterative method such as PGMRES with a locally defined preconditioner.

We define a Dirichlet preconditioner for the system as

$$F_D^{-1} = \sum_{s=1}^{N} B \begin{pmatrix} 0 & 0 \\ 0 & S_{EE} \end{pmatrix} B^T$$

where

$$S_{EE} = K_{EE} - K_{EI}K_{II}^{-1}K_{IE}$$.

After solving the reduced system, we can get finally the solution by the following computations

$$u_{\mathfrak{C}} = S_{\mathfrak{C}\mathfrak{C}}^{-1}(f_{\mathfrak{C}} - A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}f_\Lambda + A_{\mathfrak{C}\Lambda}A_{\Lambda\Lambda}^{-1}B^T\lambda)$$
$$u_\Lambda = A_{\Lambda\Lambda}^{-1}(f_\Lambda - A_{\Lambda\mathfrak{C}}^T u_{\mathfrak{C}} - B^T\lambda).$$

Due to the handling of the Neumann boundary condition of the FDM, the generated system is nonsymmetric, i.e.,

$$A_{\Lambda\mathfrak{C}}^T \neq A_{\mathfrak{C}\Lambda}$$

which is also the case for the reduced system. Therefore, we have to use PGMRES as iterative solver.

5.5. *FEM discretization*

For the SOL geometry, the generated matrix by the FDM is a nonsymmetric matrix due to the handling of the inner Neumann boundary condition. Unfortunately, the FETI-DP method is only well analyzed for symmetric operators and not for nonsymmetric ones. Consequently, we have no proof that our problem generated with the FDM may be solved by the FETI-DP method even if we can apply it by using an algebraic formulation. Therefore, we want to pursue also the FEM which leads to a symmetric matrix for which we are on solid theoretical grounds.

The FEM generates the matrix in a different way than the FDM. First, we multiply a test function on both sides of the equation and integrate over the whole domain which gives the following weak formulation

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

By using the Gauss theorem, we get the following formulation

$$\iint_\Omega \epsilon(x,y)\left[\frac{\partial\phi(x,y,t)}{\partial x}\frac{\partial\psi(x,y)}{\partial x}+\frac{\partial\phi(x,y,t)}{\partial y}\frac{\partial\psi(x,y)}{\partial y}\right]dxdy$$
$$-\int_{\partial\Omega}\epsilon(x,y)\psi(x,y)\frac{\partial\phi(x,y,t)}{\partial n}ds=\iint_\Omega \rho(x,y,t)\psi(x,y)dxdy,$$

i.e.,

$$\iint_\Omega \epsilon(x,y)\left[\frac{\partial\phi(x,y,t)}{\partial x}\frac{\partial\psi(x,y)}{\partial x}+\frac{\partial\phi(x,y,t)}{\partial y}\frac{\partial\psi(x,y)}{\partial y}\right]dxdy=\iint_\Omega \rho(x,y,t)\psi(x,y)dxdy$$

As boundary condition we choose a Neumann condition.

Next we define the finite element space which will be used as trial and test functions. In this project, we consider the piecewise bilinear function spaces on rectangular elements which is the simplest one. By applying the finite element spaces on the weak formulation, we get a symmetric system for the Poisson problem with a Neumann boundary condition. The generating matrix by the FEM has at most nine nonzero elements such that

$$\begin{bmatrix}
-\frac{\epsilon_{i-\frac{1}{2},j+\frac{1}{2}}}{6}\left(\frac{\Delta x}{\Delta y}+\frac{\Delta y}{\Delta x}\right) & -\frac{\epsilon_{i,j+\frac{1}{2}}}{3}\left(2\frac{\Delta x}{\Delta y}-\frac{\Delta y}{\Delta x}\right) & -\frac{\epsilon_{i+\frac{1}{2},j+\frac{1}{2}}}{6}\left(\frac{\Delta x}{\Delta y}+\frac{\Delta y}{\Delta x}\right)\\
-\frac{\epsilon_{i-\frac{1}{2},j}}{3}\left(2\frac{\Delta y}{\Delta x}-\frac{\Delta x}{\Delta y}\right) & \frac{2(\epsilon_{i-\frac{1}{2},j}+\epsilon_{i+\frac{1}{2},j})}{3}\frac{\Delta y}{\Delta x}+\frac{2(\epsilon_{i,j-\frac{1}{2}}+\epsilon_{i,j+\frac{1}{2}})}{3}\frac{\Delta x}{\Delta y} & -\frac{\epsilon_{i+\frac{1}{2},j}}{3}\left(2\frac{\Delta y}{\Delta x}-\frac{\Delta x}{\Delta y}\right)\\
-\frac{\epsilon_{i-\frac{1}{2},j-\frac{1}{2}}}{6}\left(\frac{\Delta x}{\Delta y}+\frac{\Delta y}{\Delta x}\right) & -\frac{\epsilon_{i,j-\frac{1}{2}}}{3}\left(2\frac{\Delta x}{\Delta y}-\frac{\Delta y}{\Delta x}\right) & -\frac{\epsilon_{i+\frac{1}{2},j-\frac{1}{2}}}{6}\left(\frac{\Delta x}{\Delta y}+\frac{\Delta y}{\Delta x}\right)
\end{bmatrix}$$

In contrast the FDM has only five nonzero elements such that

$$\begin{bmatrix}
0 & -\epsilon_{i,j+\frac{1}{2}}\frac{\Delta x}{\Delta y} & 0\\
-\epsilon_{i-\frac{1}{2},j}\frac{\Delta y}{\Delta x} & (\epsilon_{i-\frac{1}{2},j}+\epsilon_{i+\frac{1}{2},j})\frac{\Delta y}{\Delta x}+(\epsilon_{i,j-\frac{1}{2}}+\epsilon_{i,j+\frac{1}{2}})\frac{\Delta x}{\Delta y} & -\epsilon_{i+\frac{1}{2},j}\frac{\Delta y}{\Delta x}\\
0 & -\epsilon_{i,j-\frac{1}{2}}\frac{\Delta x}{\Delta y} & 0
\end{bmatrix}$$

These additional nonzero elements imply more time to perform the matrix-vector multiplication operation which is the basic operation of iterative solvers. However, iterative solvers usually need less iterations for symmetric problems compared to nonsymmetric ones. Such problems can be solved with less effort by the CGM or the multigrid method. Therefore, the performance comparison between the FDM and FEM is justified and both methods have been implemented.

5.6. *Current status and future work*

We have modified the data structure of the original multigrid method in KinSOL2D to make it possible to reduce the number of cores being involved in the coarsest level solving to just a single core. In addition, we have implemented the matrix generation routine for the finite difference method (FDM) which has the same data structure as the finite element method (FEM) except that the number of nonzero elements differs. In combination with these two methods, we have implemented the FETI-DP method for the SOL geometry including the local problem solvers. This will give use the chance to compare the original multigrid method enhanced by a single core coarsest level solver with the domain decomposition method.

As the current project came to an end it is planned to continue the work in a follow-up project. There is still the debugging phase to be completed to be able to study the numerical performance of the FDM and FEM within the FETI-DP method. Such tests are scheduled on the Helios machine at IFERC-CSC.

6. Final Report on HLST project MGTRIOPT

6.1. *Introduction*

The physics code project GEMT is intended to generate a new code by combining two existing codes: GEMZ and GKMHD. GEMZ is a gyrofluid code based on a conformal, logically Cartesian grid. GKMHD is an MHD equilibrium solver which evolves the Grad-Shafranov MHD equilibrium using a physical description which arises under self consistent conditions in the reduced MHD and gyrokinetic models. GKMHD already uses a triangular grid, which is logically a spiral form with the first point on the magnetic axis and the last point on the X-point of the flux surface geometry. The solving method is to relax this coordinate grid onto the actual contours describing the flux surface of the equilibrium solution. Such a structure is logically the same as in a generic tokamak turbulence code. In the previous MGTRI project, we had developed a parallelized multigrid solver on a triangular grid for the parallelized GKMHD code.

The multigrid method is a well-known, fast and efficient algorithm to solve many classes of problems. In general, the ratio of the communication costs to computation costs increases when the grid level is decreased, i.e., the communication costs are high on the coarser levels in comparison to the computation costs. Since the multiplicative multigrid algorithm is applied on each level, the bottleneck of the parallel multigrid lies on the coarser levels, including the exact solver at the coarsest level. The feasible coarsest level of operation depends on both the number of cores and the number of degrees of freedom (DoF). In a parallel run we need at least one DoF per core and therefore, the number of DoFs of the coarsest level problem will be at least the number of cores (parallel coarsest level limitation).

To improve the performance of the parallel multigrid method, we reduce the number of executing cores for the coarsest levels to a single core (the simplest case) after gathering data from all cores on a certain level. This algorithm avoids the parallel coarsest level limitation. Numerical experiments on large numbers of cores show good performance improvement. But this implementation still needs improvement for very large number of MPI tasks. If we manage to reduce the number of MPI tasks for the multigrid method, the method might have a better scaling property.

Modern computer architectures have a highly hierarchical system design, i.e., multi-socket, multi-core shared-memory computer nodes which are connected via high-speed interconnects. This trend will continue into the foreseeable future especially as the number of cores per node is expected to grow significantly. Consequently, it seems natural to employ a hybrid programming model which uses OpenMP for parallelization inside the node and MPI for message passing between nodes. Expected benefits from a hybrid OpenMP/MPI programming model are manifold. On the one hand, OpenMP can make a more efficient usage of the shared memory system resources (memory, latency, and bandwidth) within a node. On the other hand, the extra communication and memory overhead of MPI tasks within a node should be avoided. As MPI tasks are replaced by OpenMP threads within a node, the total number of MPI tasks can be significantly reduced. This adds coarser granularity (larger message sizes) and allows increased and/or dynamic load balancing. Thus, we expect better scalability behaviour for the multigrid method when a hybrid programming model is used.

6.2. *Multigrid with a hybrid programming model*

We enhanced the original MGTRI multigrid algorithm which was based on an MPI implementation by adding OpenMP pragmas. Thus, we introduced a hybrid programming model using OpenMP/MPI. The resulting executable was run on the Helios machine at IFERC-CSC. As a first test we compared the execution times of the matrix-vector multiplication with the solution times of the multigrid method with Gauss-Seidel smoother as a preconditioner for the Preconditioned Conjugate

Gradient (PCG) method. For a certain problem size we used a fixed number of MPI tasks with a varying number of threads per MPI task. The total number of threads times the MPI tasks matched the number of cores being allocated for the parallel job. The problem size was given by the number of degrees of freedom (DoF) for the discretization of the domain.

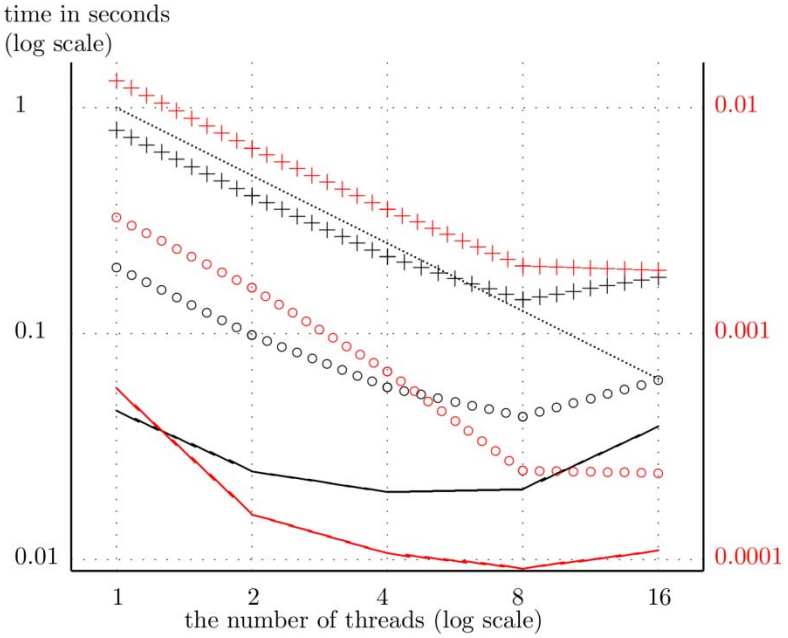


Fig. 41 The execution time of the matrix-vector multiplication (in red) and the solution time of the multigrid method (in black) in seconds as a function of the number of threads. The multigrid method with a Gauss-Seidel smoother is used as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method. The different domain sizes have 3.1M DOF (solid line), 12.5M DOF (○○○) and 50M DOF (+ + +) on 96 MPI tasks. The dotted line shows ideal scaling.

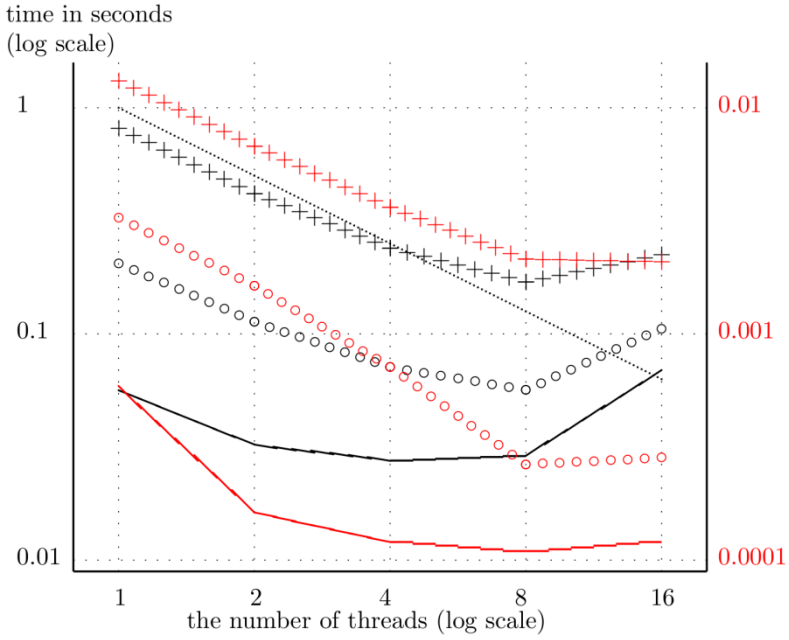


Fig. 42 The execution time of the matrix-vector multiplication (in red) and the solution time of the multigrid method (in black) in seconds as a function of the number of threads. Same parameters as in Fig. 41 except the fact that 384 MPI tasks have been used.

The results are depicted in Fig. 41 and Fig. 42 as a function of the number of threads, for a total number of 96 and 384 MPI tasks respectively. Independently of

the number of MPI tasks used it can be seen that an OpenMP parallelization becomes more efficient the larger the problem size is. However, for more than eight threads the performance seems to degrade. This might be due to the architecture of the computer node.

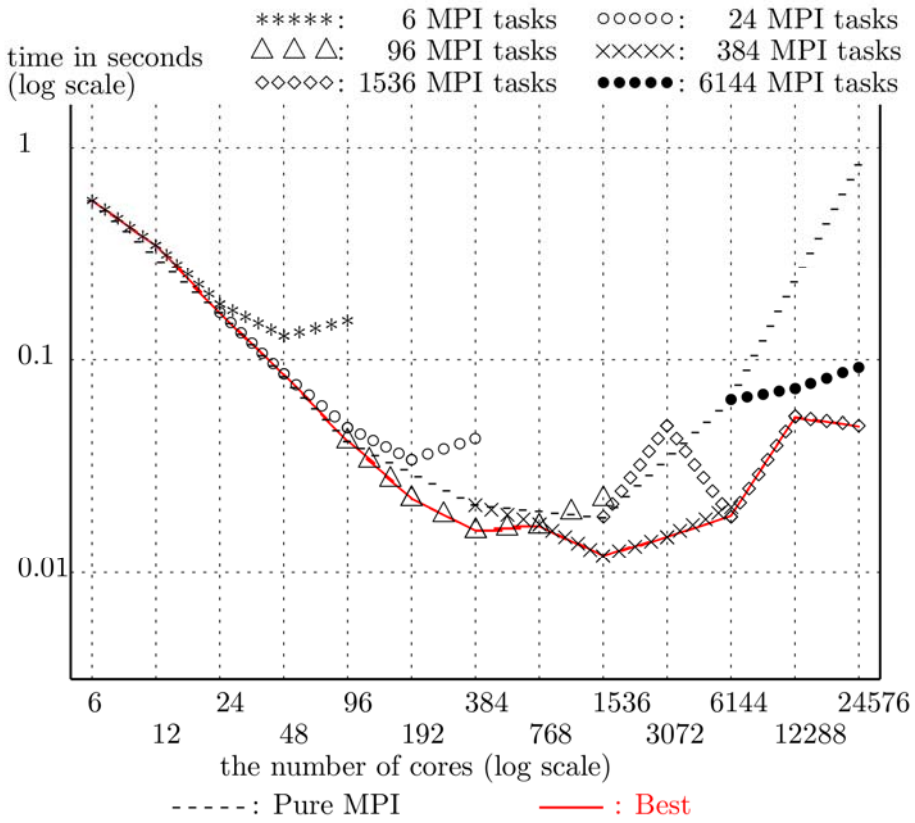


Fig. 43 The multigrid method with a Gauss-Seidel smoother is used as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method. The solution times of the multigrid method in seconds are shown as a function of the number of cores for a fixed domain size with 3.1M DOF. The number of threads varies from 1 to 16 per MPI task.

It is of key interest to compare the results of a pure MPI implementation of multigrid with its OpenMP/MPI counterpart to see which one is the most efficient for a given problem size. In the follow-up tests we fixed the problem size (strong scaling) to 3.1M DOF (see Fig. 43) and 12M DOF (see Fig. 44). The number of MPI tasks was sampled at the values: 6, 24, 96, 384, 1536, and 6144 and the number of OpenMP threads varied in steps of two from 1 to 16. The maximum number of cores used did not exceed 24,576. Each core hosted either one MPI task or one thread. If we compare pure MPI and hybrid OpenMP/MPI runs for the same number of cores we can see that the former (dashed line) performs better than the latter up to a certain number of cores. The exact number depends on the problem size, e.g. 96 cores for 3.1M DOF (see Fig. 43) and 384 cores for 12M DOF (see Fig. 44). In general, the turning point where the hybrid model becomes more efficient than the pure MPI model shifts to smaller numbers of cores, the smaller the problem size is. However, if more than eight threads are used per MPI task it can be seen again that the efficiency starts to degrade.

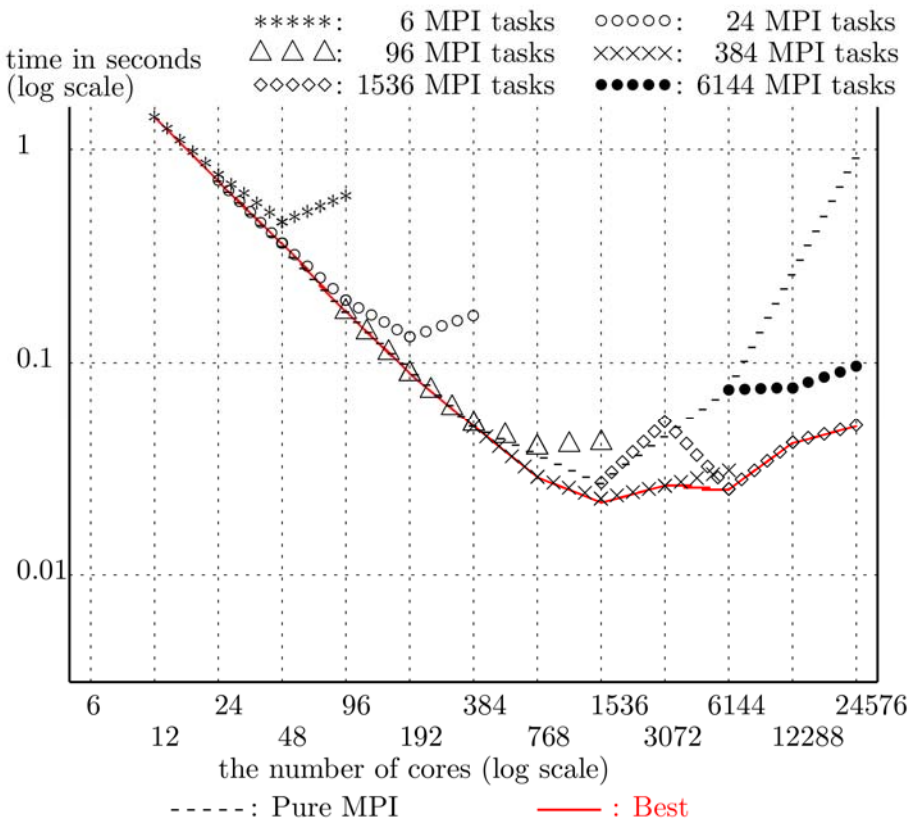


Fig. 44 The multigrid method with a Gauss-Seidel smoother is used as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method. The solution times of the multigrid method in seconds are shown as a function of the number of cores for a fixed domain size with 12M DOF. The number of threads varies from 1 to 16 per MPI task.

Next, we investigated the weak scaling property of the multigrid method which shows the solution time according to the number of cores when the number of DOF per core is fixed. For a weak scaling the number of MPI tasks or OpenMP threads respectively can increase only in steps of four. This is a consequence of the discretization of the domain by a triangular grid which can be only refined by factors of four. Therefore, we could set the number of threads only to four and sixteen. In Fig. 45 and Fig. 46 we show the results for a pure MPI run (solid line) and two hybrid runs with four (+ line) and sixteen threads (○ line) as a function of the number of cores being used. There should be always just one MPI task or OpenMP thread respectively per core. The difference between Fig. 45 and Fig. 46 is just the number of DoF which is 2000 DoF for the first and 8000 DoF for the latter.

Again, for small numbers of MPI tasks the pure MPI model is more efficient than the hybrid one. However, for a hybrid run the execution time does not increase as fast with the number of cores as it is the case for a pure MPI run. The smaller the DoF per core is, the sooner the hybrid model becomes superior to the pure MPI model. As the overhead of a hybrid model seems to scale with the number of threads, the runs with less numbers of threads overtake the pure MPI runs sooner. Nevertheless, one should be cautious with the sixteen thread results as they seem to have a general performance problem (see e.g. Fig. 41 and Fig. 42).

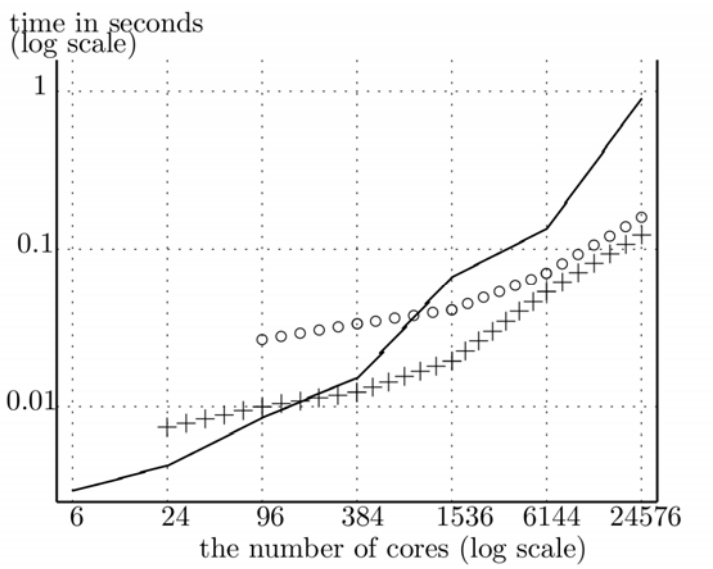


Fig. 45 The solution time of the multigrid method in seconds for 2000 DoF per core. The multigrid method is used as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method. Pure MPI (solid), 4 threads (+) and 16 threads (○).

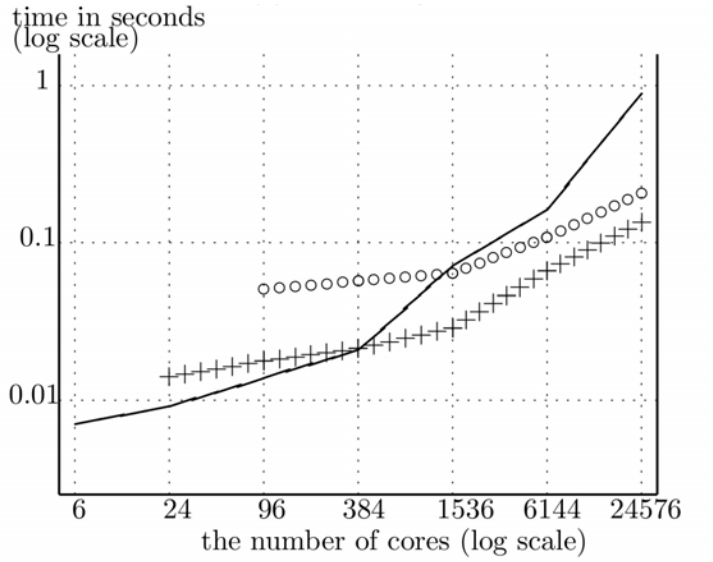
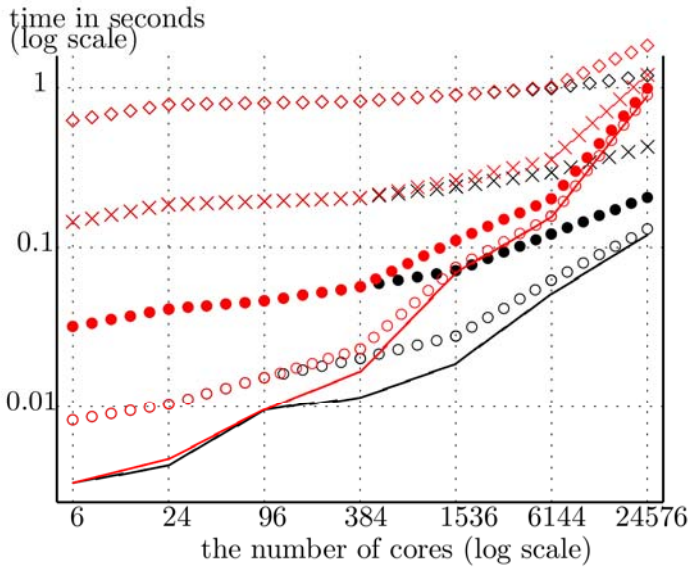
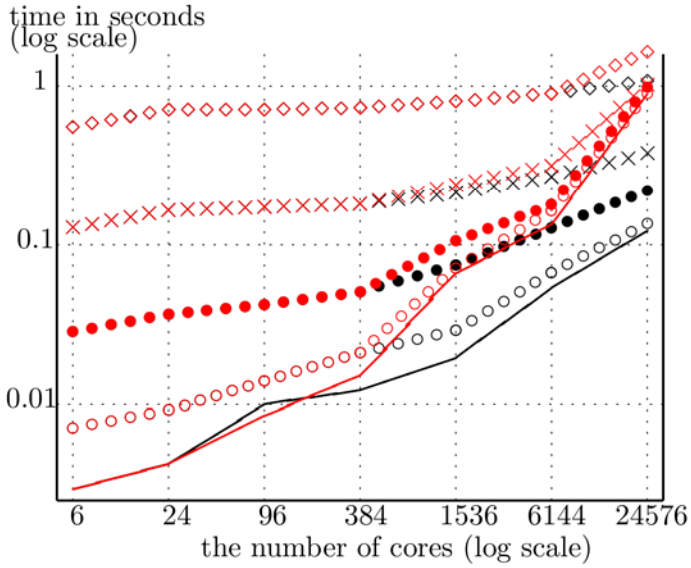


Fig. 46 The solution time of the multigrid method in seconds for 8000 DoF per core. The multigrid method is used as a preconditioner for the Preconditioned Conjugate Gradient (PCG) method. Pure MPI (solid), 4 threads (+) and 16 threads (○).

Finally, we summarize the weak scaling results for the selected test cases. The selected test cases had the following DOF: 2k, 32k, 32k, 130k, and 500k. In Fig. 47 we compare the pure MPI case (red) with the best performing hybrid case of 4 and 16 threads (black). For the pure MPI case the performance significantly degrades when the number of cores becomes large for cases with a small number of DOF per core. This situation improves with the hybrid model, i.e., problems with small number of DOF can be solved with a larger number of threads on a larger number of cores. The threads work on a shared memory inside a node which makes this method more efficient by avoiding part of the communication of the pure MPI method.



(a) As a solver with Gauss-Seidel smoothers



(b) As a preconditioner for the PCG method with Gauss-Seidel smoothers

Fig. 47 The solution time of the multigrid method with a Gauss-Seidel smoother in seconds as a function of the number of cores. The multigrid method is used as a solver (a) and a preconditioner (b) for the Preconditioned Conjugate Gradient (PCG) method. The best results of the hybrid model are in black, pure MPI runs are in red. The number of DoF per core: 2k (solid), 8k (○), 32k (●), 130k (x), and 500k (◊).

6.3. *Conclusions*

In the previous MGTRI project, we had developed a parallelized multigrid solver for the parallelized GKMHD code. Especially for a very high number of cores the pure MPI implementation seemed to be not optimal. Therefore, the parallel multigrid solver has been enhanced to a hybrid OpenMP/MPI model. In general, the turning point where the hybrid model becomes more efficient than the pure MPI model shifts to smaller numbers of cores, the smaller the problem size is. Therefore, an improvement can be achieved with the hybrid model in cases with a small number of degrees of freedom (DoF) per core on a large number of cores. However, if more than eight threads are used per MPI task the efficiency usually starts to degrade.

7. Final report on HLST project PARFS

7.1. *Introduction*

The Frascati theory and modelling group has developed, in the past years, PIC codes (HMGC, HYMAGYC) to study linear and nonlinear dynamics of Alfvénic type modes in Tokamaks, in the presence of energetic particle populations. While a large effort has been devoted to the parallelization of the "kinetic" part of the code, the corresponding field solver is still serial.

The aim of the PARFS project is to obtain a distributed version of the field solver (MARST) of the HYMAGYC code (a project developed within the EFDA-ITM task force). The current version of MARST evolves the MHD fields in time by solving the resistive MHD equations, which includes pressure tensor terms yielded by the kinetic module. HYMAGYC retains the particle nonlinearities while neglecting different toroidal mode number coupling. As a consequence, field solving reduces to the solution of a linear system of equations. It can be shown that the system size can scale up to n^3, with n being the toroidal mode number under consideration.

The distribution of memory and computational load among different nodes is crucial to allow the investigation of relevant mode numbers ($n \approx 40$) for this class of problems in ITER like configurations.

7.2. *Initial activities, problem assessment*

Given the limited project time, it has been decided in agreement with the project coordinator Gregorio Vlad to use the original HYMAGYC source code repositories for contributions from the PARFS project. This allows immediate visibility and effect of proposed changes.

The first changes stemmed from activities of "standardization"; namely improving the code's portability characteristics and enhancing adherence to modern Fortran standards by replacing obsolete constructs with current ones. The deployment on the Helios machine has been achieved.

The current built-in linear solver algorithm has been found to be inadequate to parallelization. Therefore, we proceeded towards testing third party solver packages (on a dump to file of the linear system), in order to identify appropriate numerical algorithms for the problem at hand: a block tridiagonal, though sparse unsymmetric complex system, non diagonally dominant.

According to our estimates, a major MARST code refactoring is necessary to have all of its relevant arrays being distributed among processors, so to enable an appropriate "natural" distribution of the linear problem data and retain memory scaling properties. Although useful and relatively straight forward, this would not fit into the time budget of the current project, as it would involve some major architectural changes that only the MARST developers can decide on. On the other hand, the linear solver assessment activity is more specific, as well as self-contained, so it has been decided to focus on this topic. As a consequence the full parallelization of MARST itself is left to its developers, or possibly as a separate future HLST project. However (see Sec. 7.4), on our request the project coordinator refactored the code to have at least the main (matrix related) arrays distributed.

7.3. Project Activities

After an extended search and small tests made with example programs, we identified the following solver packages as potential candidates: MUMPS [1] and WSMP [2].

The serial examples tested initially were still too small to get a realistic estimate of the packages behaviour on production-sized workloads. Nevertheless, for the test cases under consideration the appropriateness of the direct solvers could be confirmed. In contrast to some iterative methods, the direct solvers showed a robust

behaviour. According to previous HLST experience, scaling properties and memory requirements are usually problem specific, so further larger parallel scaling tests were necessary. In addition, we proceeded developing a testbed with a prospective solver interface for MARST. Such an interface should be capable of using all the solvers under investigation and solving linear systems stored on file by MARST. It should offer a unified way for controlling, benchmarking and studying the parallel behaviour of the considered libraries. Finally, this interface should be callable directly from MARST to avoid expensive disk I/O operations resulting from the writing/reading of the linear system to/from disk.

Especially the nonzero storage scheme of the sparse solvers being used for the interface development can prove useful in testing activities. Such a sparse storage scheme is less costly by a factor of (circa) seven than the current MARST dense matrices storage scheme (distributed or not). Indeed, MARST uses a dense representation of the diagonal blocks of the matrix. Unfortunately, to use a natively sparse representation of matrices within MARST is out of the scope of the project. But one may still resort to the disk-based approach when trying to reduce memory usage by loading in memory only the (previously dumped) strictly sparse matrix arrays. Since the representation of a matrix factorization can occupy an (at least) comparable amount of memory, minimizing the matrix representation overhead can be crucial for giving the solver enough memory to work with.

7.4. *Distributed MARST*

As we mentioned before, the original MARST field solver is serial. MARST “encodes” the matrix coefficients in eight three-dimensional arrays with 16 bytes per entry. Dimensions of these arrays depend on two run-time resolution parameters and some constants. After computing the matrix coefficients, only a fraction (between 1/7 and 1/8) of the aforementioned arrays locations are nonzero values. Upon our feedback, its author and project coordinator Gregorio Vlad has adapted the relevant part of the data arrays (that is, the matrix arrays) to be distributed. This has been a first step towards a distributed version of MARST (see Sec. 7.2), and enabled us to work with distributed matrices.

We proceeded into adapting our solver interface (PARFS) to the distributed matrix configuration. This involves encoding and decoding of the result and right hand side vectors from/to the own MARST format, as well as decoding of the input distributed matrix from its block form to the sparse one. Since it is envisioned that only a subset of the HYMAGYC parallel tasks should execute MARST, the solver is required to use a separate MPI communicator rather than the global one. With this setup, we proceeded into enabling larger test cases. Tests revealed that more work was necessary from the MARST developers side to achieve correctness in building the matrix in parallel and to tune the solver options to our needs.

7.5. *The MARST linear system; serial*

For the tests in the following section, the project coordinator provided us with an “equilibrium” file modeling an ITER-sized test case. However, this case is too large to be solved by the serial MARST solver on e.g. the Helios machine. The test case is made up of $NR \approx$ 1.4E6 equations, $LOC \approx$ 4.5E9 dense matrix entries, and $NNZ \approx$ 5.5E8 nonzeroes. The resulting matrix (>72 GB) would not fit on a single Helios node's memory (64 GB), and therefore, could not be solved. Instead the interfaced, distributed solver MUMPS enables HYMAGYC to run large cases, just as this one.

7.6. *The MARST linear system; parallel*

As mentioned before, on our request, the project coordinator has restructured the code in a way that the matrix arrays are distributed in the radial direction on a number of computing tasks (*TN*) and each point in the radial direction contributes to a number of tridiagonal blocks of the matrix. As a result each task hosts roughly the

same amount of local “radial points”, and therefore, contributes with a comparable amount of matrix nonzeroes.

To compute the matrix coefficients without excessive communication, it is advantageous to store two additional “*guard cells*” in the radial direction for each task. The chosen distribution favors partitioning of the matrix across computing tasks either “by rows” or “by columns”. These two partitioning schemes are the most common ones for distributed memory solvers, but require some low level sparse matrix computations (e.g.: sorting, format conversion), which are dealt with by using the “librsb” GPL licensed library [3].

7.7. *Distributed MARST with the MUMPS solver*

The distributed memory solver we tested was MUMPS (version 4.10). It is supported on Helios by CSC via the “module” system. The default options of the solver caused some problems (e.g.: uneven memory usage) and had to be tuned accordingly.

To show a sample of the strong scaling property we report measurements of the solution of an ITER-sized test case with the pure MPI (with OpenMP disabled) version of MUMPS, ran with different sets of parameters which are listed in the following:

- *NN* (Nodes Number) = {2,4,8,16}
- *TPN* (Tasks per Node) = {1,2,4,8,16} , and thus, obtaining different cases having
- *TN* (Number of Tasks) = {2,4,8,16,32,64,128,256}

By “task” we denote “MPI task”, and by “node” we denote “physical computing node”. The results are sorted by factorization time in Table 7. The cases with *NN* = 2 and *TPN* = {1, 2,16} crashed by running out of memory. While this is understandable for the *TPN* = 16 case, it is not for the remaining two cases.

NN	***TN***	***TPN***	***MAXMEMPT*** (MB)	***MEMTOT*** (MB)	***FT*** (s)	***ST*** (s)
16	256	16	636	127371	674.05	0.60
8	128	16	1116	110362	684.07	0.38
4	64	16	2158	102740	685.49	0.41
16	64	4	2113	102957	686.28	0.38
8	64	8	2135	102844	691.45	0.39
8	32	4	4052	98098	725.65	0.46
16	32	2	4352	98002	727.61	0.57
4	32	8	4263	98741	734.36	0.74
16	16	1	8648	95348	806.79	0.86
4	16	4	7563	95662	811.63	0.87
8	16	2	8984	95818	816.47	0.82
2	16	8	9138	95263	837.84	2.01
8	8	1	17024	94750	973.68	1.41
4	8	2	17299	94923	985.17	1.67
2	8	4	17656	94692	999.35	2.31
4	4	1	32509	87046	1336.62	2.62

Table 7 Execution time in seconds and consumed memory in MB for the MUMPS package used for MARST. Rows are sorted by increasing factorization time (*FT*). Please note that when performing simulations made of thousands of steps, repeated solution time (*ST*) dominates the overall solver time.

Across all runs, the maximum memory used by MUMPS per task (*MAXMEMPT*) ranges between 0.6 and 32 GB and the total memory used (*MEMTOT*) ranges between 90 and 120 GB. The best time for factorization (*FT*) was encountered on *NT* = 256: 674 s; the worst time was encountered on *NT* = 4, *NN* = 4: 1336 s. This indicates a factorization scaling of only a factor of two, and seems to stem from the “Sequential root node” option chosen. Using the “Parallel root node” option instead

resulted in unexpected crashes (segmentation faults). The optimal time for solution (*ST*) was of 0.38 s; the worst was 2.6 s. Therefore, there is at least a scaling factor of five in the solving phase.

7.8. *Assessment of the WSMP solver*

Having obtained a distributed matrix representation in MARST, it became possible to interface to the MUMPS solver library in a proper manner. This in turn enables MARST to run cases much larger than before, and this is already an improvement. However, the performance results, in particular the scaling property of MUMPS are unsatisfactory. As it was not obvious to us whether this expressed a general problem of direct solvers or a particular weakness of MUMPS we decided to further test the Watson Sparse Matrix Package (WSMP) from IBM.

However, WSMP exhibited a problem of a different nature. Even on the cases considered (smaller than the mentioned ITER sized), it uses excessive amounts of memory in the analysis phase. Namely, on small problems, the memory usage exceeded by roughly ten times that of MUMPS, reaching the factor of a hundred over the bare "COO representation"[2] of the matrix. As a matter of fact this prevented us from adequately testing large systems. Over many years we have established good relations to WSMP's lead developer, Anshul Gupta. This enables us to report problems and obtain quick response times. First analysis of the problem by Anshul Gupta showed that our test matrices are quite unusual compared to the typical problems being normally treated with WSMP. The analysis phase of WSMP is not well optimized for such a case. Hence, major changes of WSMP is necessary to overcome this bottleneck. Work is in progress but will take some time. Nevertheless, we are still interested in the outcome even though this will come after the project time is exceeded.

7.9. *Conclusions and recommendations*

The HYMAGYC code has been ported to the Helios machine. Changes in the source code were made to achieve a more "standardized" version. The field solver (MARST) now uses a distributed representation of the linear system. An ITER-sized test case has been constructed and successfully solved. Previously it was impossible to treat such cases. After some interfacing work, it was possible to solve the system with the parallel MUMPS package. Although this is already a big step forward, the achieved speed-up of a factor of two to five was not satisfactory. As an alternative we tried the parallel IBM WSMP package, but this revealed a problem with its memory usage. Contact with the lead developer, Anshul Gupta, has been established. Currently he is working on the problem, scheduled to be fixed in the near future.

At the moment it is not clear to us whether the bad speed-up scaling is peculiar to MUMPS or if this expresses a general problem of direct solvers with the ITER-like test cases of HYMAGYC. An amended version of WSMP which could run such a test case should give new evidence to answer that question. However, also parallel iterative solvers should be taken into account. Unfortunately previous experience with MARST has shown that finding an appropriate method is far from trivial: for instance, special care has to be taken when selecting efficient preconditioners for use with e.g. the GMRES iterative method. A good testbed for such investigations may be provided by the comprehensive iterative solver package PETSc [4] which gives access to several different preconditioners.

The solver interface code has been provided to the principal investigator with the appropriate usage instructions.

[2] "COO representation" here means the common two four-byte coordinate indices (row, column) plus one sixteen bytes value per nonzero.

7.10. ***References***

[1] *MUMPS: a MUltifrontal Massively Parallel sparse direct Solver* http://mumps.enseeiht.fr/
[2] *WSMP: Watson Sparse Matrix Package*
http://researcher.watson.ibm.com/researcher/view_project.php?id=1426
[3] librsb: A *shared memory parallel sparse matrix computations library for the Recursive Sparse Blocks format* http://sourceforge.net/projects/librsb/
[4] PETSc: *Portable, Extensible Toolkit for Scientific Computation*
http://www.mcs.anl.gov/petsc/index.html

8. **Report on HLST project ITM-ADIOS-2**

8.1. *Introduction*

Lengthy, computationally intensive plasma physics simulations require periodic on-disk storage of intermediate results. There can be several reasons for this: prevention of data loss due to accidental hardware or software failures, monitoring/processing the results of an ongoing simulation or a restart possibility for prolonging the simulation execution across multiple runs, beyond the batch system constraints for single executions. However, as a consequence of the ever increasing ratio between computing power to Input/Output (I/O) subsystems throughput, a relevant fraction of (wall clock) time is consumed by "waiting" for I/O operations to complete. Usage of "parallel I/O" techniques may reduce this time by increasing the I/O throughput and thus, the efficiency in resource usage. Also techniques overlapping I/O and computation, so-called "staging" techniques, are known to improve I/O efficiency.

The project ITM-ADIOS-2 has been launched by ITM (Integrated Tokamak Modelling) which is an institutional effort devoted to the development of a standardized environment for Tokamak simulations. Such effort involves several simulation codes, supported by a geographically distributed community of physicists and institutions. Given the current heterogeneity of the involved codes, solutions with a high degree of compatibility are preferred. Results of the present investigation are targeted at this community for which simplicity of possible solutions is of importance.

The primary goal is to assess the publicly available "ADIOS" software library (see [1]) in terms of obtaining better I/O efficiency. ADIOS [1] is an open source software library for large file system I/O developed at the Oak Ridge Laboratories. It offers a single interface to many I/O methods; among its goals are interoperability and high speed I/O. This gives the basics for fast and reliable checkpoint/restart.

This project is the successor of a first ITM-ADIOS project, carried on by HLST during 2011–2012 [2]. Back then, file systems performance of the Helios and HPC-FF computers had been characterized both in serial and parallel usage. The HPC-FF computer installation (hosted at the Jülich Super Computing Centre, Germany) served the plasma physics community until mid-2013, and consisted of 1080 nodes, 8 cores each, for a total of 8640 cores. Using techniques developed on HPC-FF, the study had been extended to the Helios machine, hosted in Rokkasho, Japan and put into operation beginning 2012. Helios is considerably larger than HPC-FF: 4410 nodes, 16 cores each for a total of 70,560 computing cores, of which a maximum of 65,536 are allowed to participate in a single parallel job. In the past, parallel I/O had been studied using the ADIOS-1.3 library. Much of the effort went into studying the MPI-IO interaction with user-available LUSTRE parameters. In this way, an elaborated benchmark program able to achieve close to peak I/O performance was developed. With optimized parameters, it achieved close to 15 GB/s on HPC-FF and 35 GB/s on Helios. This is respectively ≈ 75% and ≈ 50% of the maximum bandwidth. The project ITM-ADIOS provided ready to use examples for I/O code. However, most of that study was carried out using the aforementioned, elaborated benchmark program quite different from those examples, and with a rather complicated structure.

The current project aims primarily at improving our techniques in using ADIOS as simply and effectively as possible and transferring the acquired knowledge to ITM scientists. The following sections present activities that have been carried out, together with a discussion of new results and a perspective on near-term work. A technical report with recommendations for users is also in preparation.

8.2. *Aspects of the Helios file systems*

Helios features three large file systems: $PROJECT, $WORKDIR, $SCRATCHDIR; of whom the first is designated for long time keeping of data, and the last two are intended to be used by applications with large I/O needs. Additionally, each node

features a local Solid State technology based disk (SSD) capable of ≈ 180 MB/s I/O, mounted on the /tmp directory. Experiments conducted in the previous ITM-ADIOS project gave us a solid understanding about the $WORKDIR and $SCRATCHDIR I/O performance. For this instance we used a serial program to perform binary I/O based on the Unix 'dd' tool.

The synchronizing option was switched on to ensure that the measured performance would be effective (for updated measurements see Section 8.3). Effective performance indicates what the I/O hardware (LUSTRE installation + network) is capable of by measuring the speed at which data travels to disk. User perceived (*unsynchronized*) performance might be higher because of operating system level caches keeping the data buffered for disk transfer. However, unsynchronized I/O does not give guarantee of the data being actually stored on disk — a crash might still occur on a subsystem, and data would get lost. Moreover, unsynchronized I/O speed is influenced by the node's free available memory or network traffic situation at a given moment. In judging the quality of a hardware installation, effective performance may be more interesting to look at, whereas in other contexts user perceived performance may prove more interesting.

The next section illustrates results obtained in recent experiments.

8.3. *Serial I/O Performance Assessment on Helios*

As reported in [2], two LUSTRE parameters are important for serial I/O: *LUSTRE stripe size* (L_{ss}, measured in MB) and *LUSTRE stripe count* (L_{sc}, or the number of *storage targets* employed for a given file's storage). The experiment we report here is based on writing repeatedly 4 GB of data to files created using different values for L_{sc} and L_{ss}. The best observed serial I/O performed at ≈ 520 MB/s on the $SCRATCHDIR file system, and occurred while using L_{ss} = 32 MB and between 2 and 8 for L_{sc}. The worst obtained result was roughly half of that: ≈ 270 MB/s, and occurred with L_{sc} = 1. Similarly, suboptimal results (≈ 300 MB/s) occurred when using L_{ss} >= 128 MB or L_{sc} >= 32. As one can verify from Fig. 48, these values are quite similar on both $WORKDIR and $SCRATCHDIR.

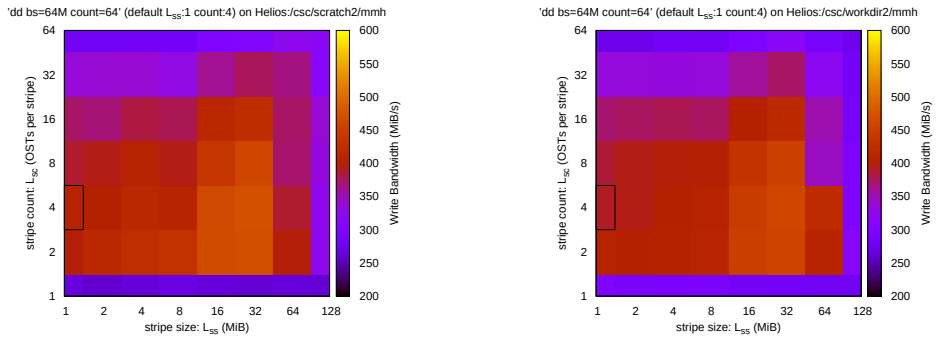


Fig. 48: Serial writing I/O performance on Helios $SCRATCHDIR and $WORKDIR file systems.

Although the general picture matches that from the old results in Ref. [2], one may observe a generalized degradation of performance, mostly in the range of 10–20% (see Fig. 49).

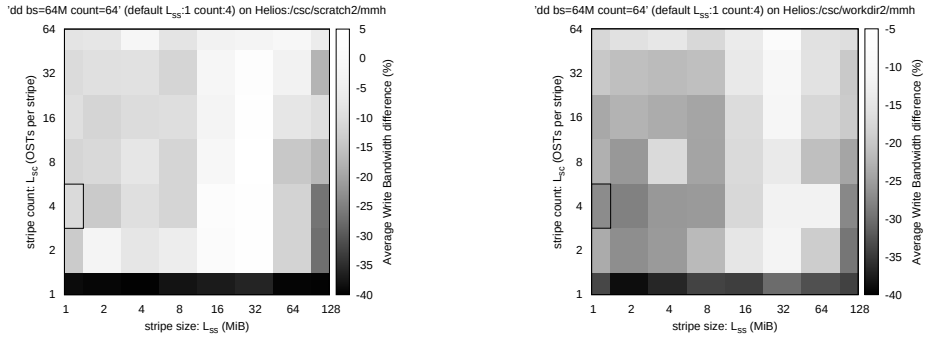


Fig. 49: Serial writing I/O performance on Helios $SCRATCHDIR and $WORKDIR file systems, relative difference to the results obtained in our previous study [2].

The complete characterization of this degradation has been submitted to CSC via the ticket system and the problem is being studied by system administrators (CSC ticket #1664).

8.4. *I/O Performance for a trivial parallel job*

Parallel I/O involves all computing resources: computing nodes, the network interconnect, and the storage area systems. We established a simple test to measure the performance of the I/O system under a heavy "parallel" load, though avoiding most of the user-space software complexity. Namely, we launched a job consisting of independent parallel processes writing to independent files, without using the MPI message interface, relying only on the SLURM job manager. In our experiment, each process wrote 512 MB to a uniquely named file, using the synchronization option, so effective performance was studied.

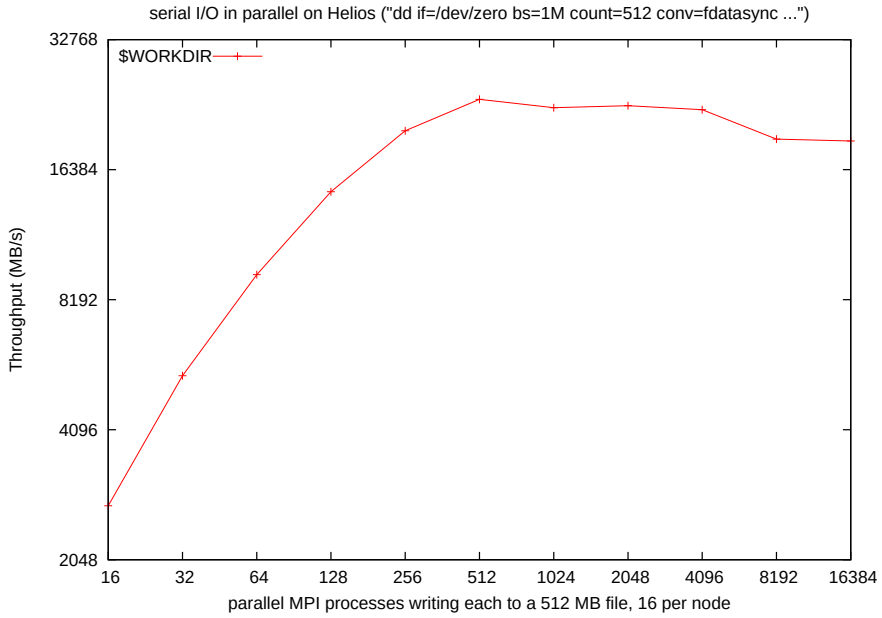


Fig. 50: I/O writing performance on $WORKDIR obtained using 1 to 1024 nodes performing serial I/O. LUSTRE parameters were chosen according to the "rule of thumb" we had extrapolated in [2].

The results show (see Fig. 50) that after a near-to-linear ascending phase from 16 to 64 parallel processes (1 to 4 nodes), the throughput stabilizes at ≈ 32 nodes, giving a plateau located at ≈ 22 GB/s. Given the simplified conditions under which this test has been performed, we regard these results as near to the optimum.

8.5. *Possible usage of /tmp for fast I/O*

Throughput results of the previous section are limited to ≈ 22 GB/s, which is reached for a quite small number of 512 MPI processes out of a maximum of 65,536 (or, 32 nodes out of 4096). Instead the system's /tmp file systems might be used for higher throughput rates. Indeed as /tmp is local to each node, its total storage capacity grows linearly with the number of nodes. Being each local file system capable of ≈ 180 MB/s (or, ≈ 11 MB/s per process), one would expect the capacity of e.g. a 1024 node job to be ≈180 GB/s.

The parallel environment erases the user-generated contents of /tmp after each job's termination. Although this is a perfectly reasonable policy, it singles out a straight-forward implementation of checkpoint-restart based on copying data after job termination. A hypothetical feasible solution based on synchronous I/O in /tmp followed by an asynchronous (that is, overlapped with computation) transfer to $WORKDIR/$SCRATCHDIR would have the potential of boosting performance over $LUSTRE by 900% (tenfold speed-up). The current amount of /tmp space made available by CSC is ≈ 50 GB. Results of a performance test using the “dd” utility are as predicted and are displayed in Fig. 51. Notice the comparison with $WORKDIR writing speed: at 2048 processes (128 nodes) break-even occurs and writing to /tmp starts being (overall) faster.

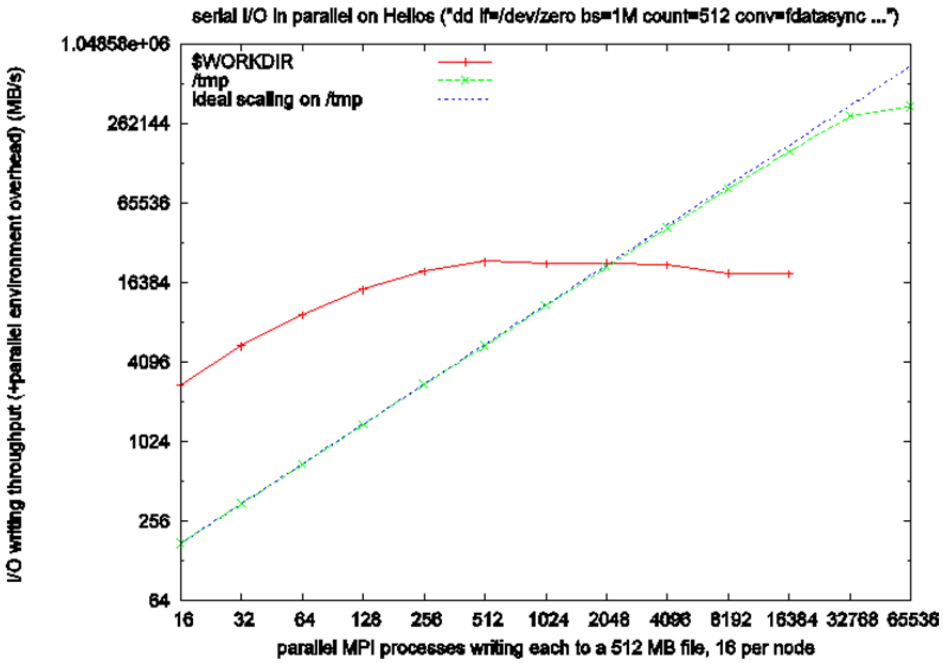


Fig. 51: Serial I/O from a parallel job writing to /tmp and $WORKDIR compared. The figure shows total written data divided by the time it took to run the writing job. Therefore, a small parallel environment overhead is also being measured (up to 10% at 32,768 processes). Notice the almost linear scaling of the aggregated I/O capacity. The largest run (65,536 processes) was slower than expected due to a parallel environment synchronization failure (confirmed by CSC ticket #1631; not expected to reoccur).

8.6. *Parallel I/O Performance Assessment on Helios*

Results in this section have been obtained by using ADIOS in an MPI enabled program. Running a few significant cases with the ADIOS-1.3 based IABC benchmark from [2] suggested that last project's I/O performance can be re-obtained. However, due to API incompatibilities, the source code of the old IABC had to be slightly adapted to ADIOS-1.5. Instead of updating the old C benchmark, we worked to obtain a minimal, yet representative Fortran based benchmark aimed at

establishing a “model ADIOS usage”. After experiments and communication with ADIOS developers, we found out a solution which seems to be simple and efficient enough to be proposed to the ITM community.

We ran our benchmark with an increasing number of MPI tasks (or *MPI processes*) P_{ntasks}, always with 16 tasks per node, and with either 128, 256 or 512 MB of data from each task (P_{dpt}) being written to a single file, as in:

- P_{ntasks} (Number of Tasks) = 16, 32, 64, 128, 256, 512, 1024, 2048, 4096
- P_{dpt} (Data per Task) = 128, 256, 512 MB

Our pilot tests with the new solution gave the performance shown in Fig. 52.

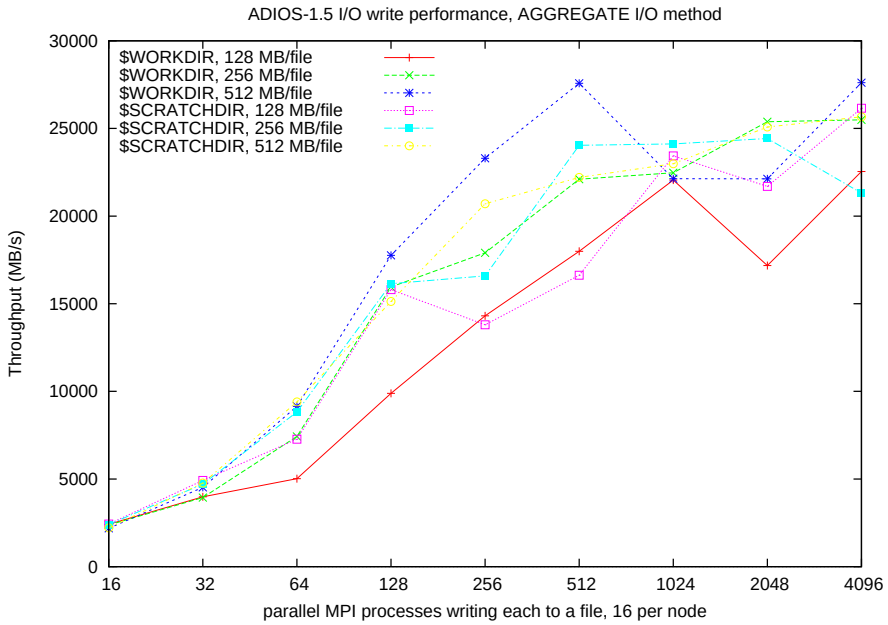


Fig. 52: ADIOS I/O write performance on Helios, for different I/O size per task and total number of MPI tasks. Notice overall write capacity saturation being already approached at a small number of 512 tasks (32 nodes).

We can see that similarly to the results in Fig. 50, saturation is being reached already at 512 tasks (32 nodes) with ≈ 25 GB/s. As expected, the cases with larger P_{dpt} reached saturation with less MPI processes. It is interesting to note that unlike our past investigation ([2]), here it is ADIOS itself which chooses the LUSTRE striping parameters (actually, $L_{sc} = 1$, $L_{ss} = 1$ MB). Reducing the number of files to one per node may be beneficial (see [2]), but apart from this we do not expect significant improvements to be within easy reach.

8.7. *ADIOS 1.5 and staging*

ADIOS-1.5 offers three methods ([3], Sec. 2.3.8), based respectively on the DataSpaces system, DIMES, and Flexpath transport systems. According to documentation and clarification from the ADIOS authors, these methods are not in production yet.

8.8. *Ongoing and foreseen work*

We have established contact with Tobias Goerler, member of the GENE code group: a solution based on our best practices in using ADIOS will be tested within GENE. Initially, we plan to make sure the performance obtained in the previous section can be replicated; then, we will study selected features of ADIOS that can improve characteristics of reliability, performance and robustness, according to users demands.

We are also in the process of making the ADIOS library available on the ITM gateway cluster (versions 1.3.1 and 1.5.0). It is the primary machine for ITM code development and integration. Having a reference ADIOS installation will allow ITM users to test e.g. our ADIOS based solutions and use them further in production on Helios. In addition, it is also planned to make our benchmark code and examples available on the ITM cluster, in order to ease knowledge transfer.

Further tests are necessary to make sure that the performance obtained with the experiment in Section 8.6 can be obtained also with the data layout and conditions of real codes (P_{ntasks}, P_{dpt}, …). In [2] we found that reducing the amount of files to no more than one per node is also beneficial: this is a further interesting case to study, as improvements may be expected.

8.9. *Conclusions*

A reasonably simple and robust method for using the most recent ADIOS 1.5 version efficiently on Helios has been established. A subset of ADIOS features (the one relevant for us) has been studied, and I/O performance properties on Helios have been characterized. We are working now on transferring our knowledge and experience to major ITM codes, and to fine-tune the solution to their needs. Once the project terminates, we expect having reached a good knowledge of ADIOS for our needs (fast checkpoint-restart) and having our codes of reference taking advantage of its usage.

8.10. *References*

[1] “ADIOS: *Adaptable IO System (ADIOS)” library website*, Oak Ridge National Laboratory, Accessed December, 2013, http://www.olcf.ornl.gov/center-projects/adios
[2] *"ITM-ADIOS Project Final Report"*, Michele Martone, http://hlst-efda.eu
[3] *"ADIOS 1.5.0 User's Manual"*, Oak Ridge National Laboratory, Accessed December, 2013, http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.5.0.pdf

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

9.1. ***Introduction***

Simulation of x-mode reflectometry using a finite-difference time-domain (FDTD) code is one of the most popular numerical techniques used, as it offers a comprehensive description of the plasma phenomena. This method requires however, to keep the error to a minimum, a fine spatial grid discretization, which also implies a high-resolution time discretization to comply with the CFL stability condition. As a consequence, simulations, especially in two spatial dimensions for x-mode, can become quite demanding on computational time. Also, as the size requirements increase in an effort to simulate large devices as JET or ASDEX Upgrade or next generation machines like ITER, memory demands become constringent. This project was devised to circumvent these questions by implementing a parallel version of the x-mode REFMULx code, developed at Instituto de Plasmas e Fusão Nuclear (IPFN).

9.2. ***Serial code optimization***

REFMULx is a serial code that simulates x-mode reflectometry in magnetized plasmas. It evolves Maxwell's curl equations using a FDTD Yee schema [1], where a transversal electric (TE) propagation mode is considered. The plasma effects are included via the response of the electron current density *J* to the electric field *E* of the probing wave. Such plasma-wave coupling implies solving as well for *J* at each time step. Two *J* solvers are available, namely, a modified Xu and Yuan [2] and a new kernel proposed by Després and Pinto [3]. Berenger's perfectly matched layer (PML) [4] is used as boundary condition at the limits of the domain.

As the first step to guide the optimization strategy, the profiling of the code was made, revealing the numerical kernel to be the hot-spot cost wise, as expected. Measurements with Gprof made on Helios (IFERC-CSC) yielded the figures listed in the Table 8 for the Xu and Yuan (XY) kernel. Similar results were yielded for the Després and Pinto (DP) kernel. From these, it is clear that the parallelization effort needs to focus on the top five functions listed, which represent more than 99% of the total cost.

Before undertaking any attempt to parallelize these regions of the code, their serial performance was analyzed. While doing so, it was noticed that the bi-dimensional arrays used in the main four kernel functions listed in the table above were declared using a general array of pointers-to-pointers construct in C-language. This, by default does not exclude the possibility of pointer aliasing, which means that writes through a given pointer can effect the values read through any other pointer available in the same context. After checking that indeed this was not the case in the aforementioned functions of REFMULx, the non-aliasing property of those pointers was declared explicitly using the *__restrict* keyword in order to restrict their scope, which in principle aids the compiler in its task of cache optimization. Doing so on GNU C-compiler (gcc-4.4.6) led to little effect, with the elapsed time for the standard REFMULx test case changing from the original 14.404 s to 12.452 s, when declaring the restricted scope of the pointers, as measured on Helios. With the Intel compiler (icc-13.1.3), the situation was different. In this case, the runtime of the same test-case went from 14.606 s to 2.61 s, therefore, yielding a total factor of 5.6 speed-up. Further activities on the topic of serial optimization are currently under way. Their status is reported in section 9.4.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
34.61	30.94	30.94	700	44.19	44.19	calcHzFieldKXY
23.19	51.66	20.72	700	29.61	29.61	calcEyFieldKXY
21.90	71.23	19.57	700	27.96	27.96	calcJxyFieldKXY
18.93	88.16	16.92	700	24.18	24.18	calcExFieldKXY
1.30	89.32	1.16	700	1.66	1.66	addFLDMTRX
0.06	89.37	0.05	22	2.27	2.27	initFLDMTRX
0.02	89.39	0.02	4	5.00	5.00	escrvField
0.01	89.40	0.01	1	10.00	10.00	setNe
0.00	89.40	0.00	1400	0.00	0.00	setTCut
0.00	89.40	0.00	700	0.00	0.00	repointJxyFields
0.00	89.40	0.00	700	0.00	0.00	setSinLn
0.00	89.40	0.00	2	0.00	0.00	escrvTCut
0.00	89.40	0.00	2	0.00	0.00	initTCut
0.00	89.40	0.00	2	0.00	0.00	setField
0.00	89.40	0.00	1	0.00	0.00	cmdln
0.00	89.40	0.00	1	0.00	0.00	escrvRef
0.00	89.40	0.00	1	0.00	0.00	escrvStatus
0.00	89.40	0.00	1	0.00	0.00	initEnvNe
0.00	89.40	0.00	1	0.00	0.00	mulEnv
0.00	89.40	0.00	1	0.00	0.00	setNeFrm
0.00	89.40	0.00	1	0.00	0.00	setPMLB
0.00	89.40	0.00	1	0.00	0.00	setPMLL
0.00	89.40	0.00	1	0.00	0.00	setPMLR
0.00	89.40	0.00	1	0.00	0.00	setPMLT
0.00	89.40	0.00	1	0.00	0.00	zFrmHorn

Table 8 Profiling of the serial code REFMULx using Gprof with gcc on Helios (small test-case).

9.3. *Parallelisation scheme: roadmap and milestones*

After the initial serial optimization of REFMULx, the project proceeded towards its main goal, namely the implementation of its parallel version and the assessment of the resulting performance gains. Since the final solution is to be applied to an envisaged three-dimensional code that will require much more resources (cores), the use of the MPI standard was proposed for the parallelization. However, in order to gain experience with the code and, at the same time, obtain improved results within a smaller time investment, it was decided to start parallelizing the code's five hot-spot routines using OpenMP threads. The plan is to come up with a hybrid version OpenMP/MPI parallelized in two-dimensions, which has the additional benefit of being in line with the present tendency of increasing the number of cores per processor/socket. Once developed, it will further provide a good comparison to the ultimately planed bi-dimensional parallelized pure MPI version.

So far, the one-dimensional parallel OpenMP has been implemented. Since both the XY and DP kernels are largely symmetric in their two spatial directions, both in terms of the numerical stencil, as well as of the domain grid-count, the most efficient way to proceed is to apply the thread-based parallelization to the slow-varying index (outer loop). Doing so on the five functions listed at the top of Table 8 led to the strong scaling results on Helios shown in Fig. 53, for the GCC (top) and ICC (bottom) compilers. The solid red curve shows the total runtime of REFMULXp with different number of threads. The horizontal dashed blue and light-blue lines indicate the runtimes of the original and the "restricted" (see section 9.2) serial code, respectively. The gray curves show the time spend on each of the four kernel functions listed at the top of Table 8, which constitute the code's algorithmic kernel.

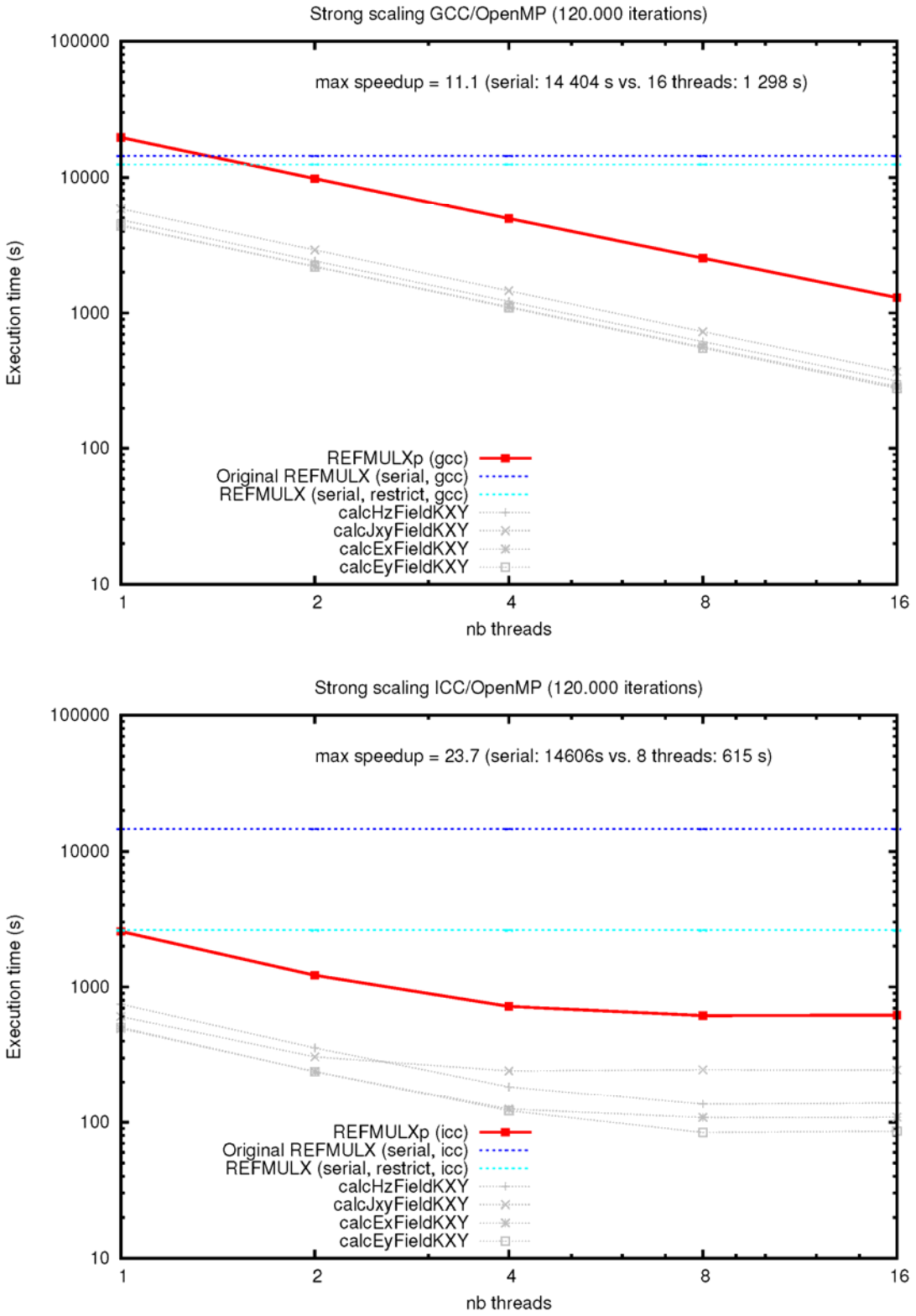


Fig. 53 Strong scaling (problem size kept fixed) of REFMULx's standard test-case on Helios. On the top, the results obtained using the GNU C compiler (v.4.4.6) are shown. On the bottom, the same is repeated using the Intel C compiler (v.13.1.3). The maximum speed-up, compared to the original serial code, was achieved when using ICC on 8 threads, which yielded a value of 23.7.

The threaded code compiled with GCC shows some overhead compared to the serial version, which is nevertheless, largely compensated when multiple threads are used. Within a single node of Helios, which has 16 cores, the scaling is perfectly linear. The maximum speed-up is therefore obtained for 16 threads, for which the code runs in

less than 1/10 of the time of the original serial version. The threaded code compiled with ICC shows a different behavior. First, there is no apparent overhead caused by using threads. Second, as mentioned in section 9.2, the effect of restricting the scope of the pointers clearly pays-off, as the difference between the blue lines shows. Finally, the scaling of threaded code starts to degrade beyond four threads, but we must point out that, in absolute terms, running with the ICC code on four threads is faster than with the CGG code on 16 threads. A maximum speed-up factor of almost 24 was obtained with eight threads.

As outlined in the previous paragraph, the next step is to parallelize the second dimension using MPI, to yield a hybrid OpenMP/MPI parallel version of REFMULx. In such a case, what makes sense is to move the thread-based parallelism to the fast-varying index (inner loop) and to decompose the slow-varying index into MPI tasks. After that, the plan is to move towards a purely bi-dimensional MPI domain decomposition, making use of the Cartesian topology provided within the MPI standard. Both hybrid and pure MPI parallel versions will be compared against each other in terms of their scaling properties. It is noteworthy that the saturation observed here for a modest number of threads constitutes no obstacle for further domain decomposing the other spatial dimension of the grid. The reason is simply that the idea is to deploy the parallelization scheme to the future three-dimensional version of the code, which will have three orders of magnitude bigger datasets, and therefore, also similarly higher number of floating-point operations (FLOP).

9.4. *The roofline model*

In an attempt to assess the node-level performance of the REFMULx's kernel, it was decided to apply the roofline model [5], which was developed as a method to quickly quantify the performance bounds for a given combination of architecture x algorithm x implementation. It basically states that the time to solution for a given algorithm is the maximum between the time it takes to transfer the data (bandwidth) and the time needed to perform the floating-point operations (FLOP). If the former is the slower process, the algorithm is said to be bandwidth-bound. Otherwise, it is said to be compute-bound. Typically the first step is to build the roofline model for a given architecture. For that one needs to calculate the peak performance using the vendor provided specifications (clock-rate, number of cores, etc.). Also necessary is the figure for the maximum sustained memory bandwidth, which necessarily goes over the slowest path, namely, from the RAM to the registers. This needs to be measured and the standard tool for that is the STREAM benchmark [6]. Once these steps are done, we can turn to our specific model to determine its arithmetic intensity (FLOP/Byte), which measures how many computations are performed for a given amount of data. Then, using a standard tool to measure the FLOP rate (Likwid, PAPI, Perflib, etc.), one can compare the actual algorithm performance with its theoretical bounds, and from that gain insight about which optimizations could be beneficial.

Since we are interested in running REFMULx on Helios, we decided to make the roofline analysis on that architecture, which uses Intel Sandybridge processors. Running the STREAM benchmark there yielded a peak memory bandwidth of approximately 80 GB/s on 16 cores, which are available on a single (dual-socket) Helios node. Using the triad algorithm of STREAM as an example

```
a[i] = b[i] + const. * c[i]
```

we calculate its actual FLOP rate as follows. With a size of 60 million elements per array, the number of calculated FLOP is

```
(60.0E6 array elements) * (2 FLOP per array index) = 1.2E8 FLOP.
```

If the STREAM benchmark is compiled without optimization (`-O0`) it needs 0.024 s on 16 threads to complete. Dividing the FLOP count by this figure yields a 5.0 GFLOP/s rate. Now we use the Likwid tool [7] to measure the same quantity by accessing the hardware performance counters. The number of FLOP measured with Likwid is close to the expected one, namely,

```
FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE = 1.26433e+08 FLOP
```

yielding a slightly overestimated rate of 5.4 GFLOP/s. However, as soon as optimization (vectorization) is switched on, the FLOP hardware counter yields greatly overestimated values. For instance, using the `-O3` compiler flag decreases the triad elapsed time to 0.019 s, but at the same time leads to a measure of

```
FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE = 1.93772e+08 FLOP,
```

which is almost twice the real FLOP count. Further, since vectorization is used in this case, another factor of two must be used to calculate the FLOP rate metric from this counter. As a result, roughly a factor of four overestimated rate of 25.3 GFLOP/s is measured, instead of the expected 6.3 GFLOP/s for our case.

To make sure that this problem is not specific to the Helios machine, the same experiment was repeated on another Sandybridge machine from RZG, in Garching. Similarly overestimated results were obtained in this case. Conversely, repeating the experiment on another architecture (Nehalem) led to correctly estimated results. All together, this raises doubts about the correctness of the hardware counters in the Sandybridge architecture, and poses the question of how to use those for performance analysis in a consistent manner. This question is still under investigation, and as a consequence the roofline analysis of the REFMULx algorithm was postponed.

9.5. *References*

[1] K. S. Yee, *IEEE Trans. on Antennas and Propagation* **14** (1966) 302
[2] L. Xu and N. Yuan, *IEEE Antennas and Wireless Propagation Letters* **5** (2006) 335
[3] B. Després and M. C. Pinto, private communication (2012)
[4] J-P. Berenger, *J. Comp. Physics* **127** (1996) 363
[5] S. Williams, A. Waterman and D. Patterson, *Comm. ACM* **52** (2009) 65
[6] J.D. McCalpin, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.
[7] J. Treibig, G. Hager and G. Wellein, *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures* (2010)

10. Final report on HLST project TOPOX

10.1. *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 project TOPOX 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) to divertor. The project TOPOX 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 generalize the scheme to make work beyond the separatrix into the SOL, including the X-point. 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.

10.2. *Alternative to standard field-alignment methods*

To understand the reasons why novel techniques for treating geometry and computational meshes are necessary, one must start by noting that the magnetized nature of fusion plasmas imposes a strong space scale separation between the dynamics taking place along (large scales) and across (small scales) the magnetic field lines. The natural coordinates are those aligned with the magnetic field, denominated Clebsch coordinates. These allow lower resolutions in the direction parallel to the magnetic field while keeping finer meshes for the perpendicular dynamics, where resolution is needed. Nevertheless, near the X-point severe coordinate-cell deformation results, and with it grows the demand for spatial resolution. At the X-point, even though the field lines themselves stay well behaved in the vicinity of any point, these coordinate transformations break down as a result of the infinite poloidal connection length. The idea to overcome this is to develop techniques which (1) retain the benefits of field-alignment (moderate spatial and temporal resolution in the parallel direction) and (2) avoid involvement of any consideration of the poloidal field line connection length in the construction of the parallel derivatives. It seems that this can be done via a combination of coordinate mappings and field line tracing as a mathematical replacement to the conventional pure Clebsch coordinate transformations, which are used to construct field-aligned coordinates. Two recent pieces of work [1], [2] lay the material supporting the previous statements. The objective is to construct a projection map of the field lines from one poloidal plane to the next, along a step size expressed as a toroidal angle difference between the planes. So far this has been developed for the region inside the magnetic separatrix but it needs to be extended to the SOL, including the X-point. The first step towards that goal is what concerns us, namely, to extend the numerical scheme of the GKMHD code accordingly.

10.3. *The 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. [3]. A large part of the method's machinery consists of finding the node-index ordering and orientation of the nearest-neighbor hexagons.

The global index 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 the global index for a particular node on flux surface k+1 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 (except the magnetic axis node which is given index 0). That number is given by the formula $3k(k+1)$. For instance, for the sixth flux surface, this number is 3*5(5+1)=90, as shown in Fig. 54 for the six nodes on the sides of the six main triangles (red). As for the local orientation of the six nearest-neighbors of each grid-node, 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 is what determines the column indexes of the Laplacian (sparce) matrix elements, which are solved using canned libraries, like IBM WSMP or PETSc. Of course multi-grid methods can also be used for the same purpose, but this is a subject that belongs to the project MGTRIOP of K.S. Kang.

In terms of extending the method to go across a magnetic separatrix, 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. Then, an artificial boundary is set between the X-point and a grid node in the outermost flux surface, as depicted in cyan on Fig. 54. Everything outside this domain is discarded. This means that the divertor itself is not included, and will have to be modeled 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, the private flux region and the X-point while maintaining the hexagonal topology of the closed field line grid. This means that the numbering and ordered method described before can be directly applied to the domain on the open field-line region (in blue). An example of nearest neighbor orientation is also given in Fig. 54 for the SOL region (magenta). The exception to this rule occurs in the grid-nodes sitting in the left (cyan) artificial boundary. The following nearest neighbor on the same flux surface lies outside the computational domain, so a different rule must be applied there. For instance, to use instead the backward point in the same flux surface. Which rule makes up the best choice is something that needs to be investigated later.

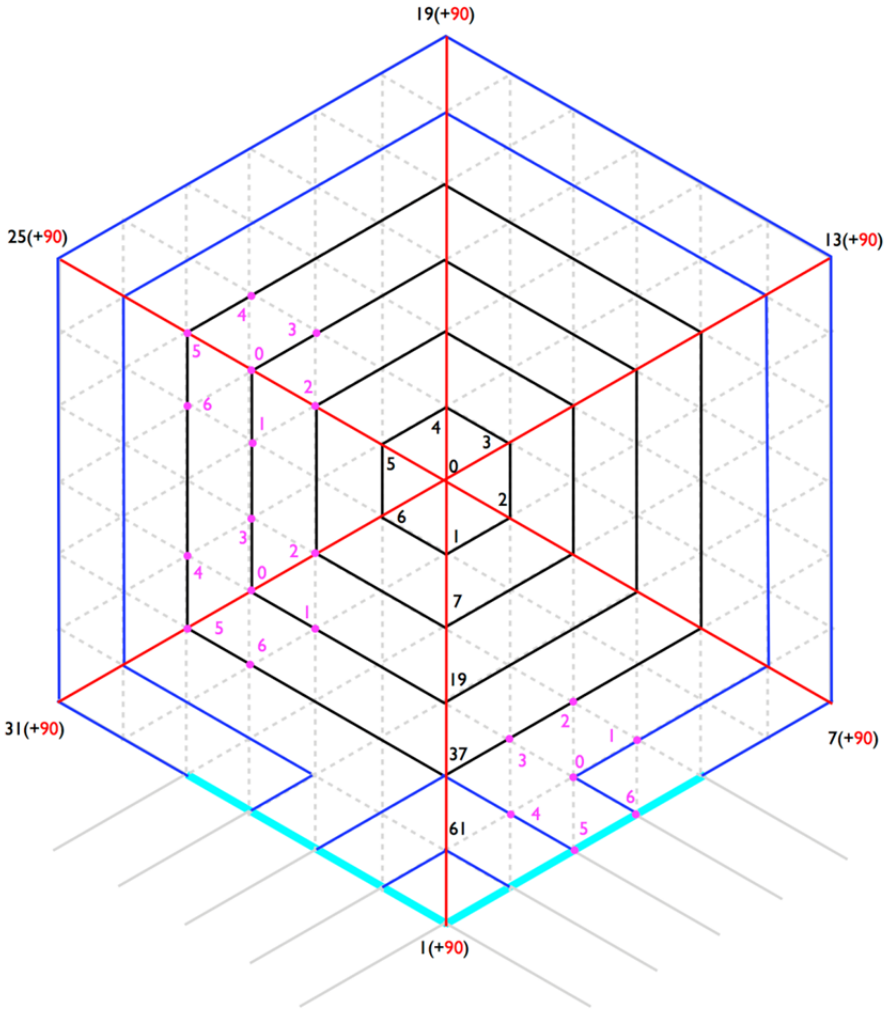


Fig. 54 Illustration of the hexagonal grid currently used in GKMHD (black flux surfaces) and its proposed extension to the SOL region (blue flux surfaces). The global numbering is hinted with a few examples given (black and red numbers). The local number ordering and orientation of the nearest-neighbor hexagons is also illustrated with a few examples (magenta numbers).

10.4. *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 them before they are 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.

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

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

with a source term given by [4]

$$f(r,\theta) = -4(p+1)r^p \cos(p\theta) \tag{2}$$

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

$$v(r,\theta) = \left(1 - r^2\right) r^p \cos(p\theta) \tag{3}$$

Setting the free parameter to *p*=10 yields the functions depicted in Fig. 55.

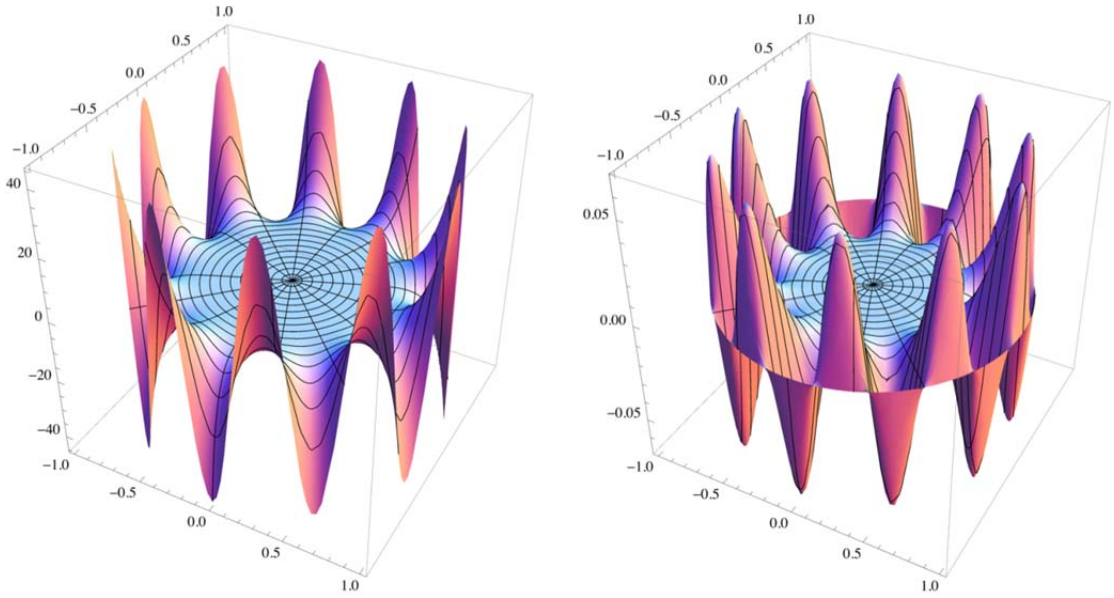


Fig. 55 Source term (2) and analytical solution (3) of the Poisson equation (1) for *p*=10.

The standard approach to address this problem would be to build a bi-dimensional solver in polar coordinates. This was done using a pseudo-spectral method based on the work by Lai [5]. 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 [5]. It shall be called Lai's method henceforth.

The same test-case (1)–(3) was implemented using the Sardourny's line integral method on hexagonal grids [3] of the GKMHD code. The choice made for the local orientation of the six nearest-neighbors of each grid-node explained in Sec. 10.3 yields a Laplacian (sparce) matrix equation, which is solved using the WSMP library. This shall be referred to as the Sadourny's method in the remainder text.

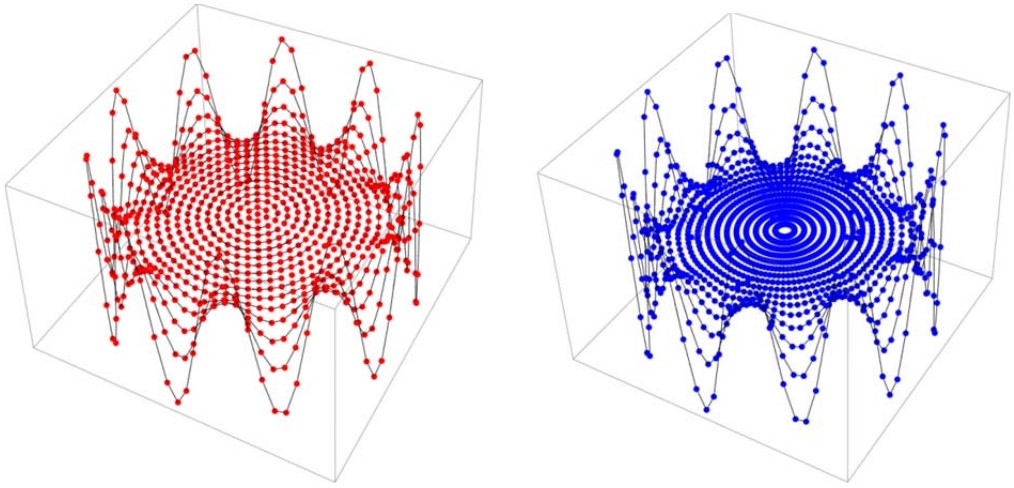


Fig. 56 Representation of the source term (2) for p=10 on Sadourny's hexagonal grid (left, in red) and Lai's polar grid (right, in blue). Both grid-counts yield the same number of grid nodes on the outer radial contour, namely 96, but it is clear that the polar total grid-count (1536) is much higher than the hexagonal counterpart (817), leading to unnecessary resolution on the more internal radial contours.

The plots in Fig. 56 show the numerical discretization of the (continuous) source term (2) of Fig. 55, for each of the two solvers methods, using the same poloidal resolution of 96 grid-nodes on the outer-most contour. Here it is noteworthy the adequacy of the chosen source term to the problem under study. 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 center 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. 56. The same does not happen for the Sadourny's hexagonal grid illustrated in 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.

Since the circular test-case possesses a non-trivial analytical solution, its comparison to the numerical solution provides an important 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 inspect the precision of the methods. The plots in Fig. 57 show the numerical error, defined 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 similar to 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 to. This confirms the higher efficiency of the hexagonal grid compared to the polar one.

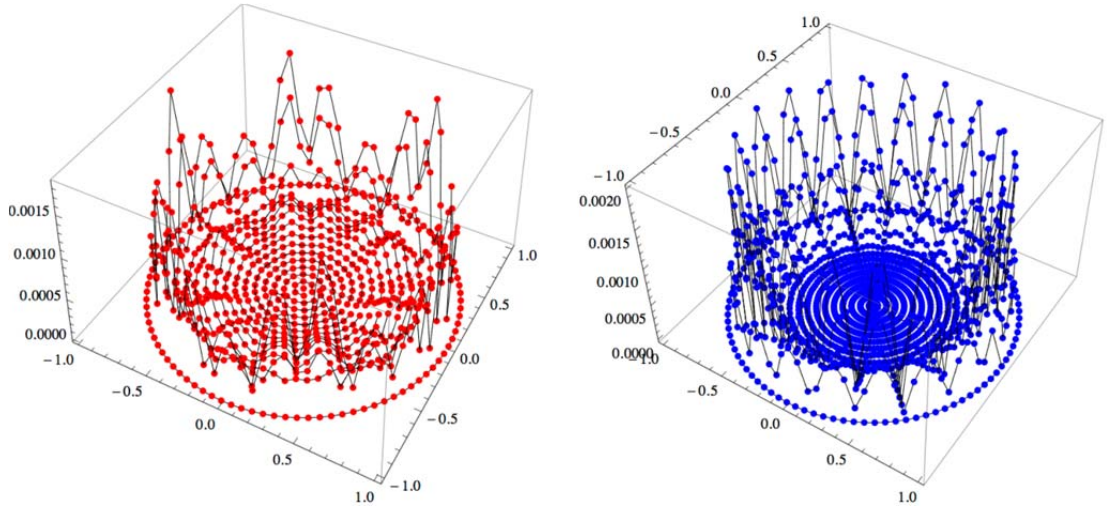


Fig. 57 Plots absolute difference (error) to the analytical solution (3) for Sadourny's method (left) and Lai's method (right), with *p*=10 on the same grid counts as before, namely, 1536 and 817, respectively. Despite this difference, both methods yield similar precisions, namely, maximum errors of 1.86e-3 (Sadourny) and 2.01e-3 (Lai) were obtained.

Finally, one should note that Lai's polar pseudo-spectral method is limited to circular domains, at least in its simple form introduced before. The same does not apply to Sardourny's line integral method on hexagonal grids [3], which can be applied to more general radial contour shapes. Fig. 58 illustrates this fact by solving (1) on the contours given by

$$\begin{aligned} R(r,\theta) &= r\cos(\theta + \delta\sin(\theta)) \\ Z(r,\theta) &= r\sin(\theta) \end{aligned} \tag{4}$$

with the triangularity set to δ=1/2, where *R* and *Z* are the Cartesian coordinates.

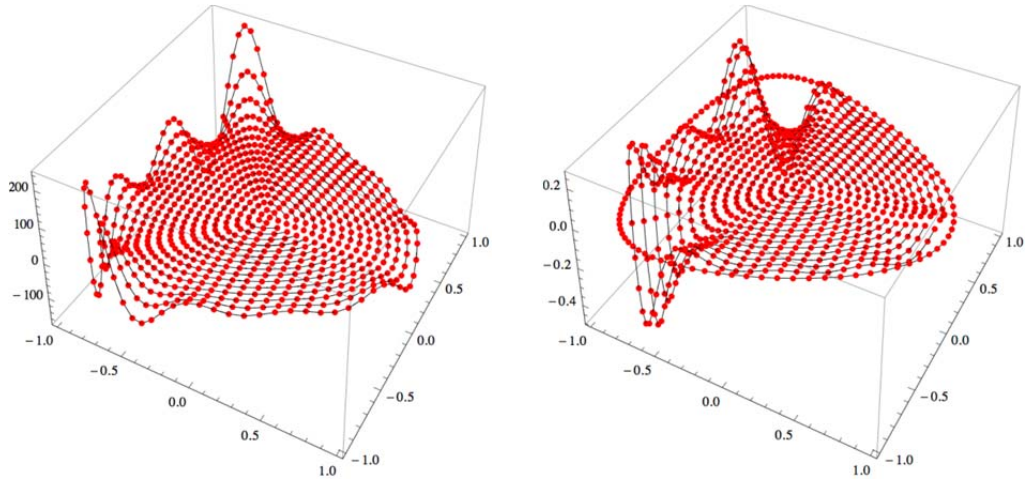


Fig. 58 The numerical solution of the Poisson equation (1) calculated using Sadourny's method on radial contours yield by (4), with the triangularity set to δ=1/2 is shown on the right. The corresponding discretized source term (2) is shown on the left.

10.5. *Conclusions and outlook*

Most of the work done so far consisted in getting acquainted with the tools already available, namely the GKMHD code and its underlying solver algorithm, and constructing an appropriate stand-alone test-case. They are necessary steps for the development of the new method and, at the same time, provide basic benchmarking tests with known analytical solutions on simplified problems. So far the test-cases

include a finite-differences pseudo-spectral method [5] and the implementation of Sadourny's method [3] using the WSMP library. The comparison of both methods demonstrated both the correctness of implementation of the latter, as well as its advantages over the former in terms of discretization efficiency. The next steps need to address the applicability of the proposed extension of the grid-numbering across the separatrix. A few ideas on how to generalize the existing sparse matrix routines of GKMHD are already available, but there might be still open issues related to the need to have the appropriate equilibrium magnetic structure (SOL, private flux and closed field-line regions with a X-point separatrix as the boundary surface between them) within GKMHD. Even though GKMHD is not a free-boundary MHD solver, the project's principal investigator Bruce Scott thinks that prescribing the appropriate boundary flux shape shall yield the needed X-point magnetic field structure. Given that the project's end was reached, the implementation of such ideas will not be attempted for now.

The relatively short duration of the project (six months) for the ambitious tasks proposed resulted, so far, mostly in checks of the existing method, rather than the implementation of new extensions. Nevertheless, the investment made led to the conceptual solution of how to do the desired extension, namely, to have GKMHD work across the magnetic separatrix, which is the main goal of the project. The corresponding implementation and subsequent testing shall be left for the future, possibly within a project extension, if the principal investigator so decides. This was actually foreseen in the original TOPOX proposal, which stated a possible extension of the project by six months.

10.6. ***References***

[1] T.T. Ribeiro and B. Scott, *IEEE Trans. Plasma. Sci.* **38** (2010) 2159
[2] M. Ottaviani, *Phys Letters A* **375** (2011) 1677
[3] R. Sadourny, A. Arakawa and Y. Mintz, *Mon. Weather Rev.* **96** (1968) 351
[4] Trach-Minh Tran, private communication (2013)
[5] M.C. Lai, *Numer. Methods. Partial Differential Equations* **18** (2002) 56