EUROfusion

EUROFUSION WPISA-REP(16) 16106

R Hatzky et al.

HLST Core Team Report 2012

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 2012

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	running
EMPHORB	Tamás Fehér	running
GYNVIZ	Matthieu Haefele	finished
ITM-ADIOS	Michele Martone	finished
KSOL2D-2	Kab Seok Kang	running
MGTRI	Kab Seok Kang	finished
NEMOFFT	Tiago Ribeiro	finished
PARFS	Michele Martone	running
TOPOX	Tiago Ribeiro	running

Table 1 Projects mapped to the HLST core team members.

Roman Hatzky has been involved in the support of the European users on the new 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 project.

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 is to implement the phase factor transformation in the NEMORB code. Phase factor transformations can be used to decrease the spatial resolution for linear calculations and thereby lower the amount of computational resources needed for the simulation.
The implementation of the phase factors involves small changes that extend throughout the whole NEMORB code. The code structure was analyzed and the phase factors were implemented for the main steps in the particle-in-cell (PIC) algorithm. This includes the matrix building routines, the field evaluation, the charge and current assignment. The equation of motion changes only for the weights, and the phase factors have been introduced for pushing the weights.
When the phase factors are introduced, several variables need 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 most of the necessary changes have been made. In the next step, the diagnostic methods will be updated accordingly. The diagnostic module uses the FUTILS library developed by Trach-Minh Tran. To gain experience with this library, work has been done to improve the FUTILS interface in the EUTERPE code. Once all the changes are done, an extensive testing will be necessary. There are several different switches in NEMORB to include additional physical effects. Test cases will be needed to check the modified code with these options.

Several codes from the BEUPACK benchmark suite (NEMORB, JOREK, GYSELA, and some libraries used in these codes) were analyzed with Forcheck. During the analysis, smaller bugs and some Fortran standard conformance issues were reported by Forcheck. Some of the reported errors were not real ones, but mistakes by Forcheck. Such mistakes were reported to the developer of Forcheck, who quickly corrected them. The remaining errors were reported to the code developers together with instruction on how to use the improved Forcheck tool.

Matthieu Haefele worked on the GYNVIZ and BLIGHTHO projects.

The aim of the GYNVIZ project was to unify and to provide support for the whole hardware and software chain concerned with the visualization process of large datasets being produced by the major European turbulence codes. Unfortunately, the original scope of the project had to be reduced because of the failure of the collaboration with the XDMF development team. As a result, the usage of existing XDMF technology in the framework of GYNVIZ became impossible. So, the initial intention of bringing a common and standard format to a range of different codes had to be given up. Nevertheless, the main focus has always been to bring the 4D visualization functionality to a mature production state which has been achieved. A data size reduction by several factors up to an order of magnitude can be expected as a result of data compression. The interactive visualization of the resulting 4D dataset is available on the remote visualization cluster at Rechenzentrum Garching (RZG). GYNVIZ users have now started to use these new tools and the first feedback is positive. The maintenance of the software package will go on with low priority although the GYNVIZ project has officially reached its end.

The BLIGHTHO project is explicitly providing support for the European scientists who use the HELIOS machine at IFERC-CSC. At the beginning, the project went into operation by supporting selected European projects during the lighthouse phase. After the HELIOS machine went into production phase, starting in April 2012, the support activities have been extended to all approved European projects.
The BLIGHTHO project gives support on different levels. The 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, testing the file transfer between Europe and CSC in Japan and assessing the hardware capabilities of HELIOS.
The HLST as a whole has submitted more than 90 tickets to the CSC tracker system. This gives a rough quantitative estimate of the contribution of the HLST at pointing out issues and helping to solve them. In the framework of this project, we also supported RZG at setting up the software environment for efficient file transfer between Europe and Japan. But the main contribution is 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. This requires approximately 9h on 16K cores for the Bull MPI implementation. Some workarounds are proposed so that such ALL_TO_ALL operation can be performed now on the full system with 64k cores. We also assessed this initialization time on different architectures. Our first results show that this phenomenon does exist on IBM and Cray machines as well, but is by far less severe than on the HELIOS machine.

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

The goal of the project was to obtain a new parallel code by combining two existing codes: GEMZ and GKMHD. GEMZ is a parallel gyrofluid code based on a conformal, logically Cartesian grid. GKMHD is a serial MHD equilibrium solver which evolves the Grad-Shafranov MHD equilibrium. Therefore the GKMHD code had to be made parallel which was achieved by implementing a parallel multigrid solver and three domain decomposition methods on a triangular grid.
In a first step the data structure had to be adapted to get an efficient parallel performance. Then we had to determine how to distribute the triangular grid information on each core and how to implement the parallel matrix vector multiplication, the smoothing operators and the intergrid transfer operators. These are typical ingredients to implement an iterative solver, especially a multigrid solver.

After having ported the code to the new HELIOS machine, it is possible now to run our test case on up to 24,576 cores. Scaling tests show that the multigrid solver has a very good semi-weak scaling property up to 24,576 cores for the larger test cases. Furthermore, we implemented three different domain decomposition methods: the two-level Schwarz method, the FETI-DP method, and BDDC method. The two-level Schwarz method proved to be inefficient for our test problem. In contrast, the FETI-DP and BDDC methods were viable. In all test cases the FETI-DP method has been found to be better performing than the BDDC method. The comparison between the FETI-DP method and the multigrid method showed a more complex picture. It is only for the smallest number of Degrees of Freedom (DoF) per core (32 DoF) that both of the methods show similar performance. For all other cases with larger numbers of DoF per core our implementation of the multigrid method with gathering of the data has been the fastest solver for our problem. For small numbers of DoF per core we expected that the FETI-DP method would perform better than the multigrid method. However, for the problem under consideration this is not the case. This might be different for a hybrid parallelization concept of OpenMP and MPI.

The KSOLD-2 project aims at the 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. The FETI-DP and BDDC methods are two candidates for efficient non-overlapping domain decomposition methods for 2D and 3D problems. Even if these two methods should be slower than the multigrid method, such an approach for SOL geometry might give us valuable experience.
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. As a first step we started to adapt the data structure of BIT2 to make the gathering of the data possible. In parallel we have been studying how to implement the two-level non-overlapping domain decomposition methods. Hence, it is necessary to adapt and implement the FETI-DP and the BDDC methods for the SOL geometry. Finally, performance tests on the HELIOS machine at IFERC-CSC will show whether the multigrid method will outperform the FETI-DP and the BDDC methods for this case.

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

The ITM-ADIOS project's initial goal was to evaluate the usage of the "ADIOS" (ADaptive I/O System) parallel I/O library (a GPL licensed software) on the HPC-FF machine. It was subsequently decided to prolong the project in the perspective of extending the investigations to the HELIOS machine when it came into operation.
Motivation for the project came from the growing gap between computing and I/O capability of modern hardware. It leads to I/O being increasingly a bottleneck in Integrated Tokamak Modelling (ITM) simulations, when checkpoint functionality is used. To explore the capabilities of ADIOS, a special purpose parallel I/O throughput benchmark ("IAB", short for "ITM-ADIOS Benchmark") has been developed. In addition, an assessment of the maximum throughput achievable when writing from a serial program was performed. Using POSIX C I/O routines from a serial program, but tuning appropriately the (user side) LUSTRE file system "striping" parameters on HPC-FF, it was possible to obtain around 1 GB/s throughput in writing. For the general user, it is common to operate at a fraction (e.g.: a quarter) of that, because of an inappropriate choice of I/O options and techniques.

In contrast, calling ADIOS from IAB and writing array data originating from 4096 MPI tasks to a subset of 64 files, with a good choice of striping parameters, enabled as much as 14 GB/s to be obtained on HPC-FF. Easy to implement I/O choices, like using either one or all tasks, have been found to be not as successful as the aforementioned ones. The first choice suffers from node network bandwidth limitation, while the second incurs excessive network usage and I/O latency. The "optimum" observed with ADIOS on HPC-FF delivered about 75% of the estimated machine hardware peak, this is approximately 20 times faster than a good serial I/O choice. These results are particularly interesting given the wide-spread practice of users operating their codes at the above cited extremes, therefore either under- or over-utilizing the machine, in both cases obtaining poor results. In contrast, we have been able to identify a set of parameters for optimal usage of ADIOS.
Our massive I/O experiments on HELIOS exposed several I/O system instabilities. We have been active in investigating these instabilities and communicating them to the CSC support team in Japan. Finally, most of the instabilities were solved so that it became possible to finish our benchmarking campaign with the following results: the maximal observed I/O throughput of HELIOS was circa 35 GB/s (roughly twice as on HPC-FF) and a significant gap of parallel to serial I/O, which can reach 70 times, was observed. This increased gap is due to both faster parallel and slower serial I/O throughput. The slower serial I/O has been identified as originating from an inefficiency in the LUSTRE version mounted on HELIOS; this problem is expected to be solved within future releases of LUSTRE.
To summarize, the major outcome of our investigation has been the characterization of the relations between LUSTRE striping parameters and general I/O quantities. By that we have been able to extrapolate rules of thumb for singling out poor performance cases and avoiding them in advance.

The project PARFS (PARallel Field Solver) began in November, in collaboration with the ENEA Frascati Theory Group (Italy). The main purpose is adapting the HYMAGYC code in order to support resolutions of ITER-like simulations. A distributed memory parallel version of the current field solver component (MARST) is necessary. The HYMAGYC code is 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 which features both Particle-In-Cell (PIC) and Magnetohydrodynamic (MHD) components. While the PIC component can currently scale up to several hundreds of processes, the field solver's computational core (the linear solver) is still serial, using its own ad hoc implementation. Hence, the solver has to be replaced by an equivalent parallel solver with low memory consumption and good scalability property.
Activities performed so far include minor changes to HYMAGYC for portability and Fortran standards compliance, estimates regarding the minimal memory requirements of the underlying linear system, and first investigations aimed to identify suitable linear solver packages. Once a suitable solver is identified, its robustness, scalability and resource usage characteristics will be investigated. Finally, it will be integrated into MAST.

Tiago Ribeiro mainly worked on the NEMOFFT project and just started the TOPOX project.

The NEMOFFT project aimed at removing, or at least alleviating, a well-known parallel scalability bottleneck of the global gyrokinetic particle-in-cell (PIC) code ORB5, in its current electromagnetic version NEMORB. This code has high demands on HPC resources, especially for large-sized simulations. At each iteration, the particle charges are deposited on a spatial three-dimensional (3D) grid, which represents the source term in a Poisson equation (the right-hand side). Due to strong spatial anisotropy in a tokamak, only a restricted set of degrees of freedom, or modes, are allowed to exist, namely, the ones which are either aligned or close to

being aligned with the background guiding magnetic field. Using this physical restriction on the Poisson solver not only reduces the required floating point operations (solve only for the allowed modes) but also improves the numerical signal to noise ratio, which is of central importance in a PIC code. This implies calculating two-dimensional (2D) Fourier transforms to apply the corresponding filters, but because the mesh domain of NEMORB is distributed across several cores, this requires large amounts of grid data to be transposed across cores. Such inter-core communication naturally impairs the code's parallel scalability, and in practice renders ITER-sized plasma simulations unfeasible.
The optimization involved the implementation of the Hermitian redundancy on the NEMORB's Fourier transforms. This enabled the reduction of the number of floating point operations involved by a factor of two. Further using a clever storage technique for the Hermitian-reduced data (half-complex-packed-format) also reduced the data-set to be transposed across cores by the same amount, which led to the expected speedup factor of the order of two on HELIOS for nominal grid-counts representative of a typical ITER simulation. On HPC-FF, the degradation observed in this factor (from 2 to 1.5) was found to be related to network congestion/latency limitations. This motivated the implementation of alternative transpose methods. Nevertheless, the performance measurements made on those, which included using the MPI_Alltoall directive (with and without derived data types) and its FFTW3.3 and MKL BLACS counterparts, as well as an alternative algorithm based on the MPI_Gather/Scatter directives, revealed that the original "hand-coded" XOR/MPI_Sendrecv algorithm gave, in general, the best results.
Another reason taken into account for recommending the XOR/MPI_Sendrecv as the default method is related to its flexibility. It allowed to exploit NEMORB's low pass filtering, used to ensure enough B-spline resolution on all physically allowed modes in the system, to further reduce the data-set to be exchanged across cores. Indeed, *a priori* knowledge of the matrix elements that are set to zero by the filter enabled parts of the matrix to be excluded from the data exchange step, such that only a partial transposition was performed. This further increased the gain obtained leading to the final figures for the speedup factors achieved, namely, about 2.2 and 1.7, on HELIOS and HPC-FF, respectively.
The last point refers to a side-study motivated by the excessive cost of all-to-all communication initialization on HELIOS with Bullx MPI. The conclusions obtained therein were taken into account when interpreting the various scaling measurements made throughout the work. The relevance of this issue led to further investigations made by Matthieu Haefele in the framework of the HLST project BLIGHTHO, in close contact with Bull.

1.2. Further tasks and activities of the core team

1.2.1. Dissemination

Hatzky, R.: The High Level Support Team, *IPP Theory Workshop*, 5nd – 9th November 2012, Ringberg, Germany.

1.2.2. Training

Hatzky, R.: The numerics behind global electromagnetic gyrokinetic particle-in-cell simulation for tokamaks and stellarators, *IPP-Kolloquium*, 17th February 2012, Garching, Germany.

Hatzky, R.: Global electromagnetic gyrokinetic particle-in-cell simulation, *4th Summer School on Numerical Modelling for Fusion*, 8th – 12th October 2012, IPP, Garching, Germany.

1.2.3. Internal training

The HLST core team has attended:

- The HLST meetings at IPP, 16th March and 23rd October 2012, Garching, Germany.
- Advanced Topics in High Performance Computing at LRZ, 19th – 22nd March 2012, Garching, Germany.
- Training for IFERC Super Computer (HELIOS) users, 23th – 26th April 2012, Garching, Germany.
- 4th Summer School on Numerical Modelling for Fusion, 8th – 12th October 2012, IPP, Garching, Germany.
- Node-Level Performance Engineering, 6th – 7th December 2012, LRZ, Garching, Germany.

Michele Martone has attended:

- Parallel I/O and Portable Data Formats at Jülich Supercomputing Centre (JSC), 28th – 30th March 2012, Jülich, Germany.

Tamás Fehér and Michele Martone have attended:

- Iterative Solver and Parallelization, 10th – 14th September 2012, Leibniz-Rechenzentrum (LRZ), Garching, Germany.
- 10th VI-HPS Tuning Workshop, 16th – 19th October 2012, LRZ, Garching, Germany.

1.2.4. Workshops & conferences

Haefele, M.: How to make an efficient all-to-all communication on a petaflop system, *IPP Theory Workshop*, 5nd – 9th November 2012, Ringberg, Germany.

Kang, K.S.: A parallel multigrid solver on a structured triangulation of a hexagonal domain, *21st International Conference on Domain Decomposition Methods*, 25th – 29th June 2012, INRIA Rennes-Bretagne-Atlantique, France.

Kang, K.S.: Parallel solvers: multigrid method and domain decomposition methods, *Computational Science – Future of HPC and programming, EU-Korea Conference on Science and Technology (EKC 2012)*, 26th – 28th July 2012, Berlin, Germany.

Kang, K.S.: A fast parallel solver on a structured triangulation of a hexagonal domain, *International Conference on Computational Methods in Applied Mathematics CMAM-5*, 30th July – 3rd August 2012, Berlin, Germany.

Kang, K.S.: *GAMM-Fachausschuss for Computational Science and Engineering (CSE), Kickoff meeting*, 17th – 18th September 2012, Leibniz-Rechenzentrum (LRZ), Garching, Germany.

Kang, K.S.: The parallel multigrid and domain decomposition methods, *IPP Theory Workshop*, 5nd – 9th November 2012, Ringberg, Germany.

Martone, M.: An efficient sparse matrix storage scheme for shared memory parallel Sparse BLAS operations, *7th International Workshop on Parallel Matrix Algorithms and Applications (PMAA)*, 28th – 30th June 2012, Birkbeck University of London, UK.

Ribeiro, T.: Improving the scalability NEMORB's Fourier filtering, *IPP Theory Workshop*, 5nd – 9th November 2012, Ringberg, Germany.

1.2.5. Meetings

Roman Hatzky attended on a regular basis:

- CSC management meeting
- CSC European ticket meeting

1.3. Recommendations for the year 2013

The call for the use of High Level Support Team resources will close on 31st January, 2013. The core team leader will present a recommendation on how to distribute the work load over the HLST members. The corresponding spreadsheet will be passed to the HLST coordinator. The final decision will be made by the HPC board.

2. 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. As we can see from recent results from the GYGLES code, with such a type of code it is possible to study global Alfvén waves and their interaction with energetic particles. NEMORB has some advantages over GYGLES, it has collisions implemented between the different particle species, and it can perform nonlinear simulations. Furthermore, it could be used to refine the study of wave-particle interaction. The first step in this direction is the linear study of Alfvénic modes.

The aim of the EMPHORB project is to implement the phase factor transformation in the NEMORB code. Such transformation can only be used for linear simulations, but it effectively decreases the amount of computational resources needed for the simulation.

The idea is the following: let $\Phi(s,\theta,\phi)$ be an oscillating quantity, a function of the spatial coordinates (s,θ,ϕ). To discretize this function we assume the following ansatz:

$$\Phi(s,\theta,\phi) = \tilde{\Phi}(s,\theta,\phi)S(\theta,\phi), \quad S(\theta,\phi) = \exp\left[2\pi i\left(m\frac{\theta}{\theta_{max}} - n\frac{\phi}{\phi_{max}}\right)\right].$$

Here *S* is the phase factor function and $\tilde{\Phi}$ is the phase factor extracted quantity. 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 for the original function. In the NEMORB code, we extract the phase-factors from the electric (Φ) and magnetic potentials $(A_{\|})$, and from the distribution function *f*.

2.2. First steps

After receiving the code and a test case from the principal investigator, Alberto Bottino, the Forcheck tool was used to check NEMORB for inconsistencies with the Fortran standard. This was a useful exercise for understanding the code and gaining more experience with the Fortran standard. In a section 2.8, this activity is described in more detail.

The actual work on the EMPHORB project has started by analyzing the structure of the NEMORB code, in order to identify parts where modifications are necessary. The basic procedures in NEMORB are 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.

In all these four steps we have to consider the phase factor. Apart from this, the diagnostic and post processing functionality has to be modified.

Fig. 1 shows the functions that are called during the PIC loop in NEMORB. Red circles highlight subroutines that implement the 1st, 2nd and 4th steps of the PIC algorithm. The work to include phase factors has started in these functions. In the next sections, these modifications are described in more detail.

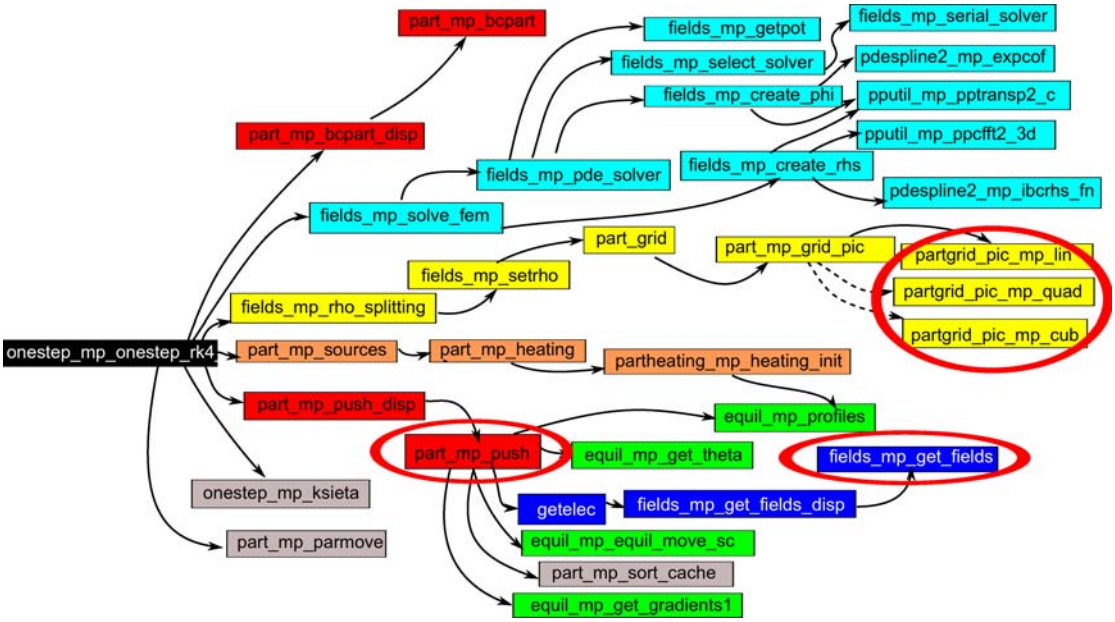


Fig. 1 Subroutines called in the PIC loop: the red circles highlight subroutines where most of the modifications were necessary.

2.3. Field evaluation

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

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

The field is represented in the code by B-splines whose polynomial order can be linear, quadratic or cubic. The spline values are interpolated from a three dimensional grid. The spline interpolation is explicitly coded for all three orders of B-splines, the corrections had to be implemented for all orders separately. To do that, the potential was evaluated, and multiplied by the gradient of S. The gradients of the coordinates ($\nabla\theta$ and $\nabla\Phi$) are provided by magnetic equilibrium functions. The modifications have been implemented in the GET_FIELDS function.

2.4. Charge assignment

Charge assignment is an important step in the PIC algorithm, as it 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

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

Here ρ is the charge density and it is discretized with splines on the three dimensional grid used in the code. Because of the spline discretization, the following integral is calculated:

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

Here Λ_k is an elementary spline function, and star denotes its complex conjugate. As the distribution function is discretized with markers, the integral corresponds to the following sum:

$$\rho_k = \sum_\nu \sum_j \sum_a q_\nu w_j \Lambda_k^*(r_j + \rho_j(\alpha_a)).$$

The integral over the angular coordinate in velocity space corresponds to a gyroaverage, and it is approximated with an *N* point average over the phase angle α. The phase factor is introduced in the weight $w_j = \tilde{w}_j S(r_j)$, and in the spline functions $\Lambda_x^* = \tilde{\Lambda}_k S^*$. Since $S^* = S^{-1}$, the phase factor almost entirely drops out, there is only a small contribution remaining because of the gyroaverage. So the integral becomes

$$\rho_k = \sum_\nu \sum_j \sum_a q_\nu \tilde{w}_j \tilde{\Lambda}_k(r_j + \rho_j(\alpha_a)) S(-\rho(\alpha_a)).$$

This summation is evaluated in the PART_GRID subroutine, separately for linear, quadratic and cubic splines. The subroutine was modified to include the phase factor contribution.

2.5. Weight evolution

The original weight evolution equation is

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

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

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

This modification has been implemented in the PUSH subroutine that calculates the time derivatives for the particle motion. The introduction of phase factors does not affect the usability of the fourth order Runge-Kutta time integrator. However, it might be possible to use a simpler second order scheme, because in that case the phase factor part can be analytically integrated using an operator splitting method. This possibility will be considered later. 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.

2.6. Matrix building

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. Apart from the matrices for the Poisson and Ampère equation, another matrix is also built to store zonal flow contribution. 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 quasi-neutrality and (in the simplest case) assume adiabatic electrons, so we can write the Poisson equation into the following form

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

After discretization with splines, the differential equation is transformed to an algebraic equation

$$A_{jk}\Phi_k = \int \mathrm{d}^3x \left(\frac{en_0}{T}\Lambda_j^*\Lambda_k + \nabla_\perp \Lambda_j^* \cdot \nabla_\perp \Lambda_k\right)\Phi_k = \rho_k,$$

where matrix A_{jk} represents the discretized equation, Φ_k represents the unknown coefficients of the field, and ρ_k is introduced in the charge assignment section. When we introduce the phase factors, then the basic spline functions are replaced by the phase factor extracted spline functions $\Lambda_k = \tilde{\Lambda}_k S$, which is a complex function. The complex conjugate Λ_j^* function has been introduced in the integral to ensure that the resulting matrix is Hermitian. The expression for A_{jk} contains the derivative of the splines; for these, we have to include the phase factors. The field derivative in perpendicular direction is approximated with the poloidal derivative

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

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

$$A_{jk} = \int \mathrm{d}^3x \left(\frac{en_0}{T}\Lambda_j^*\Lambda_k + \left[\nabla\tilde{\Lambda}_j - i\tilde{\Lambda}_j m\nabla\theta\right] \cdot \left[\nabla\tilde{\Lambda}_k + i\tilde{\Lambda}_k m\nabla\theta\right]\right).$$

The weak form of the equation contains only first order derivatives, so it will be sufficient to correct the first derivatives of the splines with the phase factor.

The code implements different methods to build the matrix and 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

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

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 in (McMillan *et al, 2010).* The phase factors contributions have been implemented for this solver.

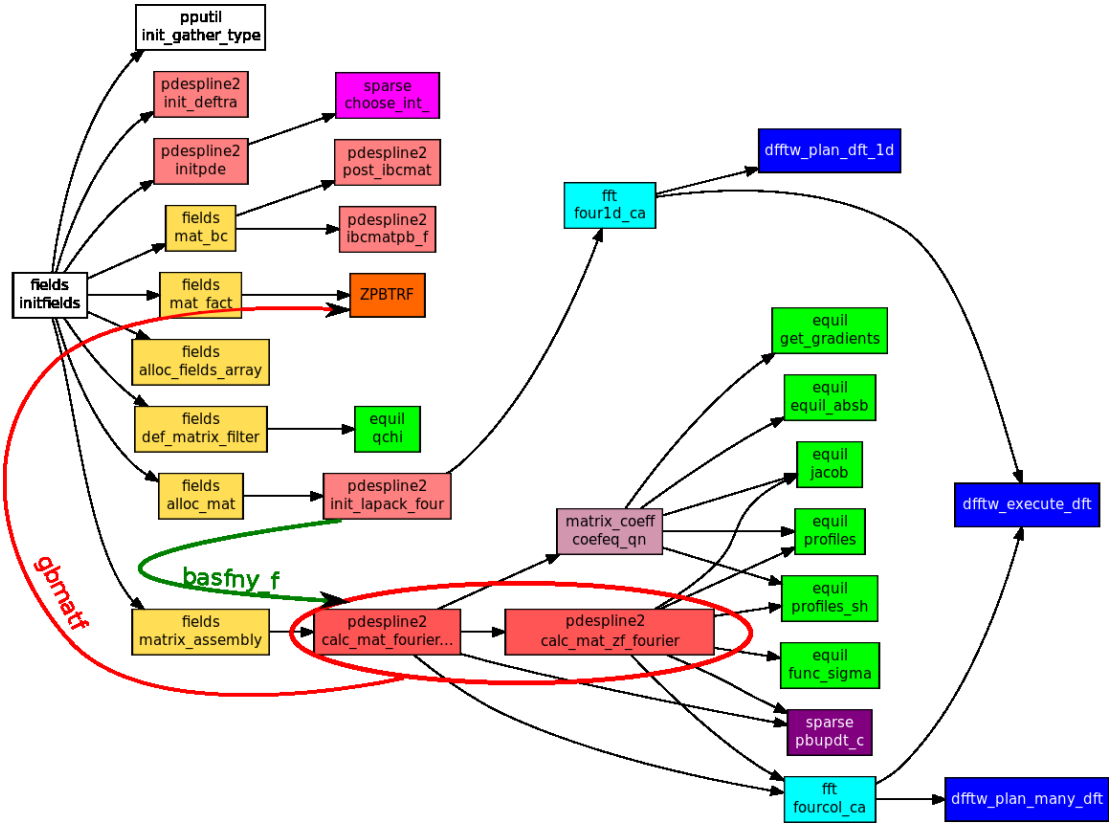


Fig. 2 Subroutines called during field initialization. The subroutines where the matrix elements are calculated are highlighted with the red circle. The matrix building uses precalculated spline values that are stored in the BASFNY_F array. The resulting matrix GBMATF is passed to the linear solver routines.

Fig. 2 shows the functions called during the initialization of the field solver. The matrix elements are calculated in the subroutines CALC_MAT_FOURIER and CALC_MAT_ZF_FOURIER. These functions use 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 Cholesky decomposition of the matrix is calculated during the initialization phase. During the simulation, the solution for the field is determined by back substitution. In the original code, the solution us reduced to its real part. For the phase factor the whole information is kept instead.

2.7. Complex variables

The phase factor S is a complex quantity, therefore, the equations will become complex, and it will be necessary to use complex variables in the code. As we introduce the phase factors, the weight and the field 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. Modifications were necessary for several function arguments, global variables in the field module and temporary variables within the subroutines.

A preprocessor switch was introduced to turn the phase factor mode on/off at compile 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 too much the performance for normal (non-phase factor extracted) operation of the code.

Most of the necessary real-complex changes have been done. There is still some work required for the diagnostics routines.

2.8. Forcheck analysis of the BEUPACK benchmark suite

Forcheck is a comprehensive code verifier that performs static analysis for Fortran programs. It can be used by code developers to identify bugs in their code and to ensure that the code conforms to the Fortran standard. Forcheck can detect more anomalies than most compilers, therefore it is a valuable tool during code development. The aim of this work is to run Forcheck on some of the Fortran codes of the BEUPACK fusion benchmark suite.

The analysis performed by Forcheck is very complex. It has to understand different versions of the Fortran standard, it can handle compiler specific extensions and preprocessing options. Due to the evolutionary nature of the development of scientific codes, sometimes they contain complex programming constructs and provide a great challenge for a code analyzer. Therefore, it is not surprising that in some cases Forcheck reports false error messages. Also, the interface for the external libraries has to be properly set up for the Forcheck analysis, otherwise this can trigger error messages that hide real problems. We support the code developers by establishing an initial Forcheck analysis of their codes. This way they have a well defined starting point, from which they can analyze their code with Forcheck and can focus on the real problems, which can occur during further development.

The Forcheck interface information of external libraries was created. One of the frequently used libraries was HDF5. Forcheck has found a few standard conformance issues in the Fortran interface of HDF5. These problems were reported to the developers of HDF5. A makefile was created to generate the Forcheck library file for HDF5.

Another commonly used library is MPI. Forcheck provides a library file that describes the MPI interface. However, the codes have to USE the MPI module, instead of including the mpif.h file, otherwise Forcheck cannot check the argument types for MPI calls. We have proposed some modifications in the codes, which allow the user to switch between the two methods of interfacing with MPI.

The error messages reported by Forcheck were investigated. False errors due to bugs in Forcheck were reported to its lead developer, Erik Kruyt, who has promptly corrected most of these problems.

The remaining errors were reported to the code developers. The simple problems (such as issues with Fortran standard conformance, or typos) were corrected, and the corrections were submitted in a form of patches. The open problems were explained to the developers so they could fix them. Detailed instructions have been given on how to use Forcheck for their code.

2.8.1. NEMORB

The NEMORB code uses the Futils library, which provides an interface to HDF5, and it is specifically tailored for turbulence simulations. Forcheck was used to analyze Futils and create to the Forcheck library file with interface information. After identfying

a few Forcheck bugs, only one minor problem was found in Futils: a non standard type kind specification. A bug report was sent to the developer, Trach-Minh Tran.

Interface information for the LAPACK library was also created, before checking NEMORB. After eliminating the Forcheck bugs, there were some simpler and some more difficult problems remaining in NEMORB. The simpler problems included some typos and copy-paste errors, and some problems with conditional variable initialization. The other problems were with uninitialized variables, most of these in parts of the code which are currently under active development.

2.8.2. JOREK

The code had some preprocessor macros that Forcheck did not understand correctly. These problems were fixed by the developers of Forcheck. The source of JOREK spans over almost two hundred files and it took time to fix the MPI interface related issues. The complex structure of this code has triggered several Forcheck bugs. These problems were sometimes compiler specific or preprocessor specific errors which were time consuming to isolate and to understand.

The makefile of JOREK was modified to provide a convenient way of calling Forcheck with the proper arguments. JOREK includes several diagnostic and post-processing tools, makefile entries for analyzing these programs were also added.

Forcheck reported some simpler errors in the code that were mostly problems with Fortran standard conformance. The real problems were uninitialized variables and issues with function arguments that were found in parts of the code that are not frequently used or that are under reconstruction.

2.8.3. GYSELA

There were problems with the HDF5 interface, which turned out to be a problem of defining the interface for Forcheck and not real errors. Forcheck problems with preprocessor instructions had to be fixed here too, and two smaller problems with Forcheck were found. The makefile has been updated to call Forcheck with the proper parameters.

A non standard conformant way of module declarations triggered a large amount of error messages, but these problems were easily fixed. After this, there were just a few errors left. These are: using a compiler specific intrinsic function, non-standard usage of optional arguments, one uninitialized variable and one problem with subroutine arguments.

2.9. FUTILS in EUTERPE code

The NEMORB code uses the FUTILS library, developed by Trach-Minh Tran, to write output data in a structured format. It was planned for the EUTERPE code to use the same output library, and it was already partially implemented. Using the knowledge gained within the project on the NEMORB code, it was feasible to support such an effort on EUTERPE. Most of the work of implementing the FUTILS output was already done in EUTERPE, except for the extensions introduced in the CKA-EUTERPE branch of the code. This branch can be used for a linear perturbative study of the stability of Alfvén modes. The I/O subroutines of CKA-EUTERPE used a direct interface to the HDF5 library which has been replaced. Now, the FUTILS library is used to handle the HDF5 I/O, which will facilitate maintenance in the future.

2.10. Summary and Future work

The aim of the EMPHORB project is to implement the phase factor transformation in the NEMORB code. Phase factor transformations can be used to decrease the spatial resolution for linear calculations and thereby lower the amount of

computational resources needed for the simulation. The implementation of the phase factors involves small changes that extend throughout the whole NEMORB code. The code structure was analyzed and the phase factors were implemented for the main steps in the PIC algorithm. This includes the matrix building routines, the field evaluation, the charge and current assignment. The equation of motion changes only for the weights, and the phase factors were introduced for pushing the weights.

With the introduction of the phase factor, several variables need 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 most of the necessary changes have been made. In the next step, the diagnostic methods will be updated for the use of complex variables. The diagnostic module uses the FUTILS library. To gain experience with this library, work has been done to improve the FUTILS interface in the EUTERPE code.

When all the changes are done extensive testing will be necessary. There are several different switches in NEMORB to include additional physical effects. Test cases will be needed to check the modified code with these options.

The modifications introduced during the EMPHORB project have been done under the analysis of the Forcheck tool, to ensure consistency of the modified code. The Forcheck tool is a comprehensive code verifier and it has been found useful for static code analysis. Several codes from the BEUPACK package (NEMORB, JOREK, GYSELA, and some libraries used in these codes) were analyzed with Forcheck. During the analysis, smaller bugs and some Fortran standard conformance issues were reported by Forcheck. Some of the reported errors were not real ones, but mistakes by Forcheck. Such mistakes were reported to the developer of Forcheck, who quickly corrected them. The remaining errors were reported to the code developers together with instruction on how to use the improved Forcheck tool.

2.11. References

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 the HLST project GYNVIZ

3.1. Initial objectives and project evolution

The aim of the project was to unify and to provide support for the whole hardware and software chain that comes from the current codes' output to the interactive and remote visualization software. Fig. 3 presents a global picture of the initial project. The evolution of the project is presented in Section 3.3.

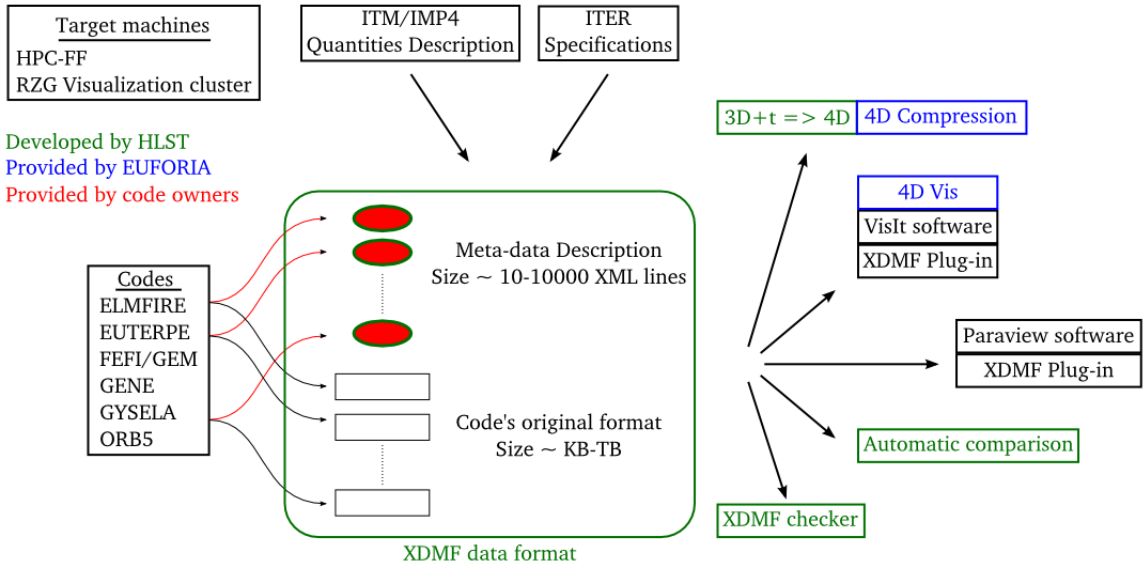


Fig. 3 Overview of the overall project.

3.2. File format technology review and initial choice

From the visualization point of view, the existing software VisIt, Paraview, Ensight, Avizio and AVS Express are the dominating ones in their field. They are mature technologies and cover almost all visualization requirements for the GYNVIZ project. So the key issue is the definition of a common data file format. It has to be technologically up to date and it must be understood by visualization software (at least VisIt and Paraview).

The latest versions of several libraries have been evaluated. They can be split into two categories. The first one focuses on computer science objects and is mainly composed of HDF5 and NetCDF4. Both are C libraries that can be called mainly from Fortran, C and C++. They are intended to create portable binary files containing regular arrays of data. The second category focuses on computational modeling objects and provides for the user a higher level of abstraction interface. Examples of this category are Silo, Exodus and XDMF. Such libraries describe with meta-data how the actual data represent computational meshes or variables that are defined on some meshes. Then, the actual data/meta-data are written by using either HDF5 or NetCDF4.

From a general point of view, the design of a brand new dedicated format represents a lot of work of development and, even more important, a lot of work of software maintenance. As GYNVIZ is a project which exists only on a fixed time frame, the software maintenance is a big issue once the project has ended. But on the other hand, to design our own format prevents having to rely on any third party. The use of an existing format presents the opposite advantages and drawbacks: little development work, little or even no maintenance cost at all, but a strong dependence on a third party.

At the beginning of the project, based on this technology review, two reasonable approaches have been identified: to design a fully dedicated HDF5 format from scratch or to use the existing XDMF format. XDMF technology offers several advantages. The first one is that the migration from the existing format of the different codes to the unified one will be easier with XDMF than with HDF5. With HDF5, either the codes' I/O is left untouched and a probably costly post-processing step becomes mandatory, or, the codes' I/O needs to be modified to include the HDF5 format. With XDMF, the different codes will only need to write on disk a small text file in addition to their current data files. This text file describes, with XDMF XML syntax, how the data are structured within the data files. As a result, the current format of the data files in the different codes is kept intact and the need for post-processing step and the development of a post-processing tool is not mandatory anymore. Another advantage is that the plug-ins for different visualization software don't need to be developed as XDMF plug-ins already exist for several visualization software tools.

On the other hand, the technology review has shown some reliability and durability issues. That is why a collaboration with the XDMF team in the USA was necessary to improve some aspects of their technology, whereas with a dedicated HDF5 format such kind of collaboration would have not been necessary. Thus, we got in contact with the XDMF development team and after an encouraging resonance we decided to choose XDMF as the common format within the GYNVIZ project. The collaboration started in March 2010.

One of our requirements in this collaboration was to implement a semantic checker tool that would check the consistency between the XML description of the data and what is actually in the data file. On the end-user side, this helps to identify and to correct any semantic errors. On the developer side, this tool helps to validate/correct all the XDMF examples and accordingly helps to correct all the existing XDMF plug-ins. HLST and XDMF teams agreed on the terms that HLST would develop the checker tool, the XDMF team would correct the plug-ins and the examples' validation/documentation would be done together. So, instead of investing the man power in the development of post-processing tools and plug-ins, it was shifted to this collaboration.

3.3. Project restructuring

Unfortunately, this collaboration failed in the middle of 2011 without any obvious reason. At the beginning of 2011, there was already a delay in the proposed development of XDMF, as the XDMF library that should have been released in September 2010 came finally out at the end of February 2011. At that time, it was still not fully functional. But as we were still in regular e-mail contact with the XDMF team, we decided to go on with our own development as scheduled. However, since the end of March, e-mails and reported bugs were not answered any more. Finally, at the beginning of the following summer we decided not to wait any longer for the XDMF official release with the consequence of abandoning the XDMF technology from the GYNVIZ project.

As a direct consequence the scope of the project had to be significantly reduced. The main impact was on the potential data types being made accessible by the project. Initially, it was planned to bring as many data types as possible under a common standard format. Such a format would have given also the flexibility to further extend it to include also 4D structured data. Without the support of XDMF and within the remaining time of the project, it was not possible to develop neither such format from scratch nor the corresponding visualization plug-ins. So we decided to abandon the complete data format standardization component of the project and to focus only on the development of 4D data visualization within a dedicated format.

In terms of lost effort, the toll is tolerable. Only the XDMF checker, developed exclusively by us for XDMF, took a significant amount of development (approximately a month) which became now needless. Instead, our manpower was mainly spent on developing the 4D technology to a mature production state. In this attempt we were successful as it is explained in the following section. However, we regret that the collaboration with the XDMF team ended that abruptly. Accordingly our users will not benefit from the so far existing XDMF technology.

3.4. GYNVIZ project outcomes

3.4.1. Visualization cluster at RZG

The visualization cluster at RZG has entered its production phase middle of October 2010[1]. The reservation of the system, collaborative work within a single session and use of 3D visualization software are the main functionalities that have been successfully tested at that time. GYNVIZ users all got access to this facility.

3.4.2. Data transfer between JSC and RZG computer centers

As the codes run on HPC-FF, they produce data on HPC-FF. In order to be visualized on the RZG visualization cluster, data have to be transferred from JSC in Jülich to RZG in Garching. The dedicated 10 Gb/s DEISA network is used for that purpose. Initially, it was planned to use the DEISA file system in order to perform these transfers transparently. After equipping every GYNVIZ member with such a DEISA account, the DEISA project ended in April 2011 and JSC was not willing to support it anymore. So we decided to switch to the *bbcp* technology for the transfer. The *bbcp* tool is built on top of *ssh* and it opens several parallel channels between the source and the destination machine in order to distribute the load for data encryption/decryption and to saturate as much as possible the network bandwidth. GYNVIZ users have been taught how to use this tool.

3.4.3. 3D+t → 4D conversion tool: gyncompress4d

This command line tool transforms 4D scalar data defined on a structured grid in 4D compressed data that can be visualized with the 4D visualization plug-in for VisIt (see next section). These 4D scalar data defined on a structured grid could describe e.g. a 4D phase space distribution function used in gyrokinetic codes (r, theta, phi and the parallel velocity) but also a 3D time varying quantity or even a 2D quantity that evolves according to two different parameters in a parameter scan, etc.

The difficulty to develop such a tool is to be able to handle a large variety of domain decompositions of 4D data. In detail, the 4D distribution function can be a single 4D block of data in a single file, or it can be scattered in multiple 4D blocks according to the data distribution in the MPI application. The 3D time varying (resp. 2D with two parameters) quantity can be a collection of 3D (resp. 2D) blocks, typically one per time step (resp. one per pair of parameters). If the prerequisite to use the tool for the end-user is to build 4D blocks from his specific domain decomposition, each end-user has to develop his own tool and it is unlikely that every potential user would invest the time to develop such tool. That is why we implemented a general solution which is able to build 4D blocks from any possible domain decomposition. In addition, as the 4D compression algorithm needs a certain number of points, i.e. $N = 2^n+1$, in each dimension, a special treatment has to be performed when this constraint is not fulfilled by the original data (almost every time). The dataset has to be either reduced or extended in order to match the constraint. Reducing the dataset is feasible but it results in a loss of information. Instead, a padding solution has been implemented in order to extend the dataset and fill the extension by using the physical boundary conditions.

[1] http://www.rzg.mpg.de/computing/hardware/miscellaneous/hp_vizcluster

The entry point of such a tool is the description of the underlying domain decomposition. Originally, the XDMF language should have been used (and extended) for that purpose. Instead, a dedicated ASCII file replaces it. The following example shows the kind of file which describes 4D data in single precision with 33 points in each dimension split into two blocks and without boundary conditions. Each block is contained in a different file.

```
dtype:  <type 'numpy.float32'>
size_tot:  [33, 33, 33, 33]
nb_bloc:  [1, 1, 1, 2]
bound_cond:  BC_NONE BC_NONE BC_NONE BC_NONE
bloc:0
filedesc:  filename=/home/mhaef/test_export-import/res_0-0-0-0.h5 dsetname=data
  start:  [0, 0, 0, 0]
  size:  [33, 33, 33, 16]
bloc:1
filedesc:  filename=/home/mhaef/test_export-import/res_0-0-0-1.h5 dsetname=data
  start:  [0, 0, 0, 16]
  size:  [33, 33, 33, 17]
```

According to the description of the domain decomposition and the available memory on the machine, the application decides how many 4D blocks are built in memory. Indeed, a large 4D dataset can easily exceed the amount of available memory. Each block is treated serially, i.e. it is read form disk (potentially several files need to be read) and is passed to the compression library algorithm developed in the framework of the EUFORIA project. In order to enable this, a python interface has been added and some internal modifications were mandatory to make the library thread-safe and to make it accept simple precision floats. The library exports directly the 4D compressed data once the compression algorithm is finished.

The development of the *gyncompress4d* conversion tool has been finished, tested and installed on HPCFF, the RZG visualization cluster and the IFERC computer. From the technical point of view, this tool has been implemented in Python. It uses the *numpy* and *h5py* legacy packages as well as the *calviExport* package from the EUFORIA project. In total it consists of 6500 lines of Python code which includes around 300 unit tests. Some bugs in the *calviExport* library have been fixed, a Python interface has been added and the library is now thread-safe.

3.4.4. 4D visualization plug-in for VisIt

The visualization plug-in enables the visualization of data generated by the *gyncompress4d* tool and was originally provided by EUFORIA. It became obvious quite soon that the plug-in was not in an operational shape. A couple of bugs were found and have been corrected. In addition, some announced functionalities were simply not implemented. Due to a complex software infrastructure, the detection of these bugs was a tedious process which required refactoring/rewriting some parts of the plug-in. As a result, the whole plug-in has been refactored and extended and is now ready for production. A document for developers has been written to keep track of the plug-in software architecture in order to ease the porting to any future versions of VisIt.

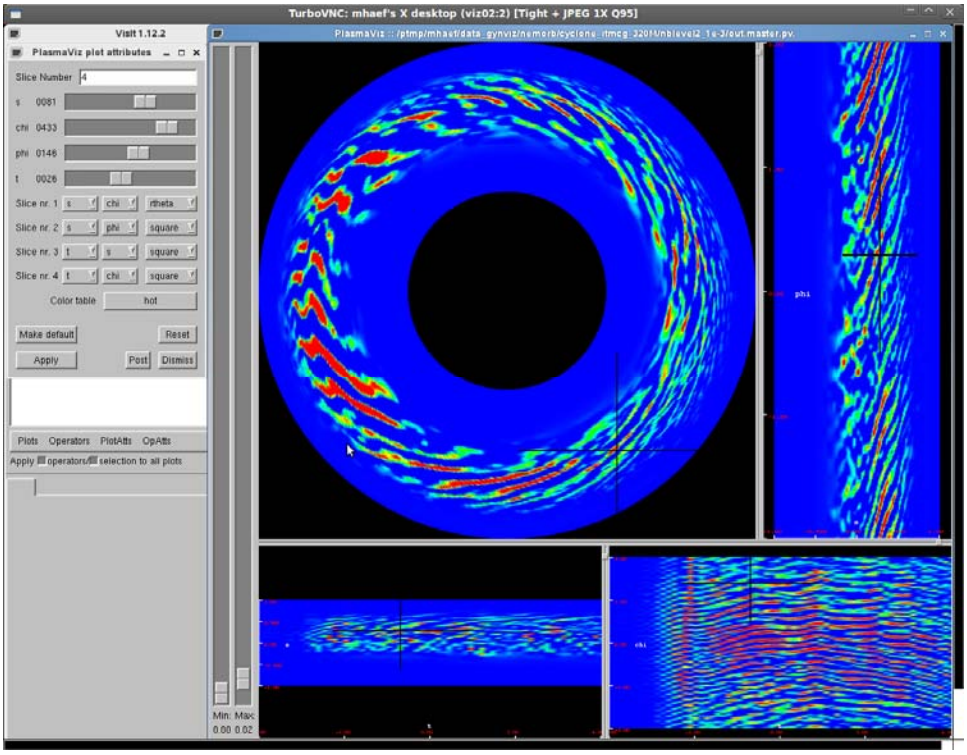


Fig. 4 Screenshot of PlasmaViz plug-in for VisIt on the RZG visualization cluster showing as a single 4D dataset the time evolution of the 3D electrostatic potential Φ-<Φ> coming from the NEMORB code.

Fig. 4 shows four 2D slices of the time evolution of the 3D non-zonal electrostatic potential Φ-<Φ> as a single 4D dataset. From top to bottom and left to right, we have a *r*-*θ* slice (poloidal plane), a *r*-*φ* slice (toroidal plane), a *t*-*r* slice and a *t*-*θ* slice. To select the *r*-*θ* slice for example, one has to fix the value for the *φ* and *t* coordinates. This is done by the position of the black cross in the *r*-*φ*, *t*-*r* and *t*-*θ* slices where these dimensions appear. The general idea of this 4D visualization method is to enable the user to move interactively the crosses in the different 2D views in order to explore the 4D volume. The challenge is to quickly refresh interactively these slices and to keep the size of stored data low. In our example the original data have been compressed by an order of magnitude from 4 GB to 490 MB. Hence, an interactive scan of the data is based on 4D compression and fast on the fly 2D reconstructions. In the same manner, Fig. 5 shows similar data with higher resolution. Here, the compression rate is even higher so that the original 129 GB of data are reduced to 6 GB after compression. This example clearly shows that an interactive scan on the basis of original data of 129 GB size would be too inefficient to be feasible.

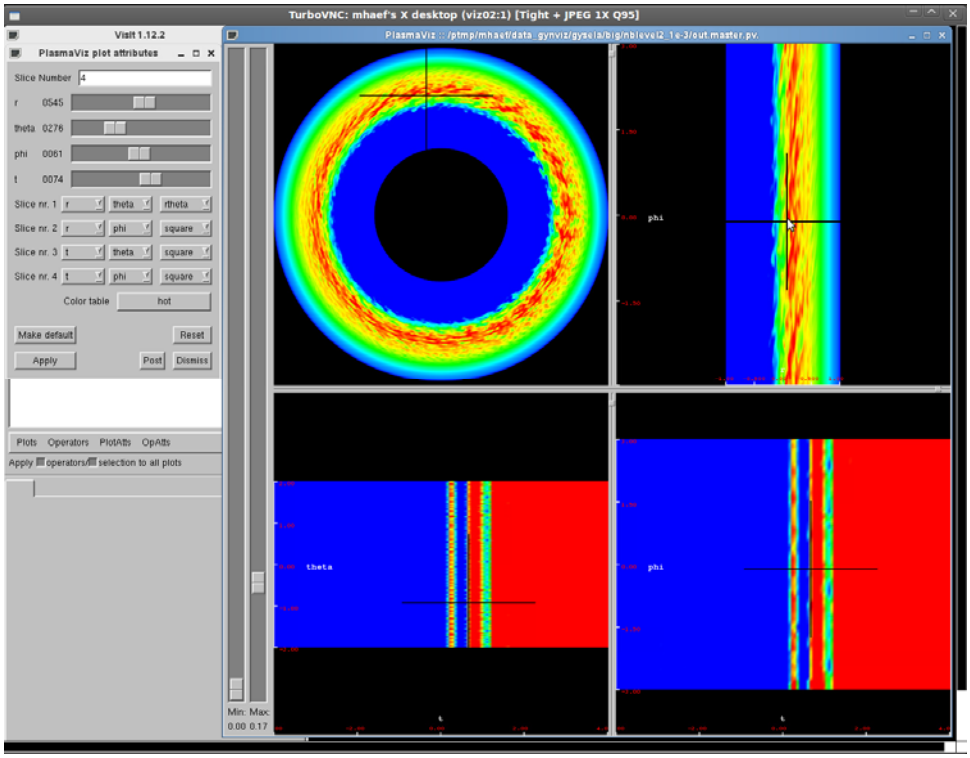


Fig. 5 Screenshot of PlasmaViz plug-in for VisIt on the RZG visualization cluster showing as a single 4D dataset the time evolution of the 3D electrostatic potential Φ-<Φ> coming from the GYSELA code.

3.4.5. Parallel I/O study

The current trend of computational power growth is mainly based on the increase in the number of cores. This phenomenon modifies deeply the way applications should be programmed. Indeed, to benefit from this computational power, applications must be parallelized and must scale very well. In particular, applications' I/O will definitely become an issue on the next generation of machines. Although this topic is not the core of the GYNVIZ project, it is strongly related as the large datasets must be first written on disk before their visualization.

The purpose of the study was to evaluate 13 different I/O methods that write a 2D array on disk from a distributed application using a block-block distribution layout. Evaluations have been performed on two different architectures: HPC-FF at JSC (Lustre file system) and VIP at RZG (GPFS file system). Interpretation of the results and programming advices are given in a separate report available on the HLST website[2]. This report has been disseminated within the HLST, GYNVIZ project, RZG and EUFORIA.

The study has improved our knowledge on parallel I/O in general and in particular on HPC-FF. This is of key interest for next generation computers, especially IFERC.

3.5. Conclusions

The original scope of the project had to be reduced because of the failure of the collaboration with the XDMF development team. The major loss is the usage of existing XDMF technology in the framework of GYNVIZ. So the initial intention of bringing a common and standard format to a range of different codes had to be given up. Nevertheless, the main focus has always been to bring the 4D visualization functionality to a mature production state, which has been achieved. A data size reduction by several factors up to an order of magnitude can be expected as a result

[2] http://www.efda-hlst.eu/training/HLST_scripts/comparison-of-different-methods-for-performing-parallel-i-o/at_download/file

of data compression. The interactive visualization of the resulting 4D dataset is available on the remote visualization cluster at RZG. GYNVIZ users have now started to use these new tools and the first feedback is positive. The maintenance of the software package will go on with low priority although the GYNVIZ project has officially reached its end.

4. 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 IFERC-CSC. At the beginning, the project went into operation by supporting selected European projects during the lighthouse phase. After the HELIOS machine went into production phase starting in April 2012 the support activities have been extended 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, testing the file transfer between Europe and CSC in Japan and assessing the hardware capabilities of HELIOS.

4.2. CSC-Europe file transfer

To post-process the data of large simulations becomes an issue when the files reside at IFERC-CSC in Japan and the scientists are located in Europe. Even if remote visualization is available at CSC, large amounts of data have still to be transferred to Europe. To make matters worse, the login nodes of HELIOS have very strict network security standards. Only the incoming port 22 dedicated to ssh is available for both login and file transfer. With the help of Rechenzentrum Garching (RZG), a dedicated node at CSC has been allocated for file transfer. It comes with optimized network parameters and additional opened ports. This allows for a much better network connection between RZG and CSC with efficient tools for file transfer. To make this opportunity even more attractive, RZG offers archive space for any European scientist working on HELIOS. Post-processing and visualization resources can be also provided on request. People being interested should contact RZG.

The technical details of the provided environment are the following:

- At RZG (Germany):
  - a dedicated node (viztrans) with optimized network parameters and additional opened ports
  - a tape-based archiving system (/r)
  - a scratch file system for post-processing purpose (/ptmp)
  - post-processing and remote visualization resources on RZG's visualization cluster (on request)

- At IFERC-CSC (Japan): a dedicated node (bbcp.iferc-csc.org) with optimized network parameters and additional opened ports to allow the usage of bbcp from viztrans

Daily bandwidth measurements show that the bandwidth from CSC to RZG is between 5 and 10 MB/s and between 10 and 15 MB/s in the opposite direction. Such transferred rates can be considered as reasonable.

4.3. MPI library behaviour on HPC-FF and HELIOS

The Message Passing Interface (MPI) library has become the standard to develop parallel applications that run on distributed supercomputers like HPC-FF and HELIOS. The MPI library provides to the developer a set of well defined communication patterns. When these routines are called, they consume a certain

amount of network resources, but also memory resources as they occasionally copy data into intermediate buffers. The purpose of this study is to evaluate the different time and memory overheads introduced by certain implementations of the MPI library. Under consideration are the Intel and Bull MPI libraries on HELIOS and their scaling property on large number of cores. HPC-FF will be used as a reference for HELIOS jobs up to 4096 cores.

We begin our assessment with the memory consumption of the MPI_INIT routine which has to be executed to set up the MPI environment. In Fig. 6 we show the memory footprint of the MPI_INIT operation per MPI task in megabytes (MB). We ran one MPI task per core for Parastation MPI on HPC-FF and for both Bull and Intel MPI on HELIOS. The memory footprint increases with the number of cores which is somehow expected as the potential number of connections increases with the number of MPI tasks. It increases up to 170 MB on HELIOS for Bull MPI and 26 MB for Intel MPI on 32K cores. This represents respectively 4% and less than 1% of the total memory per core (4 GB). The situation for Parastation MPI is more complicated. The default behaviour on HPC-FF is that the maximal number of buffers are already allocated at the MPI_INIT. This results for example in a memory consumption of more than 1 GB for 2K cores. Hence, for comparison reasons we switched off the automatic allocation for all following runs (setting the Parastation MPI environment variable PSP_ONDEMAND=1) to make the results comparable with the results form the two MPI implementations on HELIOS. It can be seen that the Parastation MPI has the least memory consumption.

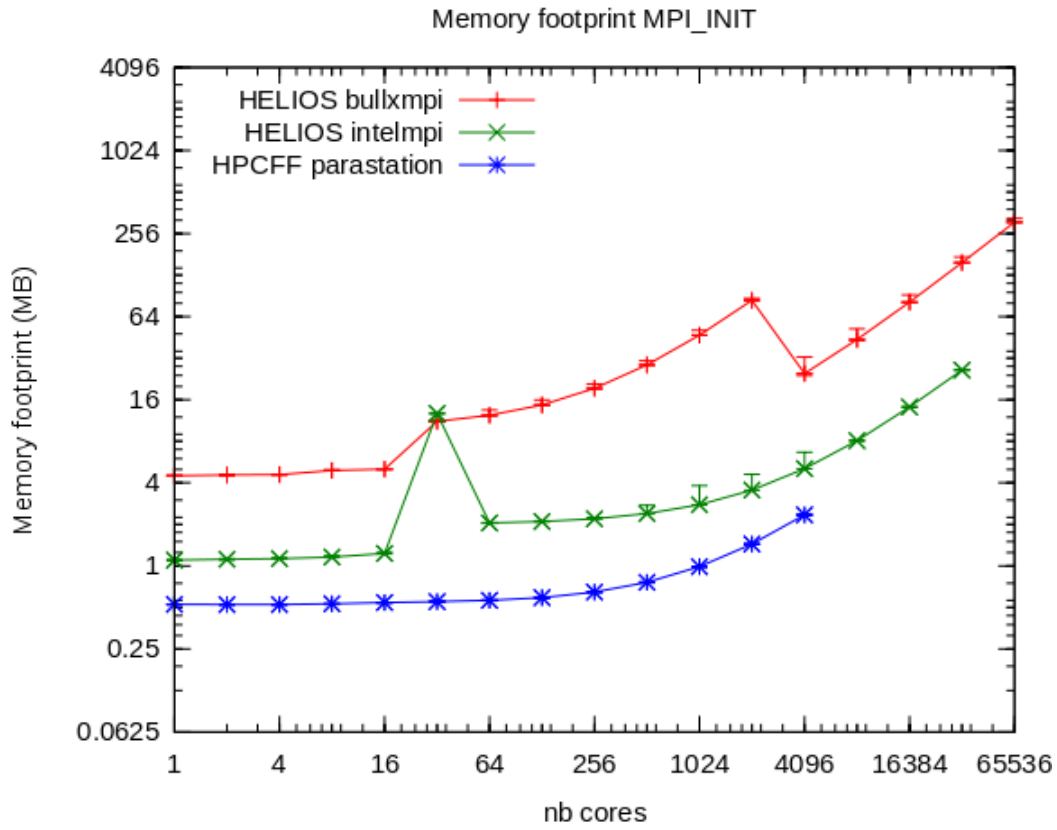


Fig. 6 The memory footprint per MPI task in megabytes (MB) of the MPI_INIT operation as a function of the number of MPI tasks. The on demand connection is set to one in this example which means that only some memory allocation is postponed to the first following MPI calls.

In addition, we decided to focus on three main functionalities of the library that are definitely needed by the users: the barrier, the gather and the all-to-all operations. Within the NEMOFFT project, Tiago Ribeiro studies ways on how to improve one of the current bottlenecks of the NEMORB code: a matrix transposition that requires an all-to-all communication pattern. Building up on his work we extracted somewhat

simpler 2D matrix transposes as test cases. In the following figures, we will investigate the scaling properties of these operations with increasing number of cores. The averaged quantities (execution time or memory footprint) are displayed as a function of the involved MPI tasks. In addition, the minima and maxima are displayed as error bars.

The execution time of a single barrier as a function of the number of cores is shown in Fig. 7 for Parastation MPI on HPC-FF and for both Bull and Intel MPI on HELIOS. The execution time increases with the number of cores, which is expected as the synchronization over more cores implies more traffic on the network. As one can see the behaviour of the HELIOS machine until 32K cores is reasonable. On 64K cores, the execution time with Bull MPI is much worse and with Intel MPI, the application crashed before all measurements could be written on disk.

In Fig. 8 we show the memory footprint of the MPI_GATHER operation per MPI task in MB. We ran one MPI task per core for Parastation MPI on HPC-FF and for both Bull and Intel MPI on HELIOS. This weak scaling study consists of gathering all the 64KB arrays allocated by each task on task zero.

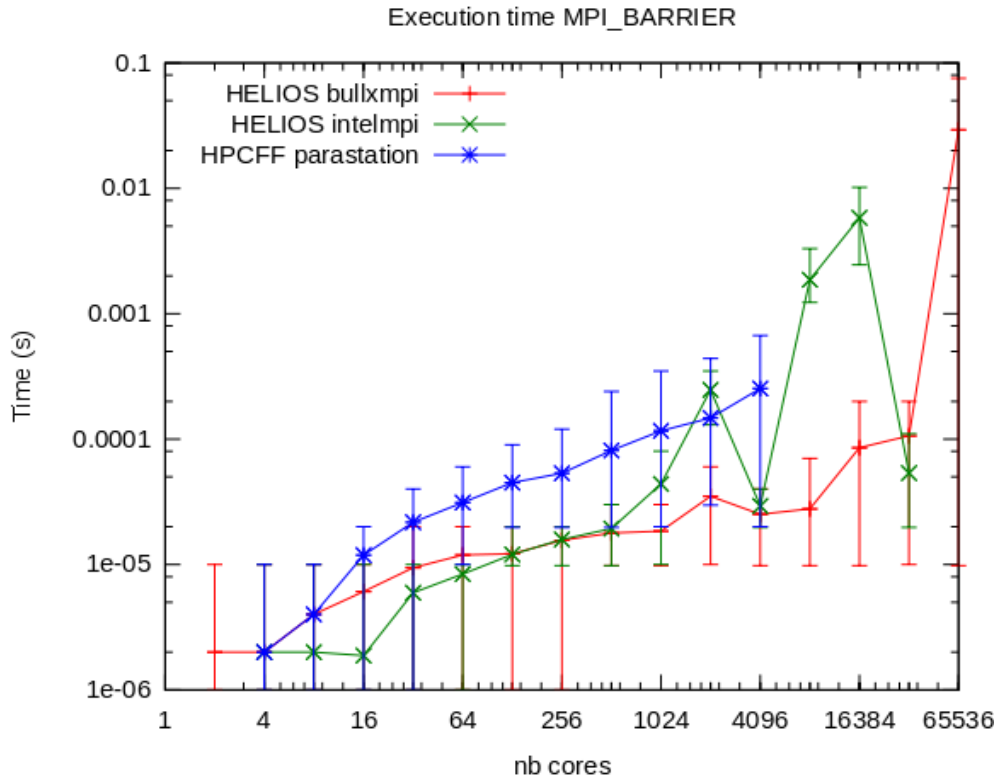


Fig. 7 Execution time of a single barrier as a function of the number of cores for Parastation MPI on HPC-FF and for both Bull and Intel MPI on HELIOS.

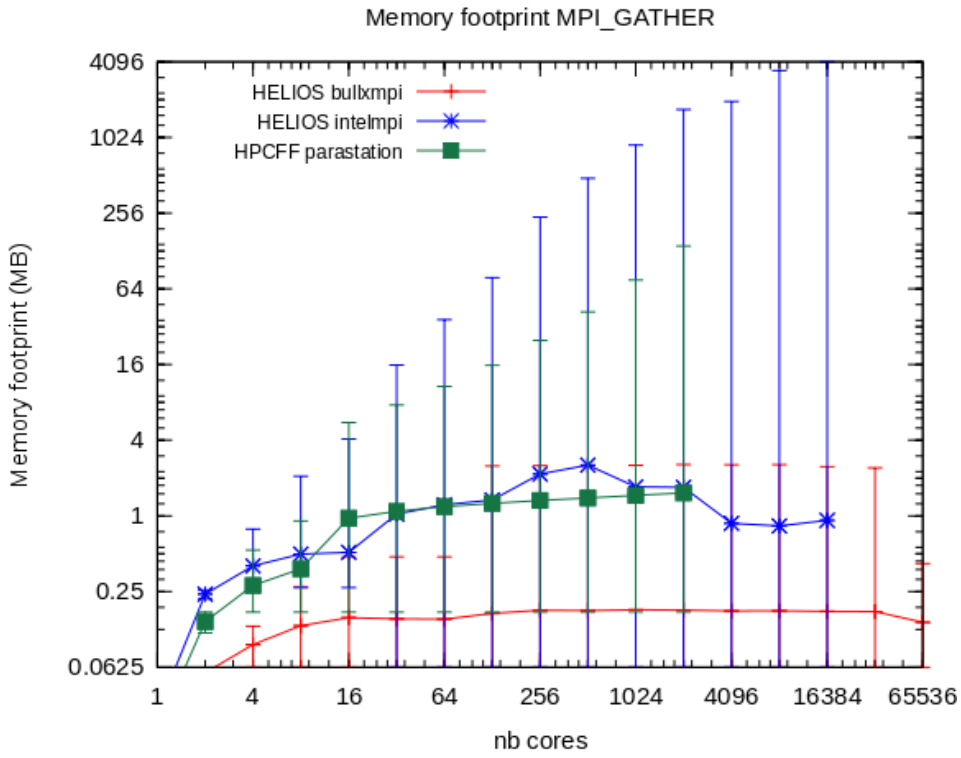


Fig. 8 Memory footprint per MPI task in MB of one MPI_GATHER operation as a function of the number of MPI tasks.

The simplest implementation would be if each MPI task would send directly its data to the gathering task. In such a case the memory footprint should be exactly the same on each core except on the gathering core hosting task zero. But this would lead to large network contention. Instead, it is much more efficient to gather the data progressively using a binary tree communication pattern. This algorithm needs only *log2(n)* messages, instead of the *n-1* of the former, but on the other hand, increases the memory footprint of the tasks that are nearer to the gathering task in the tree. This explains the discrepancy between the minimum, maximum and average memory footprint for all three curves.

The memory footprint of MPI_GATHER for Bull MPI on HELIOS is much less than for the other two MPI implementations. One reason might be that some memory had been already allocated at the MPI_INIT level, which has a higher footprint for Bull MPI. But still the maximum footprint seems to saturate at very reasonable levels whereas Intel MPI allocates more and more memory up to the point where the application crashed, on 8K cores. We believe that Bull invested special effort to optimize the memory impact of its MPI. Nevertheless it should be mentioned that Intel MPI proposes several environment variables to further optimize the behaviour of its MPI.

Concerning execution time (not depicted here), our measurements show a linear increase according to the number of cores. The execution spans from 10^{-5} s on one core up to 0.5 s on the 64K cores.

Fig. 9 (resp. Fig. 10) shows the memory footprint in MB of the all-to-all operations on HELIOS for both Bull and Intel MPI library (resp. for Parastation MPI on HPC-FF) for different number of cores. The problem consists of transposing a matrix of size 8K x (8K x nb_cores) where the second dimension is distributed among the MPI tasks. It means that it is a weak scaling where each MPI task holds a constant 8K x 8K matrix piece (256 MB). The following four methods have been evaluated:

- ALLTOALL with buffer: the ALLTOALL MPI primitive is called to make the transposition, but the data are arranged contiguously at the application level.
- ALLTOALL with derived types: the ALLTOALL MPI primitive is called to make the transposition and the data are arranged contiguously by MPI.
- GATHER SCATTER: each MPI task performs a GATHER to collect the data it has to work on and scatters back the result.
- XOR: *N/2* MPI tasks exchange data with the remaining *N/2* in a point-to-point send/receive manner. All possible pairs are formed sequentially via a logical XOR condition within a *for loop* so that every task exchanges data with every other task.

One can see that the all-to-all communication pattern is much more memory consuming than a single gather (see Fig. 8) even on a small number of cores. Except for the XOR method, the footprint remains around 1 GB and starts increasing in some cases when it is run on 256 cores or more. This represents a significant but still affordable cost. Concerning the execution time, the picture is more complex. MPI libraries can introduce large execution time overheads on the first MPI call. We named this “the initialization time issue” and details are described in the next section. For a detailed comparison between these methods, please refer to the section related to the NEMOFFT project.

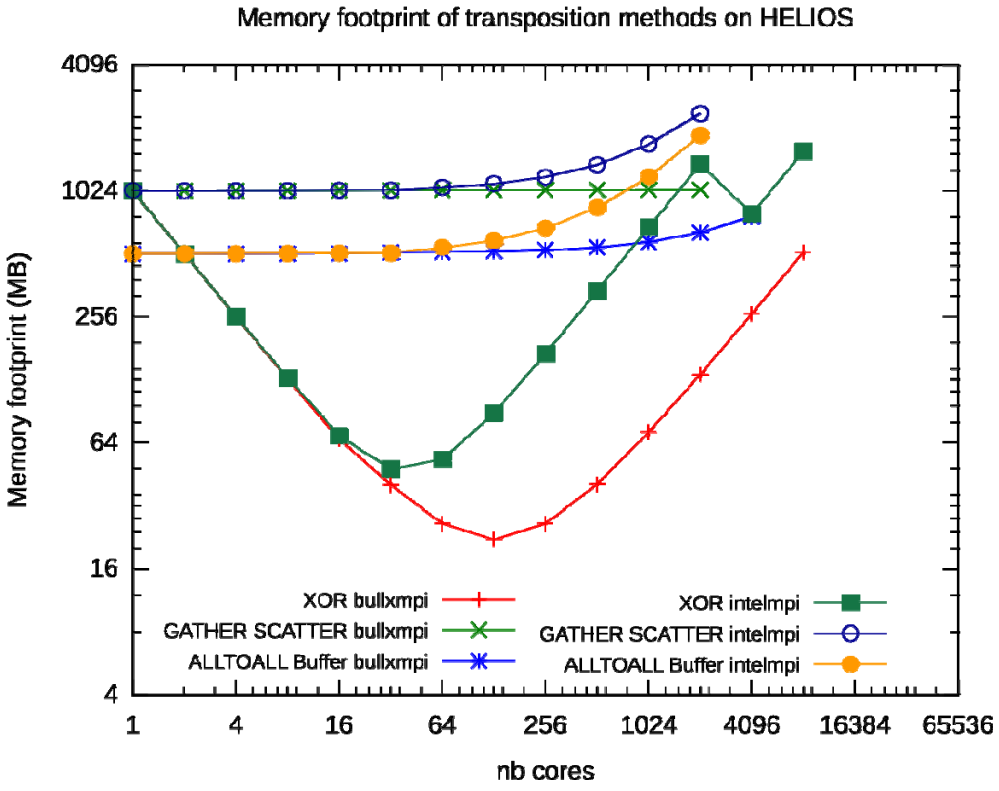


Fig. 9 Memory footprint per MPI task in MB of different all-to-all operation as a function of the number of MPI tasks on the HELIOS machine.

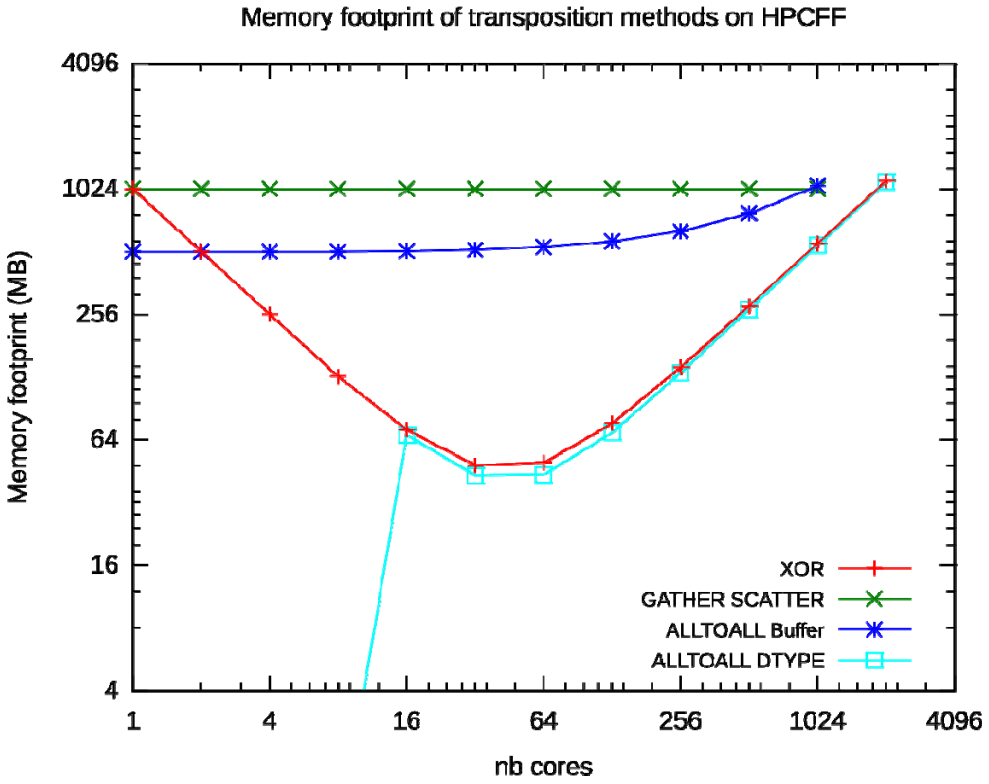


Fig. 10 Memory footprint per MPI task in MB of different all-to-all operation as a function of the number of MPI tasks on the HPC-FF machine.

4.4. MPI library initialization time

We have figured out that by executing several times the same primitive, the execution time changes drastically, with the first execution being considerably slower than the next ones.

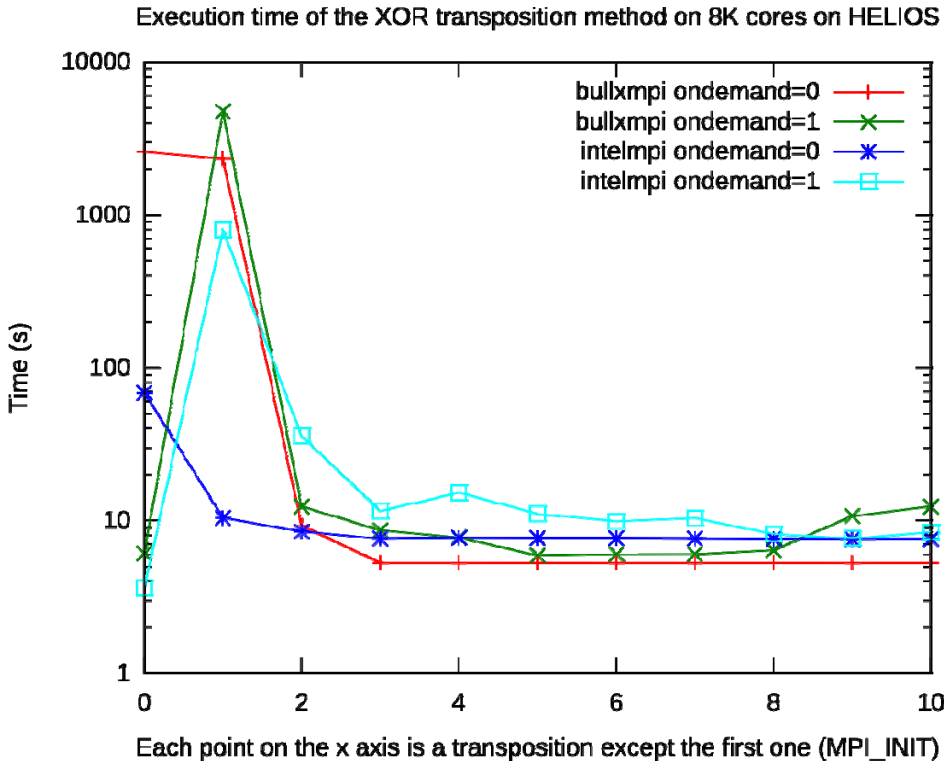


Fig. 11 Execution time of the MPI_INIT followed by ten matrix transpositions on 8192 MPI tasks on HELIOS.

Fig. 11 shows the execution time of the MPI_INIT operation and ten executions of the matrix transposition operation. This operation involves an all-to-all communication and it was performed using the XOR method on 8K MPI tasks. The two MPI libraries, Intel and Bull, are compared and for each the influence of the "on demand allocation" parameter is studied. Setting this parameter to zero demands the MPI library to build a fully connected net between all MPI tasks, e.g. each task has a connection to all the other MPI tasks. This invokes the opening of N^2 network connections where N is the total number of MPI tasks (note, each task has a connection to itself). On the contrary, setting this parameter to one lets the MPI library open the connections dynamically only when they are requested by the different MPI primitives. In our specific matrix transposition operation, a full connected net is required. Regardless the value of the "on demand allocation" parameter, after the first execution of the transposition, the MPI library has opened all the N^2 network connections. So, it makes sense to compare these time executions because they measure different ways of reaching the same software state.
One can obviously see the overhead of creating this amount of connections. This overhead is either located at the MPI_INIT when "on demand allocation = 0" and on the first transposition otherwise. The only exception is the behaviour of Bull with "on demand allocation = 0", where one can see an overhead on both MPI_INIT and the first transposition. This is not understood and the information has been forwarded to Bull. Once these connections are established, both libraries behave roughly identically with a matrix transposition that lasts between 5 and 15 seconds, which is reasonable according to the transferred data size.

By considering the time execution of the 5th operation reliable, we define the initialization time t_{init} as:

$$t_{init} = t_{MPIinit} + t_{op1} + t_{op2} - 2\, t_{op5}$$

Fig. 12 shows the evolution of this initialization time when the matrix transposition is executed on different number of cores. One can see that this time increases linearly with the number of cores for Intel MPI, but rather quadratically for Bull MPI. Especially for Bull, 1h22min initialization time on 8K cores in queues with a maximum wall clock time of 24h becomes critical. It would be even worse with greater number of cores. This issue is now clearly identified and Bull is working on it actively.

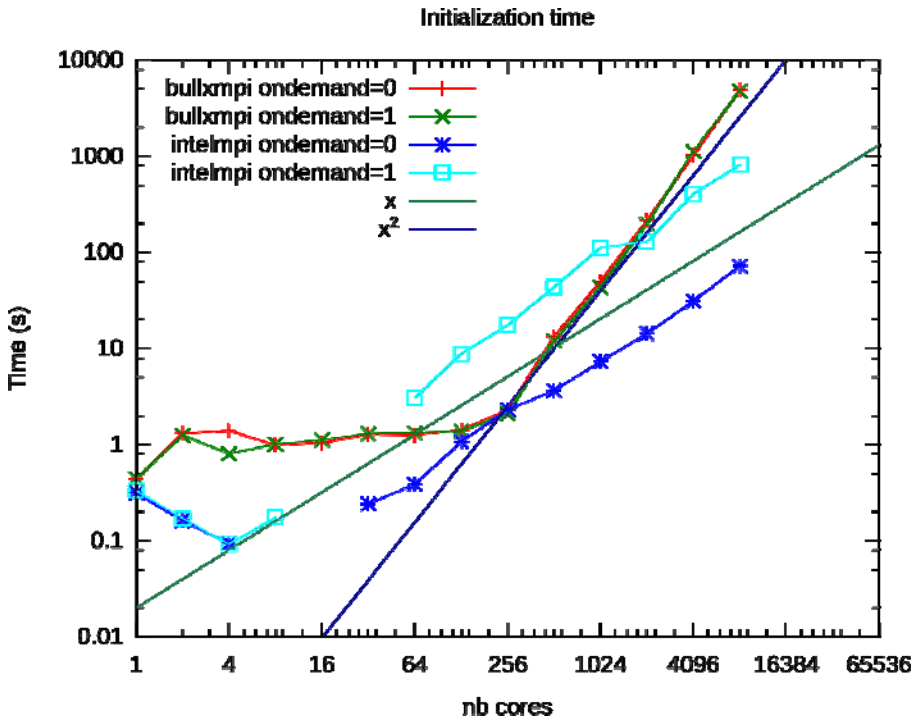


Fig. 12 Initialization time for the matrix transposition for different number of cores.

4.5. Clone computing strategy

In order to improve the scaling property of a parallel application, one can use the so-called clone strategy where it applies. Typically, instead of running the application on *N* cores, one uses *c* clones, each of which runs on *N/c* cores. In total, the same computing resources are used, but the most demanding collective operations (the ALL_TO_ALL operation in our case) are performed on *N/c* cores instead of *N*. So a better scaling on this part of the application is expected. On the other hand, the outcome of all the clones should be gatherable in some way because the parallel application solves one single problem and not *c* independent ones.

This strategy is used in ORB5. At the beginning of a time step, all the clones have a copy of the fields and hold their own set of particles. Each clone then pushes their particles according to the fields and computes the new fields. At this point, each clone has a different version of the field but thanks to the field property, the global field is simply the sum of all these contributions. From a technical point of view, a MPI_ALL_REDUCE performs this sum efficiently and allows the next step to be computed.

If we come back now to our matrix transposition problem, the clone strategy would significantly help to reduce the initialization time. Indeed, instead of creating N^2 connections, the MPI library has to create only $c(N/c)^2$ ones. Typically, on 16K cores, instead of opening $2.7 \cdot 10^8$ connections, the MPI library has to open only $1.7 \cdot 10^7$ connections with the use of 16 clones of 1K cores each.

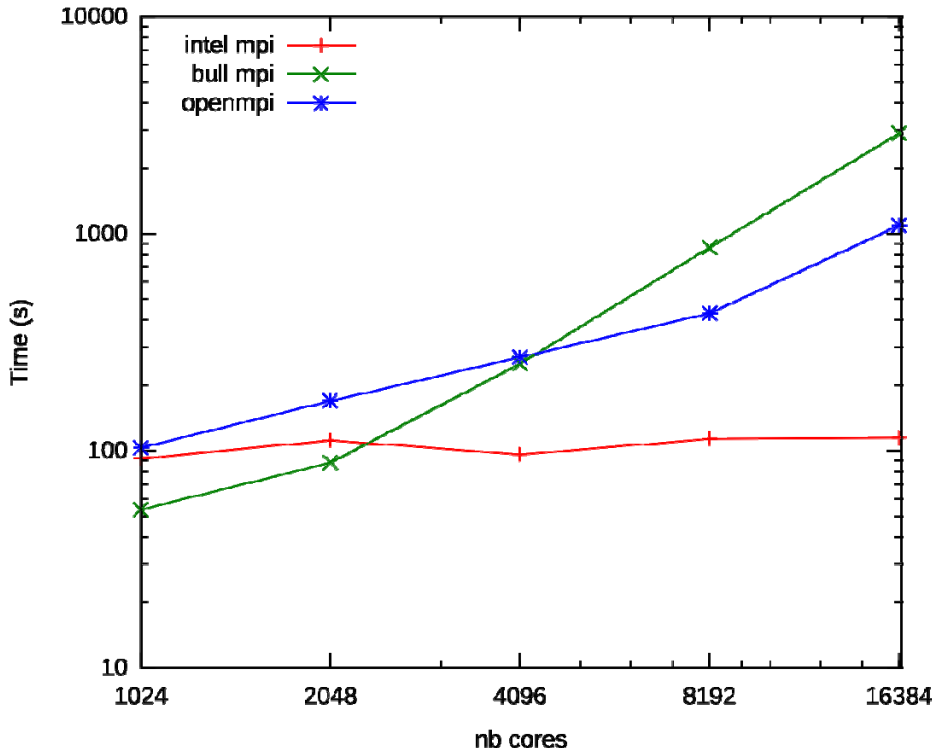


Fig. 13 Initialization time with tasks grouped by clones of 1024 tasks each.

Fig. 13 shows the initialization time for different number of clones of size 1K tasks each. In principle, the total number of connections is a hint of how long this initialization process should last. In practice it really depends on the MPI library, the operating system, the node hardware as well as the network switches. So it is very difficult to evaluate either an upper or a lower bound. We can only consider this initialization time as reasonable or not. One can see that Intel MPI is surprisingly flat whereas Bull MPI increases up to 48 min on 16K cores. This kind of timing does not seem to be reasonable, at least in comparison to Intel MPI, and we see it as strong evidence for an underlying problem.

This issue has also been reported to Bull and we are collaborating on this particular point. The HLST gave access to the source codes of some of the examples to Bull so they can have a look at it and run it by themselves. We have started to evaluate the behaviour of OpenMPI as well, because Bull used it as a base to build their own MPI library. The intention was to check whether Bull's development broke something within the library or not. As we can see, OpenMPI behaves better than Bull MPI with large numbers of clones and vice versa for small numbers of clones. According to Bull, the difference in the behaviour is likely to be a difference in the library configuration rather than in the implementation. We also ran some tests with MVAPICH2, the basis Intel used to build its MPI library in order to understand whether this good behaviour was originally implemented in MVAPICH2 or if Intel improved it. It turned out that MVAPICH2 is not able to run with 2K MPI tasks or more on the HELIOS machine. No more effort has been invested in that direction.

4.6. Ideas to circumvent the initialization time issue

We have seen that the number of connections required by our test case grows with N^2, with N being the number of MPI tasks. Up to now it was assumed one task per core in a context of a pure flat MPI application, i.e. no hierarchy exists between the tasks. Let us introduce now a one level hierarchy among the tasks and let us build groups of N_g tasks. Among each group, one task is both responsible for the communication with all the other groups, and for the data gathering and scattering needed within the group. One can see this particular task as a communication relay. By making a distinction between inter-group communicating and the remaining tasks,

one can reduce the number of required connection for our test-case. Typically, this number will fall to $(N/N_g)^2$ because the full connected topology is only created among the inter-group communicating tasks. Three parallelisation concepts are foreseen:

- MPI+OpenMP, so-called hybrid model: The idea is to allocate at least one task per computing node and to use N_g threads at the computing level. This is possible because all threads share the memory of the MPI task. This solution seems to be the most efficient one on the paper because it really reduces the use of MPI. On the other hand, OpenMP directives, as well as threads synchronisation have to be introduced in the code, which can be very time consuming for the code author.

- Pure MPI: The idea is to stick with one MPI task per core. By the use of MPI communicators, it is possible to select a single task among the group of tasks to perform the communication with the other groups. So when a collective communication is needed, first the inter-group communicating task gathers information from all the tasks belonging to the group. Then all inter-group communicating tasks exchange information and finally, the inter-group communicating task scatters the freshly received data to the other tasks within the group. This method seems to be the fastest to implement but presents an overhead in memory and requires the establishment of $(N/N_g)log(N_g)$ additional network connections.

- MPI + shared memory segment: As with the pure MPI solution, one MPI task per core is allocated. But instead of working on its own private memory chunk, all the tasks within the group would work on a shared memory segment. So the group cannot be bigger than a node. The additional resources are not any longer required. But the shared memory segment has to be the allocated and some explicit synchronizations among the tasks have to be introduced in order to avoid race conditions.

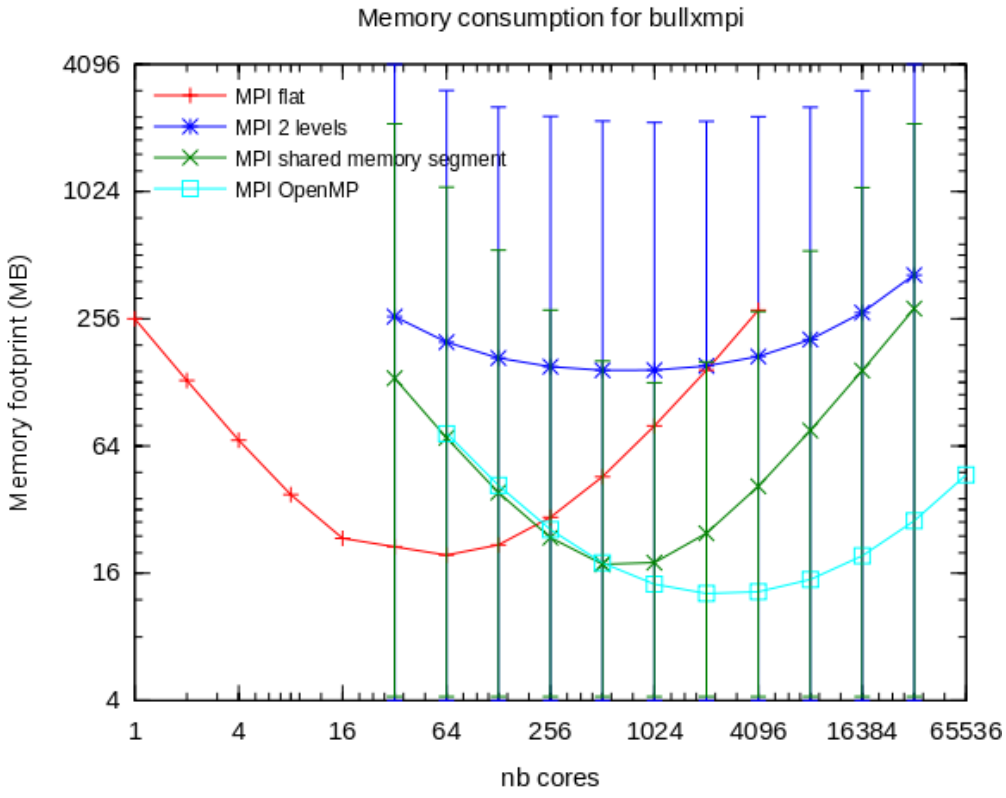


Fig. 14 MPI memory footprint for the different tested methods using Bull MPI on HELIOS.

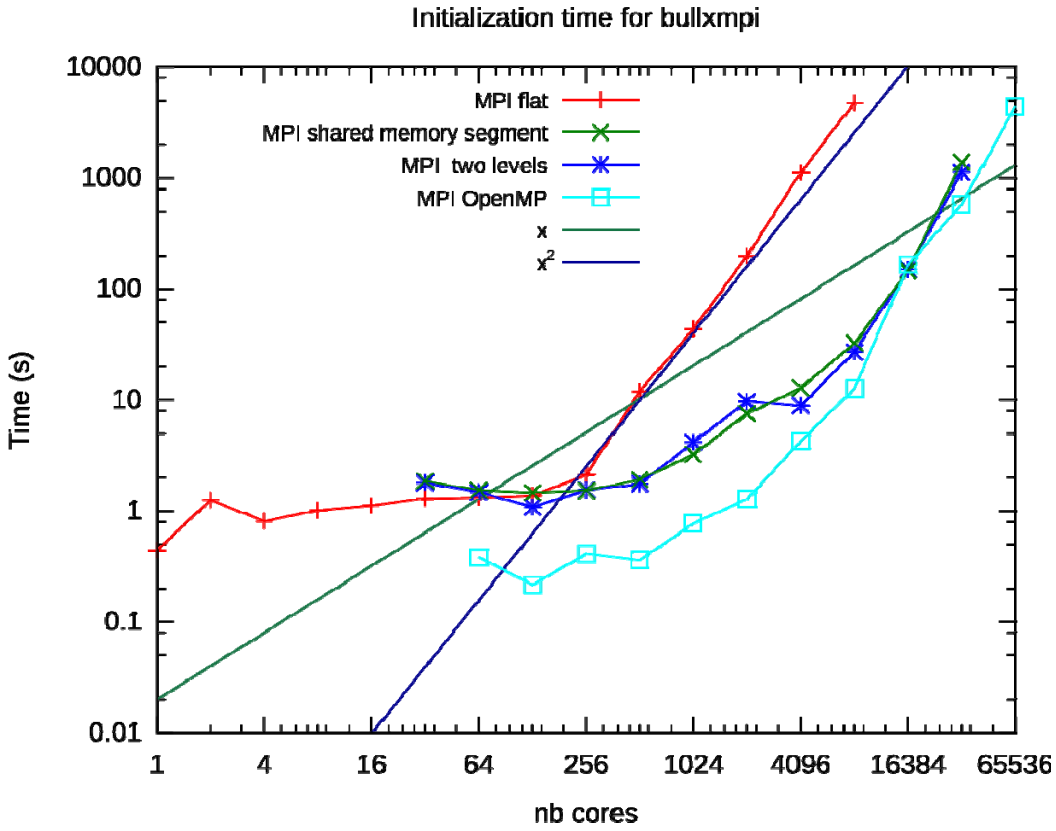


Fig. 15 Initialization time for the different parallelization concept using Bull MPI on HELIOS.

Fig. 14 shows MPI memory footprint for the different parallelization concepts as a function of the number of used cores and Fig. 15 shows the corresponding initialization time. As we can see, it is possible to shift both initialization time and memory footprint issues towards larger numbers of cores. One can note that the transposition of a distributed matrix can be performed on the full system by using these new parallelization concepts.
For the initialization time and the memory footprint, one would expect a reduction of factor 16 as the number of communicating tasks in the ALL_TO_ALL communication are reduced by the number of cores per node. However, the initialization is only reduced by a factor of 8 while the memory footprint is reduced by at least a factor of 16 and even more for the MPI OpenMP parallelization strategy. As can be seen in Fig. 16, there is a price to be paid on the execution time of a single transposition, which becomes a factor 12 more costly.

The reasons for this performance drop lay essentially at single node performance of the transpose algorithm. Handled within a *for loop*, all possible pairs of MPI tasks interchange data with each other via a logical XOR condition. Each iteration of this loop can be split into three steps.

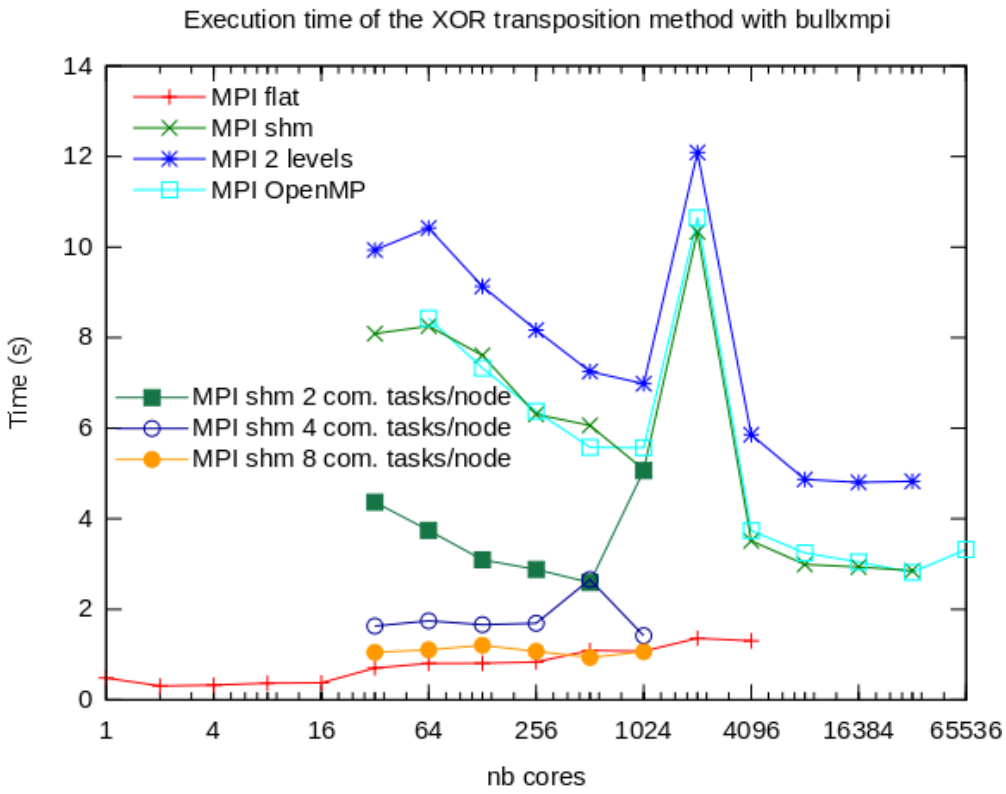


Fig. 16 Execution time of the different tested methods using Bull MPI on HELIOS.

First, the part of the array which has to be sent is copied into a buffer. Then, this buffer is sent via MPI to another MPI task which concurrently replies by sending a buffer of the same size. Finally, the received buffer has to be locally transposed and inserted into the final transposed array. Fig. 17 shows the contribution of these three steps to the total execution time according to the number of cores. We clearly see a large contribution of the local transpose to the total execution time, especially for small numbers of cores. As the number of cores increases, this contribution becomes smaller and smaller because the buffer that has to be transposed shrinks and can finally benefit from better cache reuse. So there are two key points for improvement. First, the actual transposition implementation has to reuse as much data as possible already being present in the cache. Efficient loop blocking strategies can be used for that purpose instead of the Fortran intrinsic function *TRANSPOSE*. Second, as the transpose algorithm is a pure memory operation, performance is bounded by the memory bandwidth. A single thread as we are using in our current implementation is not sufficient for saturating the bandwidth. Hence, an OpenMP implementation is mandatory to reach high single node performance. Finally, we hope to reduce the local memory copy and transposition costs in such a way that the MPI data transfer becomes as expected the dominant process. First results are promising.

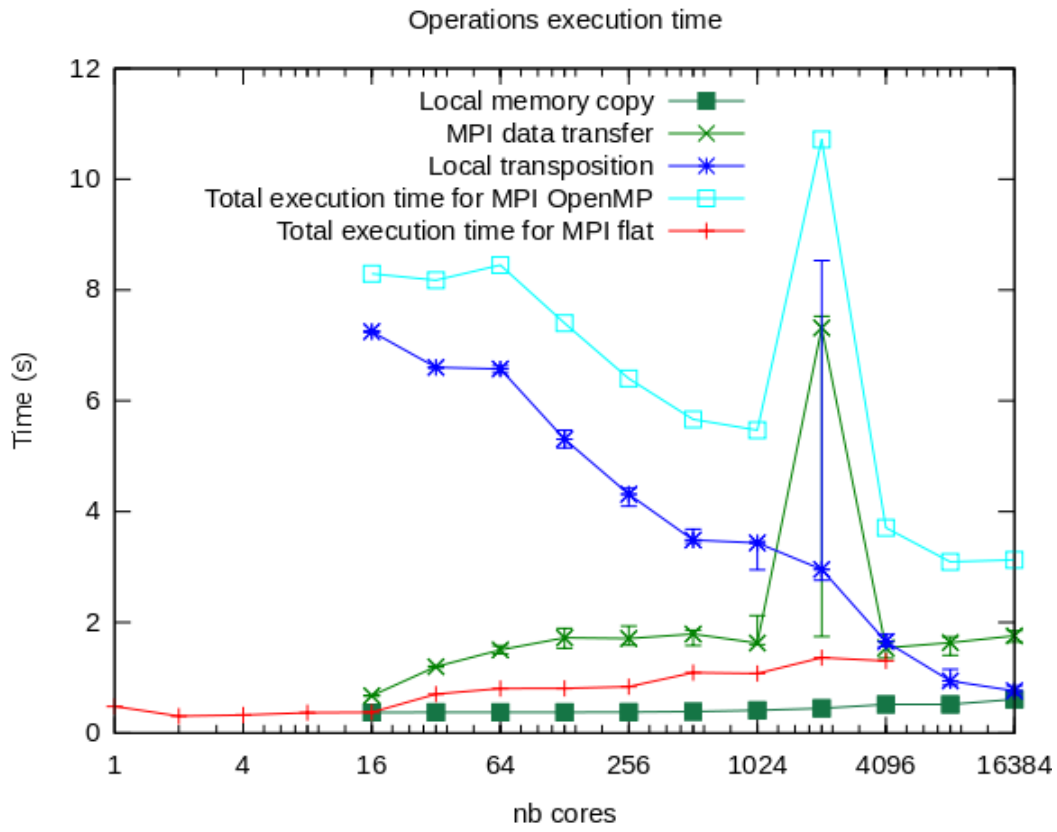


Fig. 17 Execution time of the MPI OpenMP method decomposed in its three parts. The spike that appears exclusively when using 2048 cores is not yet understood. This issue has been reported to BULL.

4.7. Computer architecture comparison

We had the opportunity to run our ALL_TO_ALL benchmark also on other HPC architectures we have access to. It gave us the chance to evaluate to which degree other HPC systems exhibit the same kind of initialization time issue. Fig. 18 and Fig. 19 show that initialization time overhead exists on all systems under investigation and that it increases with the number of cores. However, the level, at which this cost becomes an obstacle, is reached much earlier in the case of the two Bull machines IFERC-CSC and TGCC in France, although it is worth pointing out that both machines did not behave exactly the same way. It is further planned to run the benchmark on the recent IBM BlueGene Q machine at Jülich Supercomputing Center (JSC) and on the recent Intel based IBM machine of Leibniz Rechenzentrum (LRZ) in order to get a clear picture up to 64K cores on those architectures.

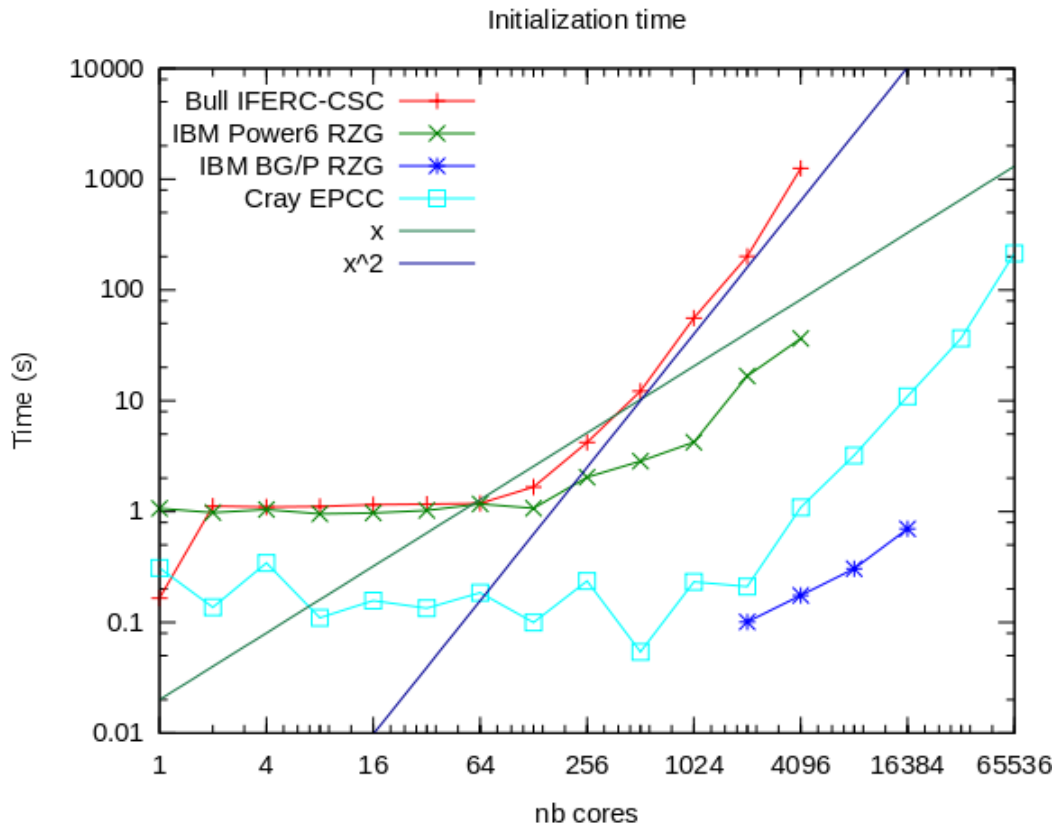


Fig. 18 Initialization time comparison for different HPC architectures.

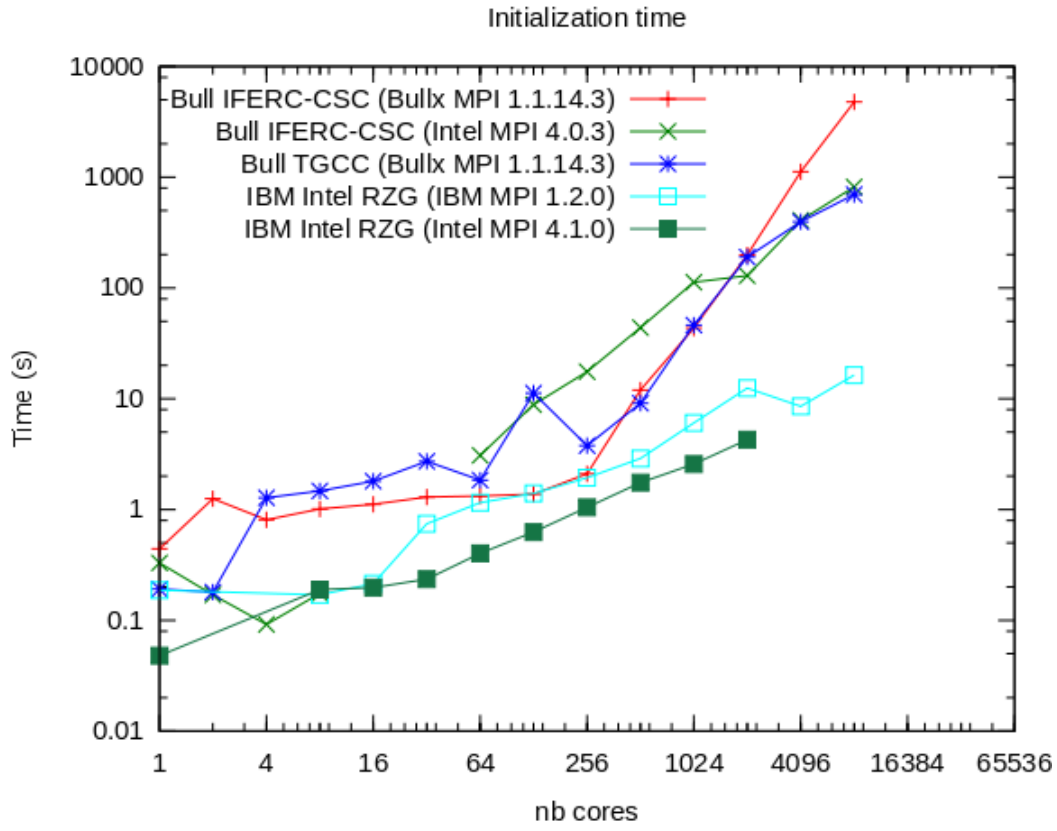


Fig. 19 Initialization time comparison for architectures based on Intel processors and Infiniband interconnect.

4.8. Future work

Two points are still pending for this project. Firstly, it is planned to run the ALL_TO_ALL benchmark on the IBM BlueGene Q machine at JSC and the Intel based IBM machine at LRZ in order to get a clear picture of the initialization time issue up to 64K cores on those architectures. Secondly, we still have to finish an efficient implementation of our MPI OpenMP and MPI shared memory segment parallelization strategies in order to improve their execution time.

5. Final report on HLST project MGTRI

5.1. Introduction

The purpose of the project is to aid on the task of 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. Presently GKMHD is not parallelized. Hence, a major building block of making the code parallel is to implement a parallelized multigrid solver on a triangular grid which is the topic of this report.

The multigrid method is a well-known, fast and efficient algorithm to solve many classes of problems including the linear and nonlinear elliptic, parabolic, hyperbolic equations like Navier-Stokes equations and Magnetohydrodynamics (MHD). Although the multigrid method is hard to implement, researchers in many areas think of it as an essential algorithm and apply it to their codes because its complexity is just *N log N*. The number of operations of the multigrid method depends on the number of degrees of freedom (DoF) times the number of levels (*log* of the DoF).

The multigrid method on triangular meshes has been studied by many researchers and has reached a mature state. However, the implementations were focused mostly on unstructured grids and accordingly very complicated data structures. Instead, the problem under consideration is based on a structured grid. Hence, the data structure has to be adapted to our problem to get an optimal parallel performance. So, we have to determine how to distribute the triangular grid information on each core and how to implement the parallelized matrix vector multiplication, the smoothing operators and the intergrid transfer operators.

Due to the ratio between computation and communication costs, the larger the number of DoF per core is, the better are the scaling properties of an iterative method, like the multigrid method. For sufficiently large numbers of DoF per core, the scaling properties are good. Unfortunately, for very large numbers of cores the number of DoF per core typically falls down critically and the communication time on the coarser levels dominates the computational time. Hence, we are seeking other methods which would be more efficient than the multigrid method for small numbers of DoF per core.

A potential candidate is the domain decomposition method (DDM) which is intrinsically parallel and scalable on massively parallel computers for solving partial differential equations (PDEs). The DDM can be further classified into the so-called overlapping and non-overlapping methods. On the one hand, the overlapping method can be used as a preconditioner of the Krylov iterative method. On the other hand, the non-overlapping method is suited for problems which have discontinuous coefficients in the PDEs. The convergence rate of the DDM for reaching a certain error threshold shows the following behavior: for the one-level DDM it depends on the number of cores and for the two-level DDM it depends on the ratio between its coarse and fine mesh size.

In addition to the multigrid method we implement and assess the following domain decomposition algorithms: the two-level Schwarz (overlapping) method, the balanced

domain decomposition method with constraints (BDDC), and the dual-primal finite element tearing and interconnection (FETI-DP) method.

5.2. Model problem and discretization scheme

Let us consider the Poisson problem on a regular hexagonal domain with a Dirichlet boundary condition

$$\begin{cases} -\nabla \cdot \nabla u = f, & \text{in } \Omega, \\ u = 0, & \text{on } \partial\Omega. \end{cases}$$

As discretization schemes, we consider the finite element method and the finite volume method. To construct the finite element method, we impose regular triangular meshes on the regular hexagonal domain as in Fig. 20 (solid lines, K) and define the linear finite element space by

$$V_J = \{v \in C^0(\Omega) : v|_K \text{ is linear for all } K \in \mathcal{T}_J\}.$$

Thus, the finite element discretization problem becomes;

$$\text{Find } u \in V_J \text{ such that, for any test function } v \in V_J,$$

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx.$$

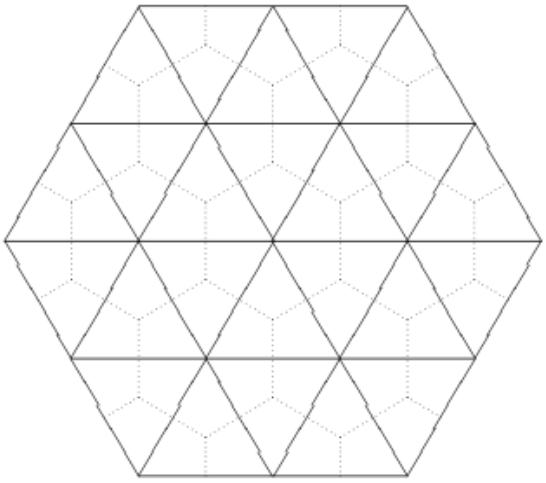


Fig. 20 The primal triangulation (solid lines) and control volumes (dotted lines).

To define the finite volume method, we construct the control volumes of the regular triangular meshes as in Fig. 20 (dotted lines, K_P^*) and integrate the equation over each control volume. As a result the finite volume discretization problem becomes;

$$\text{Find } u_h \in V_J \text{ such that, for all } K_P^* \in \mathcal{T}_J^*,$$

$$-\int_{\partial K_P^* \setminus \partial K_P^* \cap \partial\Omega} \mathbb{A} \frac{\partial u_k}{\partial n} d\sigma = \int_{K_P^*} f dx.$$

For both discretization methods the system of equations takes the form

$$A_J u = f.$$

The generated matrices of the two different methods are identical and only the right-hand sides differ. Hence, for our purposes we do not have to distinguish between them.

5.3. Decomposition of the structured triangular grid

The parallelization of the structured triangular grid on a regular hexagonal domain has to be considered in detail. In contrast to an unstructured grid, the knowledge of the structured grid can be leveraged to minimize the storage requirement for the triangular grid data and to optimize the data communication on a distributed memory computer.

Except for the single core case, we divide the hexagonal domain in regular triangular subdomains and distribute each subdomain on a core. Hence, the number of cores being usable is constrained to the numbers 1, 6, 24, 96, 384, 1536, 6144, 24576…, $6 \cdot 4^n$ as e.g. can be seen in Fig. 21.

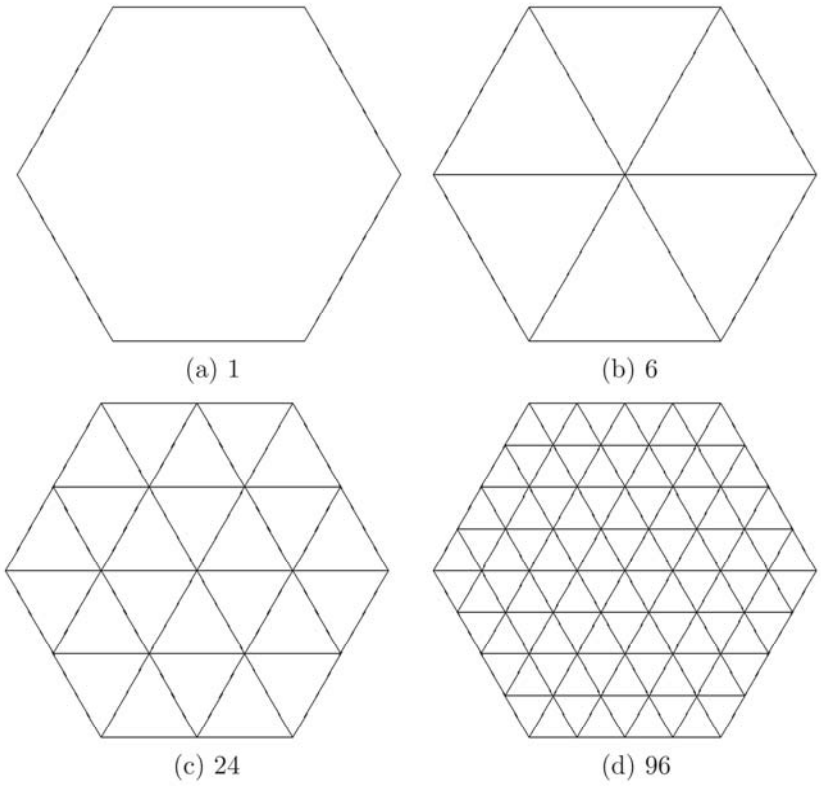
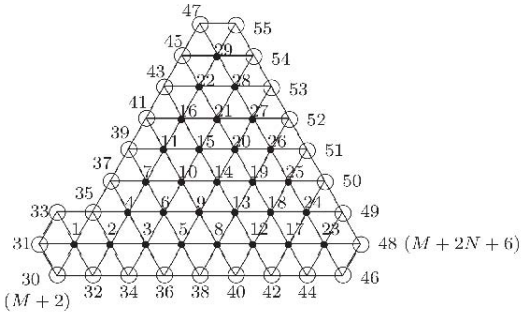


Fig. 21 The subdomains according to the number of cores: (a) one core, (b) six cores, (c) 24 cores and (d) 96 cores.

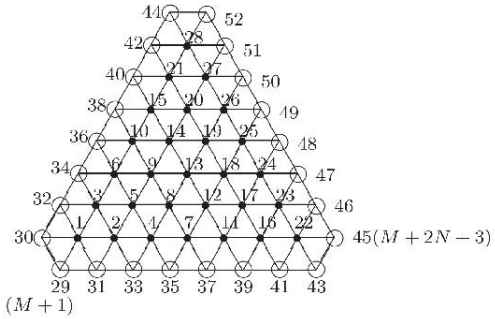
The so called real nodes are part of the subdomain which is assigned to a specific core. They are handled and updated by that core. In contrast, the ghost nodes belong to exterior subdomains located on other cores but whose values are needed for calculations done locally on the core. Hence, the values of the ghost nodes are first updated by the core to which they belong to as real nodes, and then their values are communicated to update the ghost node values.

The subdomains are classified into three types, according to their position as shown in Fig. 22. Here, we consider a Poisson problem with a Dirichlet boundary condition at the outer boundary. Nodes at the boundary of the whole domain can be handled as ghost nodes. Their values are determined by the boundary condition and thus do not have to be transferred from other cores.

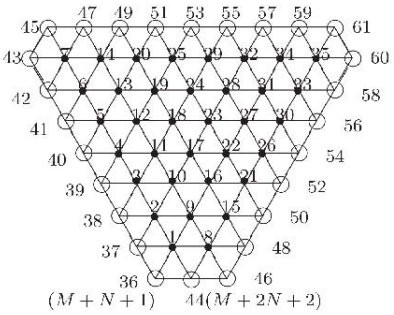
In each communication step, every core gets the value of the ghost nodes from the neighbouring cores. This process is divided into five sub-steps. It is the dominating overhead of the parallelization and thus a key issue for the performance of the code.



Type I: 0,6,9,12,15,18,21,24, ...



Type II: 1, 2, 3, 4, 5, 8, 11, ...



Type III: 7, 10, 13, 16, 19, 22, 25, ...

Fig. 22 The local numbering of the nodes on the three subdomain types being necessary to describe the inter subdomain communication pattern (bullet: real nodes, circle: ghost nodes, $N = 2^n-1$, and $M = N(N+1)/2$).

5.4. Scaling properties of the multigrid method

We choose the V-cycle multigrid method as a solver and as a preconditioner for the preconditioned conjugated gradient (PCG) method. For the multigrid method as a solver, we use the PCG method with a symmetric Gauss-Seidel preconditioner to achieve the solution at the lowest level and run two pre-smoothing and two post-smoothing iterations for all cases.

As a termination criterion for the iterative solvers we define a reduction of the initial residual error on the finest level by a factor 10^{-8}. In the past, we tested four different solvers on HPC-FF: the multigrid solver with Jacobi smoother (MGJA), the preconditioned conjugate gradient method with a multigrid preconditioner and Jacobi smoother (CGJA), the multigrid solver with Gauss-Seidel smoother (MGGS), the preconditioned conjugate gradient method with a multigrid preconditioner and Gauss-Seidel smoother (CGGS). Good weak and strong scaling properties were achieved with these solvers. In the following we will focus on the most efficient method which is the multigrid method as a solver or as a preconditioner with the Gauss-Seidel smoother. Our new studies have been conducted on the HELIOS machine at IFERC-CSC.

The lower the level of the multigrid iteration is, the more the communication overhead becomes dominant, and this has an impact on the scalability of the whole algorithm.

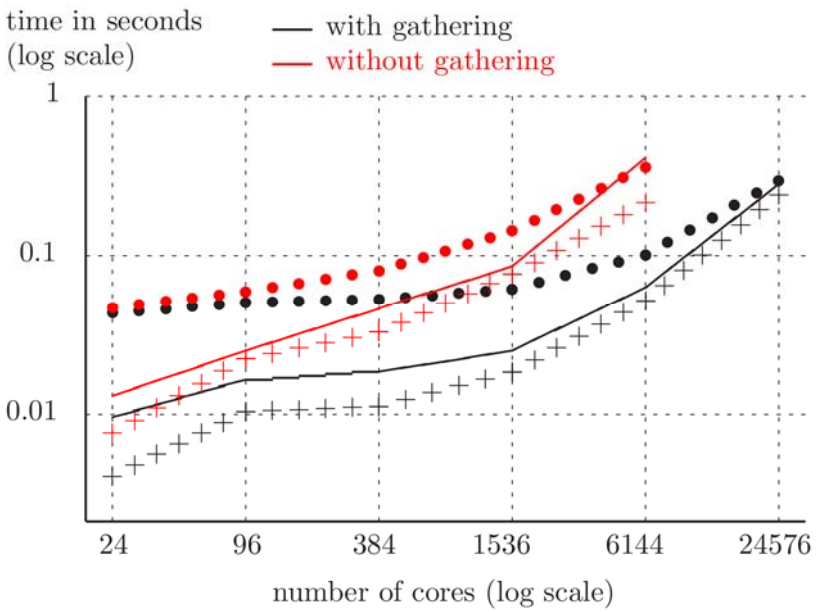


Fig. 23 The solution times in seconds of the multigrid method as a solver with and without gathering data on a certain level as a function of the number of cores. Three different cases with a fixed number of DoF per core (semi-weak scaling) are depicted [2K DoF (+), 8K DoF (solid line), and 24K DoF (bullet)].

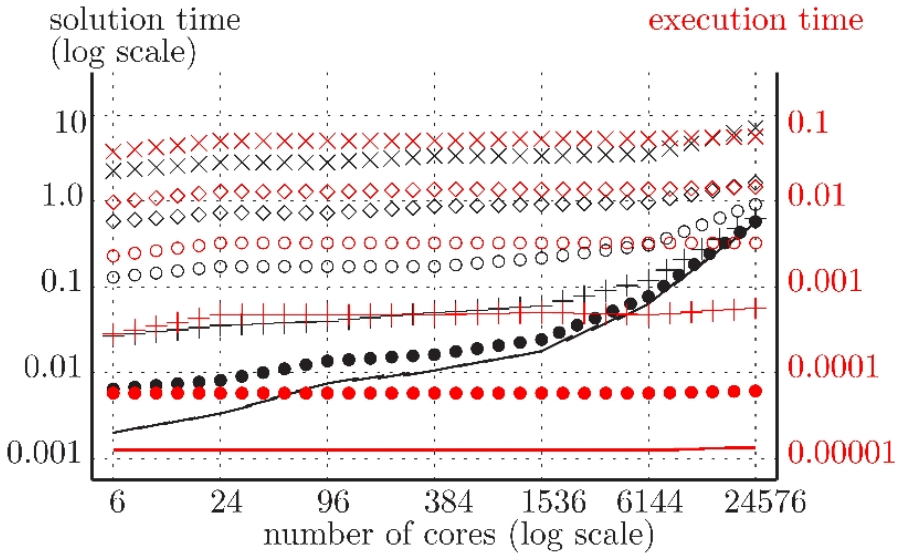


Fig. 24 The solution times in seconds for the multigrid solver (black) and the hundredfold matrix-vector multiplication (red) as a function of the number of cores. Five different cases with a fixed number of DoF per core (semi-weak scaling) are depicted [2.2K DoF (solid line), 8.5K DoF (bullet), 33.4K DoF (plus), 132K DoF (circle), 527M DoF (diamond), and 2.1M DoF(times)].

The efficiency in solving the lower levels is of key importance for getting a scalable solver for massively parallel computers. Even if every core handles only one degree of freedom (DoF) at the lowest level, the total size of the lowest level problem can be still quite large especially when we reach the range of 10,000 cores. As a consequence, the solution of the lowest level problem takes too much time. To overcome this obstacle, all data are gathered at a certain level with an all-reduce operation on each core. Afterwards the multigrid solving is continued locally by descending and ascending the lower multigrid levels on each core redundantly. In Fig. 23 we compare the solving times for two different cases: with and without gathering the data on a certain level. In addition, three difference cases with a fixed

number of DoF per core (semi-weak scaling) on the highest level are chosen. It clearly shows that the multigrid solver with gathering is much faster than without. This effect becomes more pronounced for larger numbers of cores.

In Fig. 24, we compare the semi-weak scaling property of the multigrid algorithm (with gathering) to the matrix-vector multiplication, which is the kernel routine of every iterative method as e.g. the multigrid method. For better comparison we show the solution time of the multigrid solver and the hundredfold execution time of the matrix-vector multiplication. We select five different cases with different DoF per core. The results have been obtained on the HELIOS machine for different number of cores. It can be clearly seen that the matrix-vector multiplication has an almost perfect weak scaling property. Also, the multigrid solver has a very good semi-weak scaling property up to 24,576 cores for cases with at least 128 K DoF per core. But, for cases with a smaller number of DoF the multigrid solver shows a clear degradation for more than 1536 cores.

5.5. Domain decomposition method

The domain decomposition method (DDM) was originally developed to solve large problems by decomposing them into many smaller ones. As such, the method is intrinsically parallel.

First, there is the overlapping domain decomposition (Schwarz) method in which each core handles one of the subdomains shown in Fig. 25. The convergence rate for the one-level DDM depends on the number of subdomains (number of cores) and/or the width of their overlapping regions. In contrast to this, the convergence rate of the two-level DDM does not depend on the number of subdomains, and this is a desirable property when it comes to the application on massively parallel computers. It depends instead on the ratio of the coarsest mesh size to the width of the overlapping regions. The larger the width of overlapping is the larger the costs of data communication become. Hence, we focus on algorithms with minimal overlapping which are identical to the block Jacobi iteration.

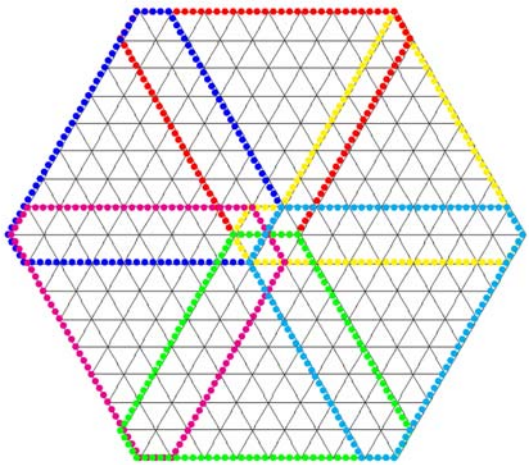


Fig. 25 The overlapping subdomains with a minimal overlapping.

Second, there is the non-overlapping domain decomposition method which was developed to solve PDEs with discontinuous coefficients as they appear in many material composition problems. Such problems are split into two steps: one to solve the local problem on each subdomain, which does not overlap with any other subdomain, as shown in Fig. 26, and the other to solve a modified local problem with imposed continuity at the boundaries of the sub-domain. 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 balanced domain decomposition (BDD) method belongs to the Neumann-Neumann

DDM and the finite element tearing and interconnection (FETI) method is a Dirichlet-Dirichlet DDM.

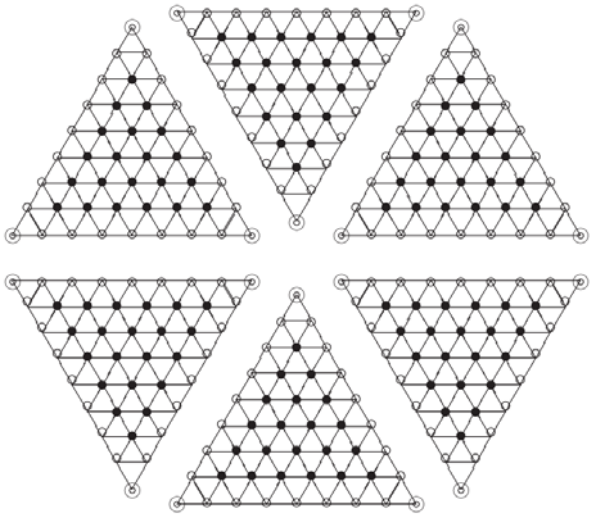


Fig. 26 The non-overlapping subdomains.

The BDD and the FETI methods are one-level DDM and have some difficulties to solve the symmetric semi-positive definite problems on each subdomain. These difficulties have been reduced by adopting both the balanced domain decomposition method with constraints (BDDC), which stems from the BDD method, and the FETI-DP (Dual-Primal) method, which stems from the FETI method. These are both two-level DDM and have to solve a global coarse problem in addition. They have a very good convergence rate which does not depend on the number of subdomains but rather on the ratio between the fine and the coarse mesh size. The more these two meshes spread apart in resolution, the larger is the number of iterations which becomes necessary for convergence.

We implemented the two-level Schwarz, FETI-DP, and BDDC methods. The required number of iterations for the two-level Schwarz method increases significantly when the overlapping region decreases, or the ratio of the mesh size of the fine level to the coarse level becomes small. In such a case the method does not scale nicely and becomes inefficient. Hence, we decided to consider only the FETI-DP and BDDC methods.

h/H	1/8		1/16		1/32		1/64		1/128	
# cores	FETIDP	BDDC	FETIDP	BDDC	FETIDP	BDDC	FETIDP	BDDC	FETIDP	BDDC
24	12	7	14	8	16	9	18	10	20	12
96	15	8	17	9	20	11	23	13	26	14
384	16	8	19	10	22	11	24	13	28	14
1536	16	8	20	10	23	11	26	13	29	14
6144	16	8	19	10	23	11	26	13	30	14
24576	16	8	19	9	23	11	26	13	29	14

Table 2 The required number of iterations of the FETI-DP (FDP) and BDDC (BD) method according to the ratio (h/H) of the mesh size of the fine level (h) and the coarse level (H).

We list the required number of iterations of the FETI-DP and DBBC methods in Table 2. It shows that the number of iterations depends on the ratio of the mesh size of the fine level to that of the coarse level, as for the Schwarz method, except that it does not increase rapidly.

We use the same gathering algorithm as in the multigrid method to solve the global coarse level problem. In both FETI-DP and BDDC, every sub-domain has some

contributions to the matrices and vectors on the coarser level and uses the solution of the coarse level problem. To solve the local and global problems, we use two direct methods and an iterative one; the LAPACK (Intel MKL) library routines for the dense matrix format, the IBM WSMP library with sparse matrix format, and the multigrid method.

We tested five different cases with fixed number of DoF per core, from 32 DoF to 8K DoF, and depicted the results in Fig. 27. For all methods the weak scaling property improves as the number of DoF per core is increased. However, the FETI-DP method is faster than the BDDC method even though the BDDC method requires a smaller number of iterations as shown in Table 2.

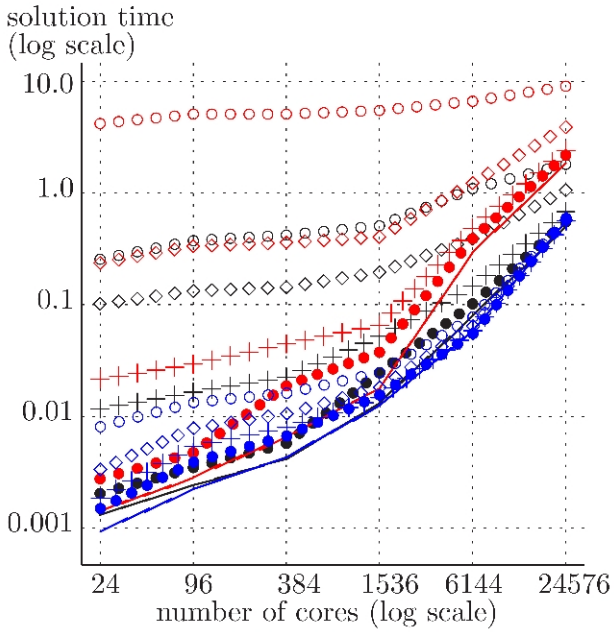


Fig. 27 The solution times in seconds of the multigrid solver (in blue), FETI-DP (in black), and BDDC (in red) as a function of the number of cores. Five different cases with a different fixed number of DoF per core are depicted: 32 DoF (solid line), 125 DoF (bullet), 500 DoF (plus), 2K DoF (diamond), and 8K DoF(circle).

For the smallest number of DoF per core (32 DoF per core) under consideration the solution times of the FETI-DP and the multigrid method are almost the same. For all other cases the multigrid method with gathering data is faster than FETI-DP method. The difference in performance of the two methods increases as the number of DoF per core is increased.

5.6. Conclusions

We have ported all our programs to the new HELIOS machine and are able to run our test cases on a regular hexagonal domain with regular triangulations up to 24,576 cores. As a first step, an efficient multigrid solver has been implemented. Scaling tests show that it has a very good semi-weak scaling property up to 24,576 cores for the larger test cases. Furthermore, we implemented three different domain decomposition methods: the two-level Schwarz method, the FETI-DP method and BDDC method. The two-level Schwarz method showed to be inefficient for our test problem. In contrast, the FETI-DP and BDDC methods were feasible. In all test cases under investigation the FETI-DP method has been found to be better performing than the BDDC method. The comparison between the FETI-DP method and the multigrid method showed a more complex picture. It is only for the smallest number of DoF per core (32 DoF per core) that both of the methods show similar performance. For all other cases with larger numbers of DoF per core our implementation of the multigrid

method with gathering of the data is the fastest solver for our problem. For small numbers of DoF per core we expected that the FETI-DP method would perform better than the multigrid method. However, for the problem under consideration this is not the case. This might be different for a hybrid parallelization concept of OpenMP and MPI.

6. Report on HLST project KSOL2D-2

6.1. 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 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 and so on. 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 regarding reaching a good scaling (~100 cores) property for the complex SOL geometry (a hollow rectangular with > 10^7 cells). Although being an efficient all-purpose Poisson solver, the project coordinator found out that further speed up is required for many BIT2 applications. Namely, the minimum time consumption for solutions in a realistic geometry is above 0.1 sec, which is still too long for applications requiring at least 10^7 time steps.

As further improvement it is planned to use 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 of 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 completely independently. The update due to the boundary conditions requires only the exchange of small-size messages between the processors. As a result, the scalability of the solver should increase significantly.

Among the many domain decomposition methods, the FETI-DP and 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 of a hexagonal domain. Even if these two methods turn out to be slower than the multigrid method for SOL geometry, such an approach might still give us valuable experience.

6.2. Current status and future work

The current implementation of the multigrid method in BIT2 suffers from excessive communication costs. The lower the level of the multigrid iteration is, the more the communication overhead becomes dominant, and this has an impact on the scalability of the whole algorithm. The efficiency in solving the lower levels is of key importance for getting a scalable solver for massively parallel computers. Even if every core handles only one degree of freedom (DoF) at the lowest level, the total size of the lowest level problem can be still quite large especially when we reach the range of 10,000 cores. As a consequence, the solution of the lowest level problem

takes too much time. To overcome this obstacle, all data are 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. Work on this has started and should be finished soon.

In parallel we have been studying how to implement the two-level non-overlapping domain decomposition methods developed during the former project MGTRI. We plan to adapt and implement the FETI-DP and the BDDC methods for the SOL geometry. Finally, performance tests on the HELIOS machine at IFERC-CSC will show whether or not the multigrid method is superior to the FETI-DP and BDDC methods for this case.

7. Final Report on HLST project ITM-ADIOS

7.1. 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 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 primary goal of the present project, ITM-ADIOS, as stated by its coordinator, David Coster, is to investigate the adequacy of a publicly available software product, the "ADIOS" software library (see [1]), to obtain a better I/O efficiency in plasma physics simulations performed on the HPC-FF computer installation (hosted in Jülich, Germany). HPC-FF is made up of 1080 nodes 8 cores each, for a total of 8640 cores. The batch system allows a maximum of 4096 for a single parallel job. Additionally, we extended our study to the new HELIOS machine (hosted in Rokkasho, Japan). This machine is considerably larger: 4410 nodes with 16 cores each; here from a total of 70560 computing cores, even a maximum of 65536 are allowed to participate in a single parallel job.

This report explains the project activities, while the last section presents our concluding remarks, as well as recommendations to users and topics for eventual project extensions.

7.2. The LUSTRE file system and ADIOS: notation

Before proceeding, we introduce minimal and simplified LUSTRE terminology and notation necessary for this report.

As mentioned earlier, the HELIOS and HPC-FF machines are equipped with a LUSTRE type file system. The main features of LUSTRE are that it distributes data over multiple *storage targets* (Object Storage Targets – OSTs) hosting the disk arrays and that it serves files over a local network. A LUSTRE installation's OST count (L_{OSTs}) is an important parameter, and users interested in I/O performance shall be aware of it. Storage of a file on LUSTRE can span over multiple OSTs, by means of the striping technique. That is, blocks of a specified size (LUSTRE Stripe Size – L_{ss}) are cyclically distributed among a specified, contiguous set of OSTs. Such interval of OSTs can be associated to a file at creation time either explicitly, by specifying the index of the OST which will receive the first block (LUSTRE Stripe Offset – L_{so}), or requesting it to be chosen randomly (with L_{so} = -1). If we indicate the number of OSTs storing a file with the LUSTRE Stripe Count parameter (L_{sc}), then the last OST index will be $L_{so} + L_{sc}$-1. Throughout this report we will call the (L_{sc},L_{ss},L_{so}) parameters triplet "LUSTRE configuration point", or "LUSTRE striping semantics (of a file)". LUSTRE striping semantics can be applied to files at creation time, either via LUSTRE programs (e.g.: lfs), with LUSTRE library functions, or *MPI-IO hints*.

The ADIOS parallel I/O library was conceived to work well with file systems such as LUSTRE. A main feature of ADIOS is that it subdivides the total I/O of a given application in a specified number of files. For clarity, we will call these *subfiles*. If we indicate with P_{ntasks} the number of parallel (MPI) tasks of our application, with P_{dpt} the

amount of data to be written per task, and with A_{sf} the number of subfiles we ask ADIOS to use, each subfile will be of size $A_{fs} = (P_{ntasks}*P_{dpt}) / A_{sf}$. Most of this study is concerned with the optimal choice of the (A_{sf},L_{sc},L_{ss}) parameters to reach high throughput I/O with LUSTRE using ADIOS. We measure throughput by dividing the total size of the I/O, (*Mebibytes* or *Gibibytes* – SI units commonly albeit often incorrectly referred as Megabytes or Gigabytes – see [2]) by the *wall clock time* it takes (seconds). Throughout this report we will use "high performance" as synonym for "high throughput".

For reader's ease, we summarize here the main acronyms used in the document:

- OST : Object Storage Target (LUSTRE jargon)
- ADIOS : Adaptable IO System
- A_{sf} : number of subfiles written in a single parallel I/O performed with ADIOS
- A_{fs} : size of a subfile written in a single parallel I/O performed with ADIOS
- L_{OSTs} : LUSTRE OST count for a file system
- L_{ss} : LUSTRE stripe size for a file
- L_{sc} : LUSTRE stripe count for a file
- L_{so} : LUSTRE stripe offset for a file
- L_{sp} := (L_{ss}*L_{sc}) to A_{fs} ratio
- L_{mc} := (P_{ntasks}*P_{dpt}) to L_{ss} ratio
- P_{ntasks} : a Program's parallel Tasks number
- P_{dpt} : a Program's Data Per Task
- MPI: Message Passing Interface (for distributed memory parallel programming)
- OpenMP: Open Multiprocessing (interface for shared memory parallel programming)
- IAB: ITM-ADIOS project Benchmark (see Sec.7.5)
- IABC: IAB C benchmark (see Sec.7.5)
- XML: eXtensible Markup Language

7.3. The ADIOS library

ADIOS [3] is the outcome of a cross-institutional research effort, primarily led by the US-based Oak Ridge National Laboratory (ORNL) and the Georgia Institute of Technology (GIT). The scope of ADIOS is to make high performance I/O easy to obtain with scientific codes running on high-end supercomputers. The most appealing aspect of it is that it can be easily integrated in MPI-enabled parallel applications. ADIOS is especially suited to offer parallel I/O capabilities to MPI-parallel programs having only serial I/O, thus increasing the overall throughput in file system I/O operations. ADIOS is based on the MPI-IO API (MPI's I/O related Application Programming Interface – see [4]), although no MPI-IO knowledge is needed to use it. By default, ADIOS offers its own storage format; however, it can optionally read/write data files using different parallel file format libraries in a transparent manner.
A notable application of ADIOS (according to [3] and [5]) has been the handling of the parallel I/O of the "Gyrokinetic Toroidal Code" (GTC) from the Princeton Plasma Physics Laboratory, reportedly running on many thousands of processors.

Our first impressions with the library have been positive: the source code archives are distributed with a number of sample programs which exemplify the typical library usage. The user manual ([3], over 100 pages) covers the various aspects of ADIOS usage. Besides the detailed description of the API, it offers extensive commentary on the example programs. In addition, some internals, as e.g. the special purpose "BP" binary file format ADIOS data is stored into and various optional "transport methods" are explained. However, we have found to be detrimental to the public user that not all of the functionality being described in the manual is publicly available (notably, the *staging* functionality). Of course, we expect an expansion, or at least more discussion

about these extra functionalities in the next releases of ADIOS and its documentation.

After having read the user manual, we have accomplished an installation of ADIOS on the HPC-FF cluster, under a normal user account. A base installation of ADIOS only requires two external libraries, namely the "Mini-XML" [6] library, and any standard MPI installation. On our request, the "Mini-XML" library has been installed by the system administrators in a system-wide location on both the HPC-FF and HELIOS machines; then it has been made usable with the "shell modules" system. Configuration and build phases of the ADIOS library itself did not pose any problems. As there are several options at installation level, a future reinstallation of the ADIOS library could become appropriate to achieve improved performance or enable additional features. After the installation, we have been successful in executing example and test programs distributed in the library sources archive.

A note about software versions; ADIOS releases considered by us were v1.3 at the project beginning (September 2011); a modified 1.3 version received via email from with the authors in October 2011; then the 1.3.1 version released in November 2011. A new release v1.4 was made available in July 2012, but due to its API incompatibilities and the end of the project approaching, we did not have chance to use it, so most of this report refers to version 1.3.1.

7.4. Collaboration with other parties

Over the course of the project, we have been in contact with ADIOS authors: a couple of times we provided them with a bug report and relative fix; at other times we asked for clarifications. In all cases we received good feedback, and we see this as favorable to an eventual adoption of ADIOS by the ITM. Interaction with the authors is especially valuable because of the rather restricted community of ADIOS users. Since the beginning of the BLIGHTHO project on IFERC HELIOS, we have been in contact with its support group, CSC. This has resulted in a fruitful collaboration in which we could give feedback about a wide spectrum of issues affecting users, and work on solving such problems. Questions we were involved were:

- Installation of ADIOS and its dependency the MXML library.
- During ADIOS usage, several I/O instabilities, e.g.: crashes occurring when using ADIOS with more than one node, i.e. switching from 16 to 17 MPI tasks.
- Bringing to the attention of system administrators that the serial I/O performance was unusually slow; later identified as a defect specific to the newest LUSTRE versions (HELIOS runs version 2.1).
- Collecting knowledge from documentation, experiments and other supercomputing centers experts about the consequences of *mounting* the LUSTRE file system with the 'flock' option resulted in adoption of it by the HELIOS administrators.
- In July 2012, severe striping related problems on the LUSTRE file system; here we contributed with the identification of individual faulty OST servers, and provided the system administrators with information and special purpose benchmark scripts.
- Miscellaneous documentation feedback.

For technically demanding questions, the corresponding tickets required a non-negligible amount of time to be processed, on both sides. In parallel with the

communication with CSC, we established direct communication with Bull (i.e. the vendor) engineers on site, especially when severe problems had to be dealt with. Regarding HPC-FF, prompt answers of the Jülich Supercomputing Centre support team allowed us to quickly identify the origin of crashes (either node failures or our responsibility) in IABC use on HPC-FF, and acted consequently. We did not encounter any instability on HPC-FF.

7.5. Development of an ADIOS benchmark

As our first ADIOS related activity, we began developing a flexible benchmark program (IAB) in order to get accustomed with the different ADIOS features available to us.

IAB evolved considerably: in the beginning we were using a pre-release ADIOS version received from the authors through e-mail communication; we adapted to the manual recommendations for the later ADIOS version. IAB consists of a "core" benchmark program (IABC) written in the compiled language C and a number of scripts. The scripts drive the core program through various experiments using the batch systems available on HPC-FF and HELIOS, collect performance data and post-process the results. We have chosen to implement the benchmark program core in the C language rather than the traditional numerical computations programming language FORTRAN for two main reasons. First, because the exposed ADIOS C API is slightly richer and more expressive than FORTRAN's one. Second, because it is less error prone to use C: the lack of available interface headers or modules for FORTRAN may lead to incorrect arguments to ADIOS routines being accepted by the compiler without warning, or other similar inconsistencies. As we write this report, we note that the newest ADIOS v1.4 solved that deficiency by building and installing the FORTRAN module files.

So far, we have been able to implement the following ADIOS features: single or multi dimensional global/local arrays, writing to/reading from a number of subfiles different than the number of MPI tasks, ADIOS buffer control and specification of program data with and without the XML-file interface. The "IAB" benchmark has been of fundamental importance to our investigation, as it has been adapted to simulate different ADIOS usage scenarios and collect performance statistics. We have been continuously modifying the benchmark as we progressed with our knowledge of ADIOS. Now that IAB is pretty mature, it can be used with little effort on HPC-FF/HELIOS in different scenarios.

7.6. Serial I/O on HELIOS and HPC-FF's LUSTRE file systems

Before proceeding with explicitly parallel I/O (ADIOS on MPI I/O) on the parallel file systems we have on HPC-FF and HELIOS, we recognized the necessity to understand the factors influencing I/O performance starting with a base case: that of using a strictly serial API from a serial (non-MPI, single-threaded) program.

HPC-FF's "$WORK" and HELIOS's "$WORKDIR" file systems are based on LUSTRE, so even I/O from a serial program should benefit from higher throughput due to LUSTRE's striping. Actually the choice of the file "striping semantics" influences I/O in a relevant way also for a serial program. In this context, we have developed or used a number of trivial serial programs using the strictly serial I/O API from within FORTRAN and C, as well as companion scripts for running these programs, collecting their performance data and producing summary graphs out of a series of runs. These programs are instructed to perform I/O of the same data load (in terms of both size and contents), but with a different approach regarding LUSTRE-independent parameters, as write block size, number of blocks, file system synchronization options, and (source code) language.

This type of experiment is necessary, as the semantics of basic I/O constructs in POSIX C (read(), write(), and related functions – see [7]) and FORTRAN languages differ; prominently, the FORTRAN runtime library buffers data before effective I/O, while POSIX C is unbuffered, and thus closer to the operating systems interface. We also do not consider the C standard (e.g.: ANSI C, or C99) I/O functions, which may be buffered.
By running these programs, we noticed that FORTRAN run time library's buffering mechanism masks considerably the system I/O calls, thus preventing us from exercising control over when to perform effective I/O. Its outcome (in terms of performance) can be either good or bad, depending on the particular case, but it generally mitigates extremes.

On the contrary, C's POSIX API I/O semantics does not specify buffering, and the effective I/O pattern influences performance considerably. One could investigate into methods for exercising more control over buffering and "block writing" in FORTRAN, be it using available language constructs, or non-standard compilers specific extensions. By studying these baseline cases, we already found that even writing a non-trivial but still small amount of data to disk (e.g.: a few hundreds of MiB), the gap between the fastest and the slowest approach can be even an order of magnitude. Our benchmark script performs different I/O operations by invoking the Unix "dd" program, and varying LUSTRE parameters only. We have chosen "dd" because it invokes POSIX C's I/O routines (and thus, the operating system's I/O calls) in a manner which can be precisely controlled. In addition, we run "dd" with a strict data synchronization option; thus, preventing the file system caching on the operating system side.

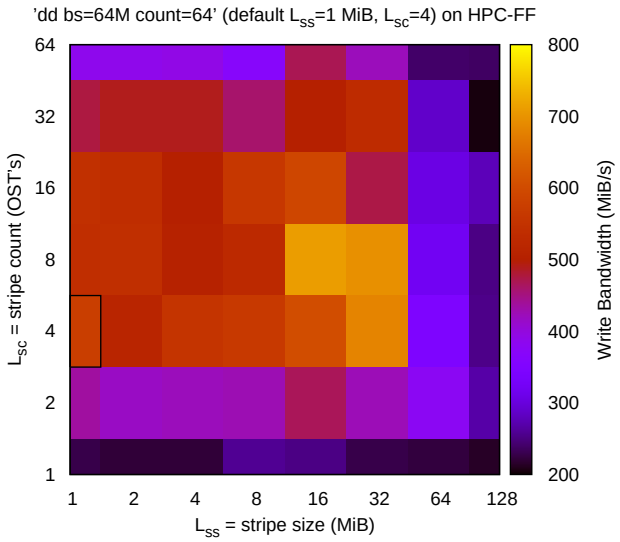


Fig. 28 Serial I/O throughput of a single file on HPC-FF. Notice the relatively large distance of the default configuration (L_{ss}=1 MiB, L_{sc}=4) from an optimal one (L_{ss}=16 MiB, L_{sc}=8) for a 4 GiB I/O.

HPC-FF's $WORK partition is equipped with L_{OSTs} = 120 and according to the system administrators, has a practical upper limit performance (under artificial conditions obtained during acceptance testing) around 21 GiB/s. See Fig. 28, for results on HPC-FF's $WORK.

With this experiment we have attained at most around 800 MiB/s on HPC-FF, and we regard this value as a reasonable best case when using serial I/O. Without the synchronization option, we have been able to measure around 1 GiB/s. We omit plots corresponding results obtained with the FORTRAN based I/O; they were generally less informative than the ones obtained with the dd-based script, mainly because of

the damping effect of software caches involved in the FORTRAN run time library. Please notice how the worst (lower throughput) results surround the best ones, and their difference: worst cases perform 200 MiB/s, i.e. one fourth of the best. We note also that the default system configuration (boxed in Fig. 28) yields circa 3/4 of the best results (circa 600 MiB/s). On lighter, general workloads the systems defaults are likely to be the best choice, though.

HELIOS is equipped with three LUSTRE file systems; of importance to our experiments is mainly $WORKDIR. $WORKDIR has $L_{OSTs} = 168$, and according to the system administrators, has a theoretical upper limit throughput of 76.8 GiB/s. On HELIOS (see Fig. 29) results ranged from about 300 to 550 MiB/s, and top results were inferior to those on HPC-FF; this can be surprising at a first glance.

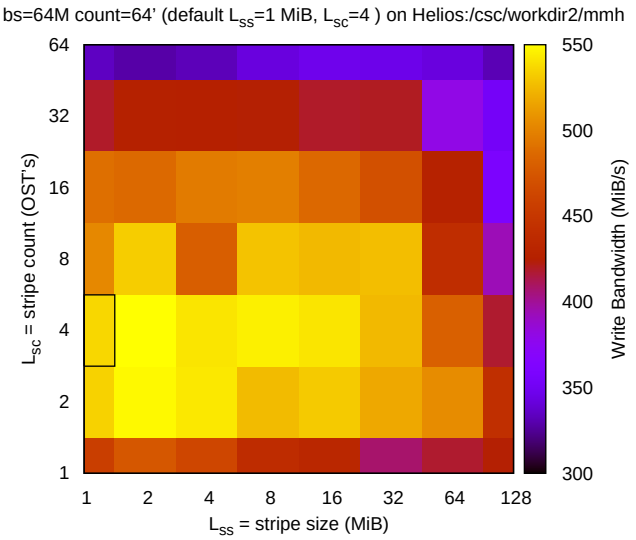


Fig. 29 Serial I/O throughput of a single file on HELIOS. Notice how here an optimal configuration (L_{ss}=2 MiB, L_{sc}=4) is very near to the default one for this 4 GiB I/O.

After some inquiry, it was found that this is a defect peculiar to the 2.1 version of LUSTRE (the one present on HELIOS). We are confident that the LUSTRE development group will fix this issue. HPC-FF runs the 1.8.4 version of LUSTRE, which does not suffer from this problem. Of interest in this measurement is also the lower end results (e.g.: for stripe count 1), slightly better than for HPC-FF. It is important to keep in mind that these results are susceptible to great variation, either from machine load or by the measurement being taken just after maintenance. It is interesting to compare values for these serial I/O runs with the best attainable via explicitly parallel I/O (that is, via a program with multiple writing processes); we will do so in forthcoming sections.

7.7. Parallel I/O on HPC-FF

At the beginning of this project, the HELIOS machine was not operating, so we performed our first LUSTRE-oriented studies on HPC-FF, which we describe first.

Our IABC benchmark proceeded by writing to different numbers of output files, namely $A_{sf} = 1,2,4,...,4096$; write time was measured each time. A total of 15 combinations of LUSTRE configuration pairs (L_{sc},L_{ss}) were explored, with and without synchronization (via the fsync() POSIX call). We were using $L_{so} = -1$, that is automatic stripe offset choice. For each LUSTRE configuration, ten samples have been collected and averaged; each MPI task contributed with $P_{dpt} = 64$ MiB. Each experiment was repeated two times (so, two independent runs and sample collections). We considered each L_{sc} value in (4,8,16,32,64) and each value of L_{ss} in (8,16,32) MiB. The maximal performance attained with these experiments was

around 14 GiB/s. The most noticeable trend in the performance curves (see Fig. 31) was that the ones for a higher L_{sc} reached earlier high values of throughput. This behaviour is explainable by the higher number of OSTs serving each output file: higher capacity available enabled a higher throughput. The second noticeable trend is due to L_{ss}: higher values were beneficial in reaching top performance earlier as well. In Fig. 30, one can see the effect of increasing L_{ss} with an already high L_{sc}.

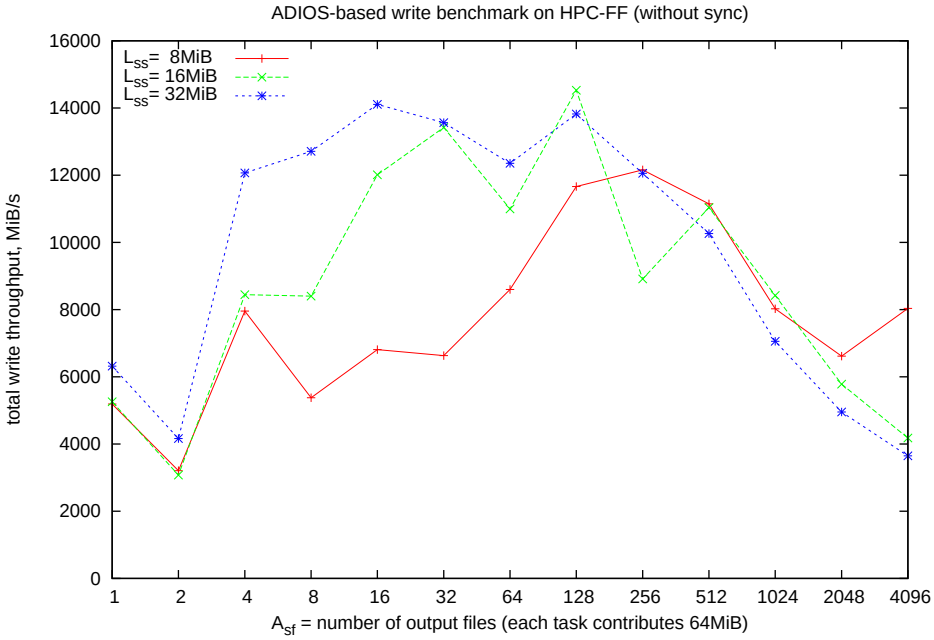


Fig. 30 Throughput with Parastation MPI on HPC-FF, 4096 tasks, $L_{sc} = 64$, different L_{ss}. Notice how configurations with larger stripes reach higher throughput first, but also loose it sooner, when increasing A_{sf}.

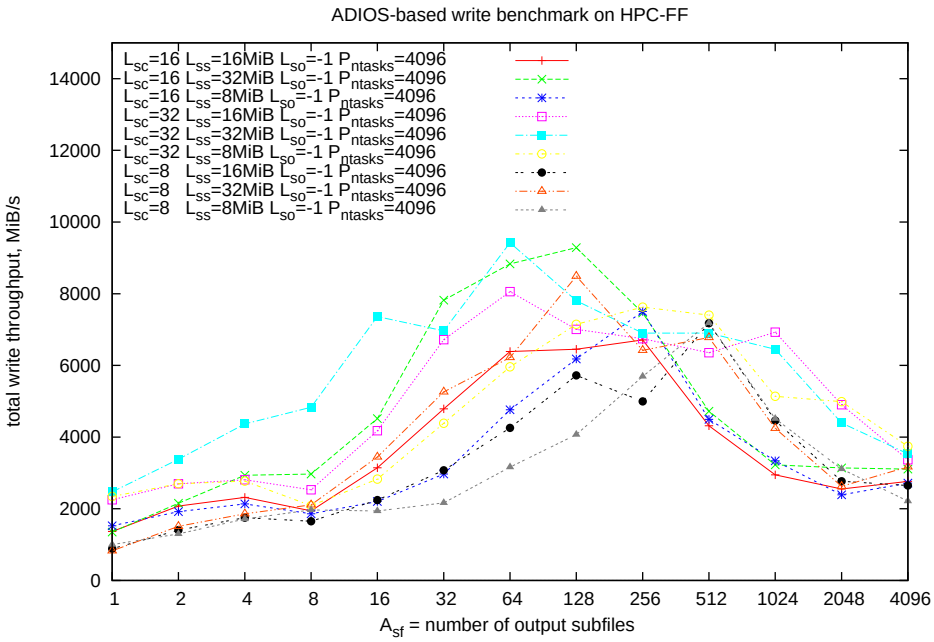


Fig. 31 Throughput with Intel MPI on HPC-FF, 4096 tasks. Notice how the larger stripe configurations reach throughput higher than the others.

However, high values of both L_{ss} and L_{sc} can also degrade performance. Increasing L_{sc} further gets more OSTs to participate in the I/O; in conjunction with high counts of output files, this may pose an excessive amount of communication directed at the I/O subsystem.

A more detailed discussion is in order. Let us discuss L_{ss} first. By increasing it, the average message size increases as well, and one can reduce the amount of communication overhead (fewer messages), while still taking advantage of the parallel file system. Indeed, the plot in Fig. 30 indicates the advantage of the L_{ss} = 32 MiB case over the lower-L_{ss} ones. However, another phenomenon is also noticeable: this curve is also the first one to decline when increasing A_{sf}, just after a maximum is reached. Indeed, distributing the same amount of data in more subfiles while keeping the striping parameters (L_{sc}, L_{ss}) fixed reduces the size of each of them (A_{fs}). At some point the size of a single file will not be large enough to send a data block to each of the L_{sc} OSTs. It is extremely likely that LUSTRE performs a pre-allocation of resources (e.g.: communication channel, disk blocks) on all of the L_{sc} OSTs assigned to a given file. So when $A_{fs} < L_{ss} * L_{sc}$, underutilization of resources degrades the performance. This explains why the degradation of many curves corresponding to higher $L_{ss} * L_{sc}$ is accelerated by increasing A_{sf}.

Having acquired a sufficient understanding of ADIOS+LUSTRE performance basics, we went into a second experimental campaign on HPC-FF, this time with a lighter load: half the samples per configuration point (5 instead of 10), and one run only per configuration (see Fig. 31). Furthermore, we limited the range of (L_{ss},L_{sc}) to the Cartesian product of (8,16,32) MiB and (8,16,32). However, we extended this study to both MPI implementations available on the system (Parastation and Intel), and considering all job sizes from 16 tasks upwards, exponentially. Up to 512 tasks, we used P_{dpt} = 128 MiB of data per task, in order to reduce the influence of noise on the measurements.

A comprehensive representation of all of these experiments (synchronization on, Intel MPI, P_{ntasks}=16,32,…,4096) is shown in Fig. 32. There, we rearrange the throughput data in a different way; namely, we align points on the abscissae according to the

$$L_{sp} := (L_{ss} * L_{sc}) / A_{fs}$$

metric. This representation of performance data suggests that for each given run size (so, for each P_{ntasks}), the best results are clustered in the middle of the L_{sp} range. For P_{ntasks} = 4096, such an optimum is located around $L_{sp} = 2^{-2}$. It is slightly higher for smaller task counts. The main benefit of this rearrangement of data is that it singles out lower throughput configurations: these are at the horizontal end-points of the plot. For this reason, we choose the L_{sp} metric for establishing a first "rule of thumb" for avoiding low throughput configurations; that is, avoiding the ones falling outside

$$2^{-5} <= L_{sp} <= 2^{2}.$$

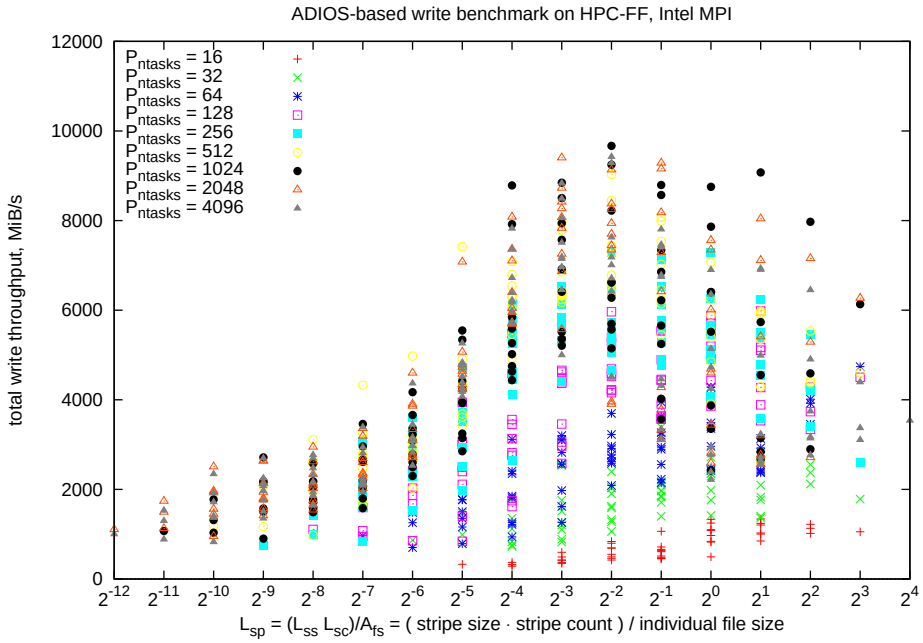


Fig. 32 Throughput with Intel MPI on HPC-FF, summary. Representation according to the L_{sp} metric allows us to single out many low performance configurations. Indeed, the best configurations are the ones between L_{sp} near to 2^{-2}.

We have some additional remark about these results. The top results in Fig. 32 are around 10 GiB/s; indeed we did not use any L_{sc} = 64 configuration here, and failed reaching the previous 14 GiB/s (see Fig. 30). Besides different ADIOS version/build flags, we suspect that a major role here is played by the fact we chose to run our experiments around December 31, 2011, with the machine very lightly loaded. Looking at the statistical fluctuation of results (we log each single I/O sample), we feel the approximate 10% drop in performance could be explained in this way. Moreover, eight months passed since that measurement, and by looking at the machine history (See [8]) we should be aware that minor software changes occurred, as well as at least one hardware reparation.

7.8. Parallel I/O on HELIOS

Concerning the execution of IABC on HELIOS, we employed an experimental setup similar to that used for the second benchmarking campaign on HPC-FF; that is 8,16,32 MiB for L_{ss}, and 8,16,32 for L_{sc}, and L_{so} = -1. Given the potential for larger jobs, we extended the measurement up to 8192 MPI tasks per run. We kept the size of the per-task data contribution P_{dpt} = 128 MiB up to 1024 tasks, then P_{dpt} = 64 MiB for larger runs. Our refrain from using larger quantities stemmed from the considerable impact on the machine / users, and because of the significant computing time budget necessary for completing the whole experiment. On the other hand, using 8192 tasks on HELIOS amounts to the same number of nodes as 4096 tasks on HPC-FF (8 MPI tasks per node on HPC-FF; 16 MPI tasks per node on HELIOS).

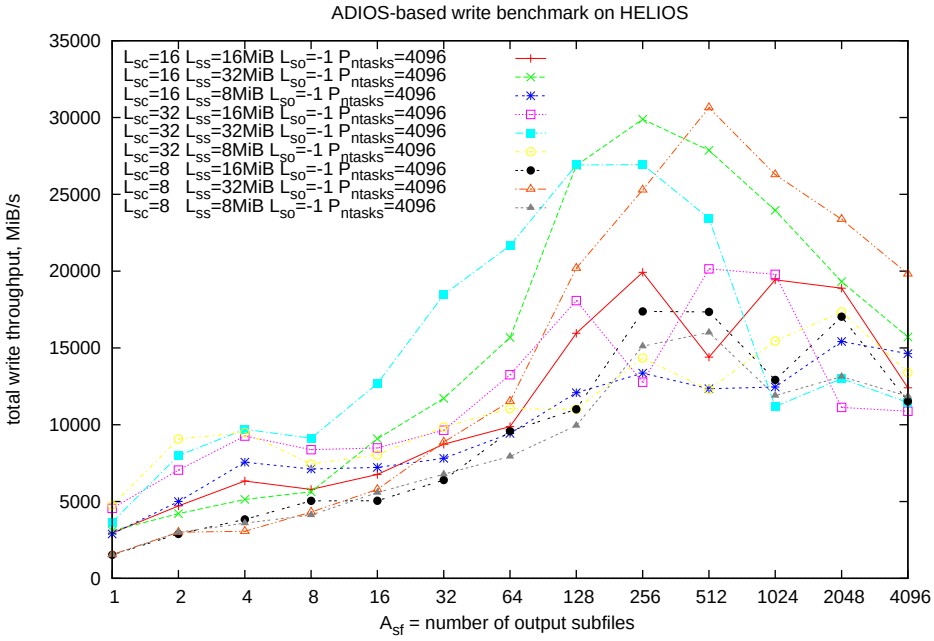


Fig. 33 Throughput with Intel MPI on HELIOS, 4096 tasks. It is significant to note how certain configurations reach just more than the top performance, regardless the value of A_{sf}.

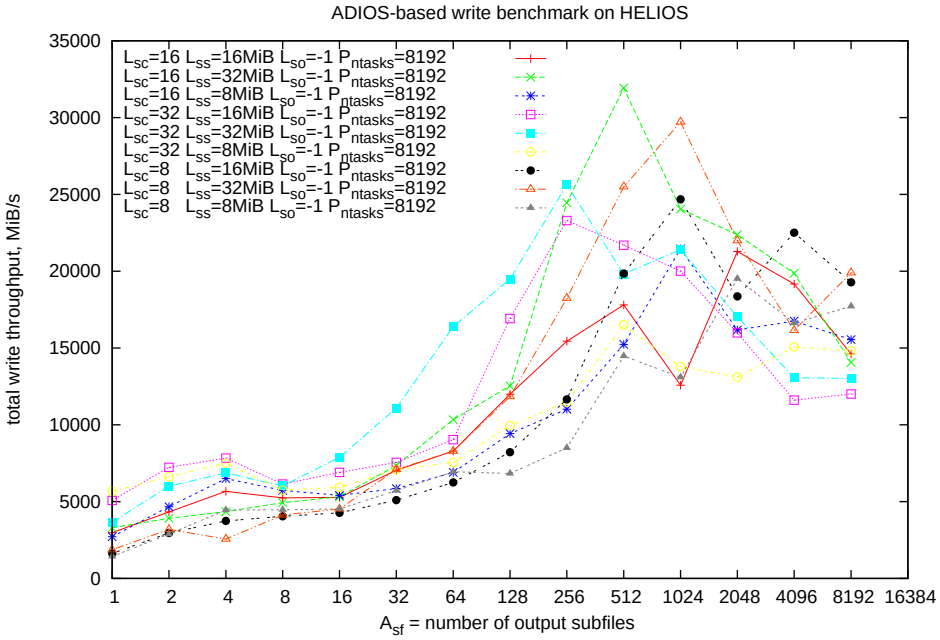


Fig. 34 Throughput with Intel MPI on HELIOS, 8192 tasks. These results are pretty similar to the P_{ntasks}=4096 case, thus indicating that system capacity is being approached.

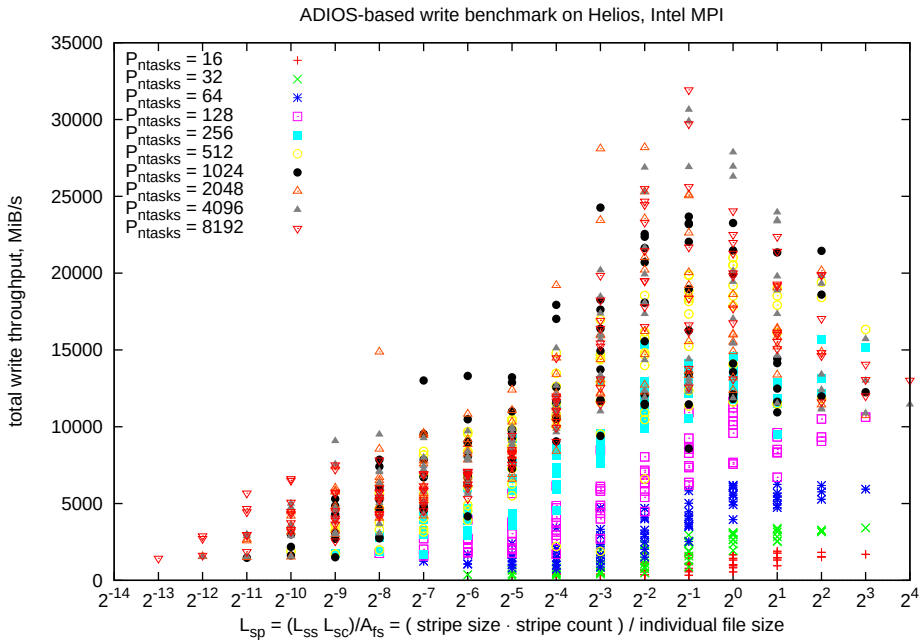


Fig. 35 Throughput with Intel MPI on HELIOS, summary. Notice how best results tend to cluster around a value of $L_{sp} \sim 2^{-1}$.

In absolute values, HELIOS throughput reached a peak of slightly over 30 GiB/s (see both Fig. 33 and Fig. 34). However, unlike HPC-FF, we see that not all the curves (e.g.: LUSTRE configuration points) reach this level, but rather only the ones with a higher (L_{sc}*L_{ss}) product do so. A large discrepancy originates already at A_{sf} = 1: there, throughput is just below 5 GiB/s for higher (L_{ss}*L_{sc}) cases, and just more than half of that for other cases. A similar proportion continues also for $1 << A_{sf} < P_{ntasks}$, where the fastest configurations are located.

In general, results on HELIOS follow similar trends to HPC-FF ones. For instance, higher L_{sc} cases anticipate the reach of both maximum performance and decline afterwards, with increasing A_{sf} (see Fig. 34). Similarly, looking at throughput over L_{sp} (see Fig. 35) confirms the phenomenon observed on HPC-FF, so that poorly behaving configurations are at the extremes of the L_{sp} range. That is, an excessive L_{sc}*L_{ss} to A_{sf} ratio is likely to cause unnecessary allocation of resources; an excessively small one reflects an excess in network / LUSTRE messages being necessary to complete the I/O. So if in the first case ($L_{sp} > 2^2$) it is likely that an under- or misuse of the resources occurs, on the other extreme ($L_{sp} < 2^{-4}$), latencies are likely to prevail because of the excessively fragmented communication. Even within the "good" range of L_{sp}, for each given job size there are large differences between the maximum and minimum results, thus indicating that additional criteria are needed for singling out bad cases.

We identify a secondary criteria as follows: the total number of messages necessary to complete the I/O (L_{mc} := (P_{ntasks}*P_{dpt}) / L_{ss}): we find that within each job size P_{ntasks}, the best cases are located in correspondence of the minimum of it. This suggests that increasing L_{ss} can have a good role. We did not develop our performance model further than this; we consider the conformance to both these criteria to be sufficient to reach fairly high throughput.

Another aspect of our interest relates to the relation between 4096 and 8192 tasks peak results. These are pretty close: respectively 32 and 30 GiB/s (see Fig. 34); indeed, summary results for all job sizes (see Fig. 35) seem to suggest that overall system capacity is being reached. However, while 4096 tasks use half of HPC-FF's system, 8192 is only about 1/8 part of HELIOS. So we feel that the top HELIOS

capacity should be somehow higher than what we reached here; however we would need some additional experiments to verify this.

7.9. Discussion and extra experiments

Supplementary to the results presented in the previous section, we considered a number of additional aspects that we discuss here.

First, the 64 MiB of (I/O) data per task that we used can be considered on the lower end in the range of the needs of most applications. A question may arise about how I/O performance may be affected by using significantly more than that. For our I/Os, in addition to ADIOS usage, we use synchronization, by means of a specific system routine (fsync()). This gives us guarantee that after the call, the system effectively writes data to the file system, instead of keeping them in operating system caches just after the write routine invocations. Time measurements up to before fsync() call confirm this: synchronization takes a relevant fraction of time, suggesting that effective I/O is being completed. When it comes to performing larger I/Os, no penalties are expected by us from the synchronization option; rather, improvements due to the sustained I/O. The limited number of experiments we performed confirms this difference, which showed to be in a range from negligible to circa +10%. Another interesting interaction to study may be that between large I/O sizes and memory usage in ADIOS. Although the main ADIOS buffer is under the user control and this should not pose a problem, we did not make extensive testing here, and we deem it to be worth of investigation.
Another aspect we considered was the impact of the MPI library on the performance. In our experience (see [9]), the impact of the adopted MPI library can be significant. HELIOS gives users options to use either the Bull MPI library (a version of OpenMPI customized by Bull), or the Intel MPI library: we tried both on the range of parameters presented in the previous sections. By inspecting results on HELIOS, we have the suggestion that with the Bull MPI library, the 8192 task peak case performs less than for 4096. With Intel's MPI, this does not happen, and the peak we observed was almost 20% higher than Bull's. On the HPC-FF machine, we found out Parastation MPI's throughput results to be slightly inferior to Intel MPI's, but still reaching similar top throughput.

During the development of IAB, we were in contact with ITM code authors Bruce Scott and Alberto Bottino, and obtained from them code snippets relevant to their respective codes (GEMZA and NEMORB) I/O part. We ported these code snippets into using ADIOS, and successfully performed some preliminary runs. We may use these prototypes for a further collaboration in fully integrating ADIOS in their applications.

7.10. Comparison of HELIOS to HPC-FF

In this report we present performance results of different ADIOS and LUSTRE configurations which are likely to be used by a real world application doing parallel I/O. We conducted our tests with the same parameters on both machines; therefore it is now possible and meaningful to get an estimate of the HELIOS to HPC-FF ratio in I/O throughput by comparing the results.

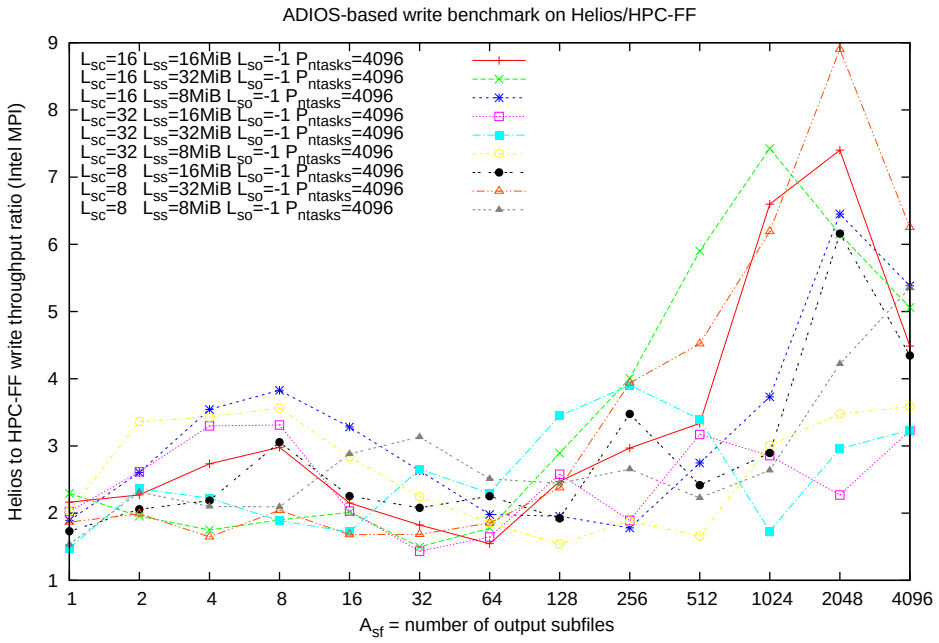


Fig. 36 HELIOS to HPC-FF throughput ratio, Intel MPI's. Here, for each configuration, ratios are reported. Note that the peak ratio value here does not coincide with the ratio of the peak configurations on the two machines. Rather, this peak is due to the strong degradation of performance on A_{sf} ~ 2048 on HPC-FF.

On Fig. 36 it is possible to see the throughput ratio of HELIOS to HPC-FF while using Intel's MPI, for different cases. We see that HELIOS performed always better, mostly by a factor of two or three. In a few cases the ratio grows to more than five (e.g.: at A_{sf} = 2048): these cases correspond to configurations where HELIOS is operating efficiently, while HPC-FF is not, and are therefore far less significant.

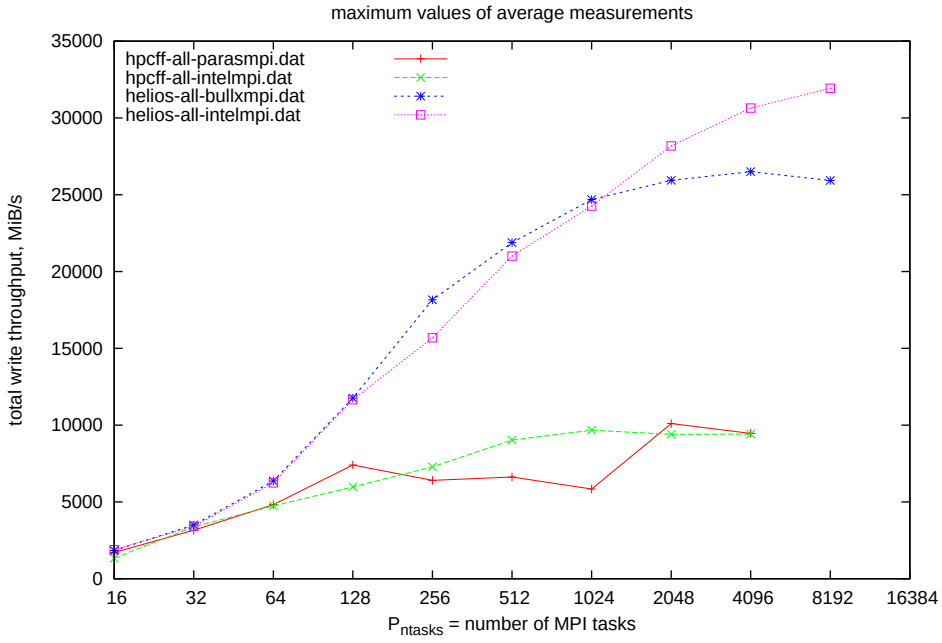


Fig. 37 HPC-FF's and HELIOS's attained peak, for different MPI's, whatever choice of L_{ss} or L_{sc}. This gives a good picture of the I/O capabilities of the two machines. Peak cases on the HELIOS machine can be three times as fast as on HPC-FF.

In Fig. 37 we show the best obtained results among all of the configurations mentioned in the preceding sections dedicated to HELIOS and HPC-FF individually. That is for each value of P_{ntasks}, we show a point corresponding to the maximum result among all of the considered combinations of L_{ss},L_{sc},A_{sf} parameters.

We notice that for 16 and 32 tasks there are no large differences between both machines (peaks at about 2 and 4 GiB/s); on $P_{ntasks} > 32$ HELIOS throughput begins surpassing HPC-FF's, doubling it around 128 tasks (12 vs 6 GiB/s). Then while HPC-FF's maximal observed throughput stabilizes after 512 tasks (around 10 GiB/s), HELIOS continues to improve (up to around 30 GiB/s). However, we observed a marked trend in which with Bull MPI performed worse than Intel's, starting from $P_{ntasks} = 2048$. Moreover, we noticed that Bull's MPI performs better on 4096 than on 8192 MPI tasks. Intel's MPI results exhibited a tendency to stabilize as well, although the curve seems to project a peak to more than 8192 tasks. Maximal distance between Intel and Bull MPI was at $P_{ntasks} = 8192$: circa 32 vs 26 GiB/s.

On the observed range, HELIOS can outperform HPC-FF by a factor of almost three. However, in the previous section we mentioned that in past experiments (and different parameters) we have been able to push HPC-FF's write performance up to around 14 GiB/s. This would suggest that a factor of two would be a better estimate. Nonetheless, since we think further experiments are necessary to determine the true HELIOS throughput peak, we estimate that this should be more than twice as on HPC-FF.

7.11. Conclusions

This project aimed at the assessment of parallel I/O techniques with ADIOS on machines of interest to the ITM community, i.e. HPC-FF and HELIOS. From the beginning, we found that precise control over the main ADIOS and LUSTRE file system parameters is of primary importance towards I/O performance. For this reason, we devoted most of our efforts into exploring the possibilities of both groups of parameters. We are confident that, to a good extent, these results can apply to other similar parallel I/O libraries (e.g.: SIONlib [10]).

Studies on HPC-FF allowed achieving around 14 GiB/s in writing; this is almost 75% of what the I/O system is capable of in ideal conditions, and we deem it to be unlikely to achieve considerably more under normal usage conditions. On the HELIOS machine, additionally to the performance characterization study, we did testing contributing towards proper machine operation. Although here we were able to achieve a top of circa 35 GiB/s (that is, circa 50% of the theoretical maximum) it might be possible to achieve even more.

Maximal ratio of parallel ADIOS to plain serial POSIX I/O can be quantified as exceeding 20x on HPC-FF (14 vs. 0.7 GiB/s), and more than 70x on HELIOS (35 vs. 0.5 GiB/s). Our study was conducted entirely under normal user accounts, in order to provide results under ordinary conditions. However, although being user oriented, we had to pay attention to a correct benchmark methodology. In order to minimize the effect of software caching and buffers, we were focused on results obtained by using synchronization. This allowed us to obtain measurements which are both effective and repeatable to a much better degree than without synchronization. Since most of the users normally do not use synchronization, their I/O performance results may apparently exceed ours; furthermore, 'peak' values may be distributed differently: therefore our results should be regarded as lower bounds in this sense. Being aware of this, we would recommend to use synchronization primitives for critical data, e.g.: output of a long/costly simulation; this is a further guarantee against data corruption (crashes because of node failures are common). We summarize and discuss our recommendations for users in the following list.

- Because of its relative simplicity, many users often adopt only a basic setup of MPI-I/O routines, by using either $A_{sf} = 1$ (a single global file) or $A_{sf} = P_{ntasks}$ (one subfile per task). For significant size I/O cases we experimented with, best effective performance was never located on these extremes: with $A_{sf} = 1$, ratio to pure serial I/O never exceeded circa 8x, likely because of a network bottleneck. By correlating LUSTRE parameters and I/O size to performance, we observed that keeping an application flexible on A_{sf} is a prerequisite for achieving high performance. In favour of ease of programming, this is an argument for using a library like ADIOS.

- Long term storage of ADIOS BP files is not recommended: ADIOS is relatively young software and this file format is likely to evolve. Therefore, if long term data storage is foreseen, one may employ a different data format (e.g.: one of the popular scientific data description ones) for archiving first, and convert it to the current BP version when needed, with a special purpose converter (which would be quite easy to write and maintain). In the presence of a large population of users, it would make sense to maintain an archive of files in the BP format, and convert them to the newest version whenever is available, assuming that the users will update their applications to the newest version of ADIOS as well.

- If I/O were the only criteria involved in the choice of a number of parallel tasks, a serial job would be more effective in terms of accounted CPU time: I/O scaling is inferior to the investment in CPU time budget. Indeed, our recommendation regarding a hypothetic format converter (e.g.: between ADIOS and a custom format) is to write a serial one. The ADIOS v1.4 manual (see [3] Sec. 2.5.1) documents support for compiling MPI programs and linking with ADIOS by using a custom provided MPI stub library; this should ease the development of a format converter. However, the rules governing the parallel tasks number choice in an application are different, so for a parallel application we recommend switching to parallel I/O in the case its serial I/O consumes a relevant portion of total time. With future hardware, such a fraction is deemed to increase.

- In our study, we identified a quantity ($L_{sp} := (L_{ss} * L_{sc}) / A_{fs}$), computed from the I/O configuration, which correlated in a sound way with throughput. Although our understanding on this subject is not fully complete, it is clear that this metric is readily usable for a ‘rule of thumb’ to single out configuration points where throughput will be low for sure. Additionally, one should avoid writing more files (A_{sf}) than nodes ($P_{ntasks}/8$ on HPC-FF, $P_{ntasks}/16$ on HELIOS), since this could be detrimental to throughput, as results have shown.

- In our study, we did not cover I/O of large numbers of scalar variables (as e.g. in FORTRAN namelist constructs) in practical applications it is unlikely that their volume may pose significant I/O problems. ADIOS supports scalar variables, so it is easy to write/read them together with the large numerical data. However, we find it acceptable to write scalar values in a separate small file: this would not impact performance in a noticeable way, as long as this file's content is read once in a traditional way (FORTRAN I/O, POSIX, C I/O ...) and its contents broadcast to all processes. Of course, a user should be aware of the portability consequences of such a choice. Ease of exchange of data may be coupled to the need for archival storage.

7.12. Future work

In this report's experiments, we found many advantages in using parallel I/O with ADIOS, and addressed the primary aspects of interest. However, interfacing to a

parallel I/O library may introduce some additional questions. There are many directions that could be investigated:

- Extend and refine the “rule of thumb”: we may refine the performance model in order to obtain an algorithm for the suggestions of good I/O configurations, given some I/O job requirements.

- Hybrid programming model: It would be interesting to refine our characterization for a hybrid OpenMP + MPI programming model.

- Task grouping/affinity: effects of regrouping the MPI tasks for ADIOS communicators.

- Impact with strongly asymmetric data quantities/LUSTRE striping.

- Usage limits: investigate maximal/minimal ADIOS memory consumption (buffers), and how this impacts on performance.

- Investigate the ease of reading checkpoint data generated with a number of MPI tasks different than during write (e.g.: more files than tasks).

- Comparison to some other similar libraries (e.g.: SIONlib [10], developed at FZ Jülich).

7.13. References

[1] *ADIOS project home page* ; http://www.olcf.ornl.gov/center-projects/adios/
[2] NIST Reference on Constants, *Units, and Uncertainty; International System of Units (SI) Prefixes for binary multiples*; http://physics.nist.gov/cuu/Units/binary.html
[3] *ADIOS 1.4.0 User Manual*; http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.4.0.pdf
[4] *MPI-I/O documentation web site; introduction*; http://www.mpi-forum.org/docs/mpi22-report/node262.htm
[5] J.Cummings, J.Lofstead, K.Schwan, A.Sim, A.Shoshani, C.Docan, M.Parashar; *EFFIS: an End-to-end Framework for Fusion Integrated Simulation*; http://info.ornl.gov/sites/publications/files/Pub24705.pdf
[6] *Mini-XML library*; http://www.minixml.org/
[7] The Open Group *Base Specifications Issue 7 (IEEE Std 1003.1-2008)*; http://pubs.opengroup.org/onlinepubs/9699919799/
[8] *HPC-FF* ; http://www.fz-Jülich.de/ias/jsc/EN/Expertise/Supercomputers/JUROPA/JUROPA_node.html
[9] Matthieu Haefele; *Comparison of different Methods for Performing Parallel I/O*; http://edoc.mpg.de/get.epl?fid=72188&did=498606&ver=0
[10] *SIONlib library home page* http://www2.fz-Jülich.de/jsc/sionlib/

8. Report on HLST project PARFS

8.1. Introduction

The Frascati theory and modeling group has developed, in the past years, PIC codes (HMGC, HYMAGYC) to study linear and nonlinear dynamics of Alfénic 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 project is to get 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 numbers 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 face the investigation of relevant mode numbers ($n \approx 40$) for this class of modes in ITER like configurations.

8.2. Initial activities

Given the limited project time, in agreement with the project coordinator it has been decided 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 steps were activities of "standardization"; namely improving its portability characteristics (deployment on the HELIOS machine is foreseen) and enhancing adherence to modern Fortran standards by replacing obsolete constructs with current ones.

Identification of the main problem (distributed the field solver) has also begun. The current, built-in solver algorithm has not been found to be apt to parallelization. Therefore, we proceeded towards testing third party solver packages for a test problem (obtained via 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. Although useful and relatively easy, it does not fit into the time budget of the current project. It would involve some major architectural change decisions that only the developers can make. On the other hand, the linear solver assessment activity is more specialized, as well as self-contained, so we will focus our efforts in this direction. As a consequence the parallelization of MARST itself is left over to its developers, or as a separate future HLST project.

8.3. Further activities

Once the appropriate parallel solver packages are identified, we plan to verify their scaling properties and memory consumption. Finally, the best suited solver will be interfaced to HYMAGYC, giving the user the option for choosing between the parallel and original serial solver.

9. Final report on HLST project NEMOFFT

9.1. Introduction

The NEMOFFT project aims at removing, or at least alleviating, a well-known parallel scalability bottleneck of the global gyrokinetic particle-in-cell (PIC) code ORB5, in its current electromagnetic version NEMORB. This code has high demands on HPC resources, especially for large-sized simulations. At each iteration, the particle charges are deposited on a spatial three-dimensional (3D) grid, which represents the source term in a Poisson equation (the right-hand side). Due to strong spatial anisotropy in a tokamak, only a restricted set of degrees of freedom, or modes, are allowed to exist, namely, the ones which are either aligned or close to being aligned with the background guiding magnetic field. Using this physical restriction on the Poisson solver not only reduces the required floating point operations (solve only for the allowed modes) but also improves the numerical signal to noise ratio, which is of central importance in a PIC code. This implies calculating two-dimensional (2D) Fourier transforms to apply the corresponding filters, but because the mesh domain of NEMORB is distributed across several cores, it requires large amounts of grid data to be transposed across cores. Such inter-core communication naturally impairs the code's parallel scalability, and in practice renders ITER-sized plasma simulations unfeasible.

In principle, there are two possible ways to deal with this issue. Either to change the spatial grid to be globally aligned with the equilibrium magnetic field, which eliminates the need for Fourier filtering, or to improve the way the parallel data transpose and the Fourier transforms are done. While the former would completely eliminate the aforementioned scalability bottleneck, it is not transparent how such an approach could be implemented in the finite element basis used in NEMORB. It would certainly imply fundamental changes to the original code. Conversely, the latter, even though potentially less effective, can be implemented as an external library, with minimal changes to the original code required. Furthermore, since there are several other codes of the EU fusion program that share the same parallel numerical kernel with NEMORB (e.g. EUTERPE), they would also directly benefit from it. Therefore, this project focuses exclusively on the second approach.

The bulk of the work on improving NEMORB's 2D Fourier transform algorithm can be divided into two main parts. Namely, the exploitation of the Hermitian redundancy inherent to NEMORB's purely real input data and the optimization of the distributed transpose algorithm. Both together aim at speedup factors of the order of two. The corresponding steps made to achieve this improvement, as well as the performance measurements made on different HPC machines are detailed in the remaining sections of this report, which finalizes the project NEMOFFT.

9.2. NEMORB's 2D Fourier transform algorithm

9.2.1. Domain decomposition

NEMORB's 3D spatial grid comprises the radial, poloidal and toroidal directions, discretized with N_s, N_{chi} and N_{phi} grid-nodes, respectively. This domain is then decomposed into N_{cart} sub-domains over the toroidal direction and distributed across the same number of cores (Fig. 38), typically as many as toroidal grid-nodes.

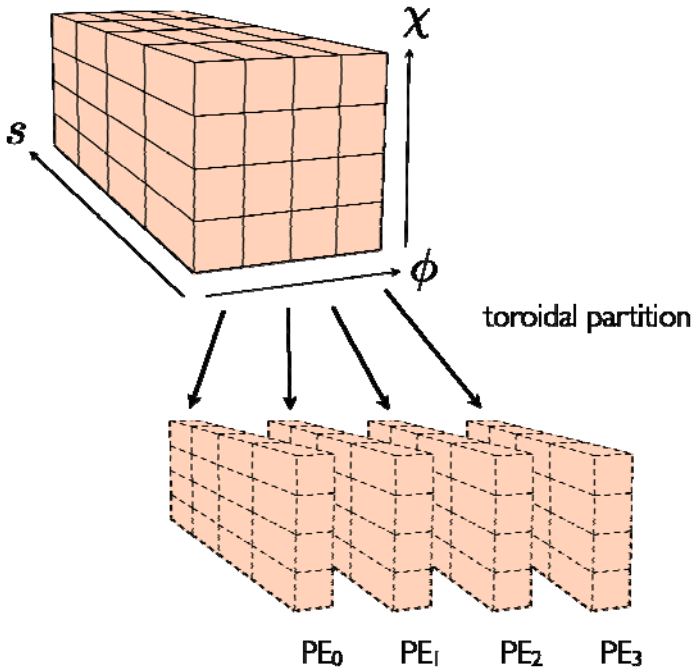


Fig. 38 Illustration of the spatial grid parallelization of NEMORB, with the sub-domains distributed in the toroidal direction for the case N_{cart}=4.

9.2.2. Multi-dimensional Fourier transforms

The basis functions for Fourier transforms are sines and cosines, which do not have a compact support (they are not spatially localized). Therefore, for numerical calculation of discrete Fourier transforms (DFT), data locality is of key importance. The same obviously applies to multi-dimensional DFTs and this directly regulates its parallelization (and data distribution) concept. In particular, an *N*-dimensional DFT can be obtained from *N* one-dimensional (1D) DFTs computed sequentially, one for each of the *N* dimensions. Since, as discussed before, each of these 1D transforms is a non-local operation, for the sake of communication efficiency, all the corresponding input data should be locally available on the core doing the computation. For the case of NEMORB, a 2D DFT on the toroidal and poloidal angles needs to be computed. The toroidal decomposition of its spatial domain (Fig. 38) sets the poloidal direction as the first one to be (1D) Fourier transformed, since the corresponding data is locally available to each core. Subsequently, the toroidal Fourier transform must be carried out, but since the corresponding data is distributed across the cores, it must first be made local to each of them. This is done with a matrix transpose that swaps the toroidal and poloidal data, as explained next.

9.2.3. XOR/MPI_Sendrecv distributed transpose algorithm

The original NEMORB's distributed matrix transpose algorithm, written by Trach-Minh Tran and Alberto Bottino, implements the necessary all-to-all communication pattern via a pairwise exchange algorithm. First, the (N_{chi} x N_s x N_{phi}/N_{cart})-sized data local to each core, represented by the slabs in Fig. 38 and Fig. 39 is sub-divided into as many sub-blocks or “pencils” as there are toroidal sub-domains (cores), i.e. N_{cart}. This corresponds to the blue cuts in Fig. 39 which yield sub-blocks of size N_{chi}/N_{cart} x N_s x N_{phi}/N_{cart}. Then, an “exclusive or” (XOR) condition establishes the exchange pattern of the sub-blocks across all cores. Excluding the diagonal sub-blocks for simplicity, since they don't need to be communicated, this pattern yields a set of (N_{cart}-1) x N_{cart}/2 pairs of sub-blocks to be swapped, as illustrated by the red arrows in Fig. 39. Furthermore, these pairs are organized into N_{cart} sub-groups in a non-overlapping manner. Therefore, the point-to-point data exchanges corresponding to all pairs within such a sub-group can proceed in parallel through MPI_Sendrecv calls, without race conditions. After sequentially going through all these N_{cart} sub-groups of pairs, the whole matrix transpose is achieved. This yields an extremely efficient all-to-all

communication algorithm, at least for the problem sizes under consideration within this work.

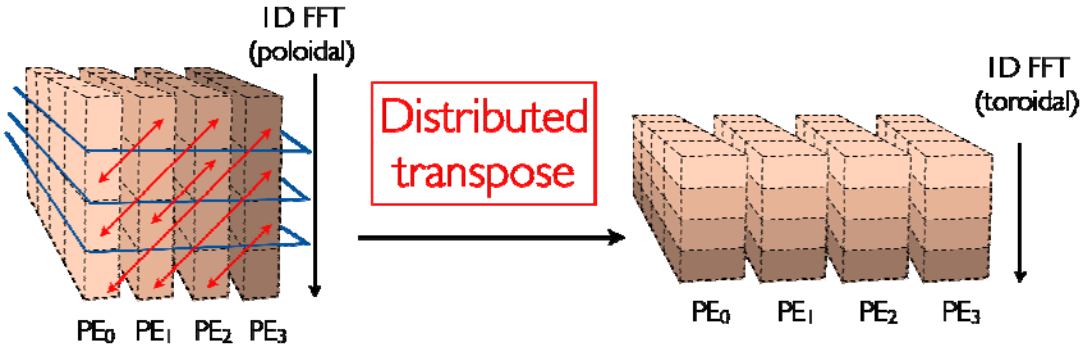


Fig. 39 Illustration's distributed 2D FFT algorithm. It consists of a 1D FFT along the poloidal (local) direction, followed by a transposition between poloidal and toroidal (distributed) directions and lastly by a 1D FFT in the toroidal direction.

9.2.4. Basic algorithm profiling

Before proceeding to the optimization of the algorithm in hand, it is necessary to measure the cost of each of its components. The major ones have already been presented in the previous two sections, but an overview of all the steps involved is still missing. This is the purpose of the following lines.

The initial step prepares the input data to be Fourier transformed. Since it corresponds to the right-hand side of the Poisson equation, it is therefore real-valued. Because the complex-to-complex (c2c) Fourier transform algorithm is used, the real-valued input data array is copied to a new complex array with the same size and imaginary part set to zero. Additionally, it has the first and second indexes exchanged, from the original order (*s*,*chi*,*phi*) to (*chi*,*s*,*phi*). The reason for this is simply that NEMORB's subroutines performing the 2D DTFs acts on the 1st and 3rd indexes of a 3D complex array. This operation of index-swapping results in a local matrix transposition (all the data is locally available) and constitutes the first place where optimization can be applied. The relative cost of this compared to the whole algorithm can be checked in the profiling table (Table 3), where it corresponds to the "Reord" entry.

The next steps involve calculating the Fourier transforms. First in the poloidal (local) direction for all radial and toroidal grid-nodes. This is done via a standard c2c fast Fourier transform (FFT, a particularly efficient method to calculate DFTs for power-of-two sized arrays) and called "FTcol1" in Table 3. The result is stored in a 3D complex matrix with size $N_{chi} \, x \, N_s \, x \, N_{phi}$ that is distributed across N_{cart} cores over the last dimension. It further needs to be Fourier transformed in the toroidal direction, but since the corresponding data is now distributed, it must first be made local to each core. Such task is achieved using the transpose algorithm outlined in Sec.9.2.3, which in Table 3 is denominated by "Transp". The resulting transposed 3D complex matrix with size $N_{phi} \, x \, N_s \, x \, N_{chi}$, also distributed across N_{cart} cores over the last dimension (here the poloidal direction), can now be efficiently Fourier transformed in the toroidal direction using the same c2c FFT algorithm as before. Its cost is measured by "FTcol2" in Table 3. Fig. 39 schematizes the last three steps just described in this paragraph.

		Inclusive			Exclusive		
Subroutine	#calls	Time(s)	%	MFlops	Time(s)	%	MFlops
Main	1	72.392	100.0	614.469	0.014	0.0	4.990
:Reord	500	5.690	7.9	0.002	5.690	7.9	0.002
:FTcol1	500	8.303	11.5	2745.803	8.303	11.5	2745.803
:Transp	500	47.634	65.8	0.199	47.634	65.8	0.199
:FTcol2	500	7.357	10.2	2803.278	7.357	10.2	2803.278
:Normal	500	3.395	4.7	310.034	3.395	4.7	310.034

Table 3 Performance measurement of NEMORB's 2D FFT algorithm executed 500 times on a typical ITER-sized spatial grid (nchi=2048, ns=512 and nphi=1024) distributed over 1024 cores in HPC-FF. The "hotspot" is the distributed transpose, highlighted in red.

Table 3 shows clearly that the distributed matrix transpose represents the most significant part (65%) of the whole time cost of NEMORB's 2D FFTs. This percentage can even increase for smaller exchange block array sizes, as shall be discussed in more detail later. Since this part of the algorithm is mostly inter-core communication (note the lower MFLOPs compared to the 1D FFTs), it means that the algorithm is communication-bound. Most of the optimization effort has to be put, directly or indirectly, on this part of the algorithm. Nevertheless, since the index-swapping part ("Reord") offers complementary room for improvement, that shall be the starting point addressed in the remaining text.

9.3. Index order swapping: local transpose

The index order of the data in the main NEMORB code is (*s,chi,phi*) but its Fourier transform subroutines require (*chi,s,phi*). This requires performing a local transpose between indexes 1 and 3 on the code's 3D data array. This is done in the original code via two nested DO-loops. Such method is here compared to both Intel MKL (mkl_domatcopy) and FORTRAN intrinsic local transpose counterparts for several matrix sizes. An example of the results obtained is given in Fig. 40.

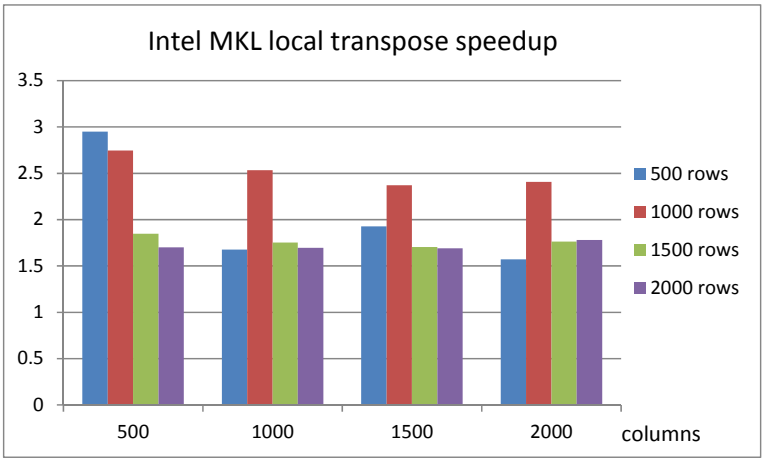


Fig. 40 Intel MKL (v10.2.5) local transpose speedup relative to a DO-loop transpose (ratio of the time cost of the latter over the former) on real-valued data matrices with sizes between 500 *x* 500 and 2000 *x* 2000, measured on HPC-FF in Jülich.

For these matrix sizes, which are in the range of NEMORB's ITER-sized simulations, the MKL transpose is always faster. Even though this result is not generalisable for all matrix sizes (the speedup pattern can be quite complex, with degradation or even slowdown occurring for specific matrix sizes), the rule of thumb is, the bigger the matrix, the better the MKL transpose performs. This is intuitive since for smaller matrices the overhead of using the MKL algorithms dominates.

9.4. Fourier transform of real data: Hermitian redundancy

The optimization plan for NEMORB's Fourier transforms invokes the Hermitian redundancy inherent to its real-valued input data (right-hand side term of Poisson's equation). Such redundancy implies that the DFT of a N-sized series can be unequivocally represented with only $N/2+1$ independent complex values. Taking this property into account allows to calculate DFTs of real data in a more efficient manner compared the full complex-to-complex (c2c) transform, both memory- and CPU-wise. These are called half-complex transforms (real-to-complex – r2c, or complex-to-real – c2r for the inverse transform).

In NEMORB, the Hermitian redundancy can be used in the first array index to be Fourier transformed, namely, the poloidal direction (*chi*). This effectively reduces the number of needed floating point operations (FLOPs) by roughly a factor of two. The same applies to the size of the transformed stored data-set, which for the N_{chi}-sized real-valued input array, falls from the full N_{chi} complex Fourier modes to a $N_{chi}/2+1$ subset of them. The subsequent Fourier transform, in the toroidal direction (third array index), acts upon the already poloidally Fourier transformed data, which is complex. Therefore, it no longer exhibits the Hermitian redundancy and so, for each poloidal Fourier mode, the full-complex (c2c) toroidal Fourier transform has to be carried out. However, since the number of poloidal Fourier modes was cut by roughly a half before, so is the number of full-complex toroidal Fourier transforms that needs to be calculated. Together this leads to an expected speedup factor of two for the whole of the Fourier transform operations in NEMORB's 2D filtering algorithm.

A direct consequence of using half-complex poloidal Fourier transforms is that, due to the effective reduction in the size of the data-set, less information needs to be exchanged across cores. Therefore, a similar performance gain should be yield by the distributed matrix transposition part of the algorithm. However, it must be noted that, $N_{chi}/2$ rows are easily divided into a uniform set of N_{cart} sub-blocks, as the XOR/MPI_Sendrcv algorithm demands, whereas $N_{chi/2+1}$ would require substantial zero-padding for high N_{cart} values. Therefore, the expected 2x speedup is only possible provided that a storage trick is applied to reduce its poloidal dimensions by one element. Such trick is possible because the zero (F_0) and Nyquist ($F_{N/2+1}$) Fourier modes are purely real (for even N). Therefore, without any loss, the real part of the latter can be stored in the imaginary part of the former, in what is denominated the half-complex packed format. Obviously, after the data transposition, before the second (toroidal) Fourier transform can be computed, the data must be converted back to the “un-packed” half-complex format. Since the poloidal direction is at this stage distributed across different cores, this task requires an additional point-to-point communication between core rank 0, which needs to send the $F_{N/2+1}$ value, and core N_{cart}-1, where it it is received. This is done with a call to the MPI_Send and MPI_Recv directives at no extra significant relative cost, as shall be seen next.

Putting all together, including the index swapping optimization described in Sec.9.3, leads to the performance results shown in Fig. 41 for the whole 2D FFT algorithm on the HPC-FF machine, Jülich Supercomputing Center (JSC). This plot shows the weak scaling (same amount of work load per core) speedup factors achieved with the new method compared to the original one. Both cases, with (yellow) and without zero-padding (packed format, green), are shown.

The degrading effect caused by the zero-padding can be inferred from the difference between both curves. Indeed, for 1024 cores, the need for zero-padding on the half-complex data yields a matrix with the same size as the original full-complex one. No transpose speedup is yield in this case (yellow curve) and the gain measured corresponds solely to the reduction in the Fourier transforms FLOPs. Conversely, the green curve, where the half-complex packed format was used shows better scaling

properties. As expected, the total speedup is always the same or higher than before, even for the cases that originally required very small zero-padding percentages. This means that the overhead communication costs related to the packed-format conversion are negligible. For higher numbers of cores (N_{cart}), which without the packed format requires a substantial amount of zero-padding, the gain is clear and arises from having less amount of communication to do (smaller matrix to transpose). Still, for the two highest N_{cart} values, the yellow curve shows a degradation of the two-fold speedup that can not be accounted by the matrix size increase, as there is no zero-padding involved whatsoever

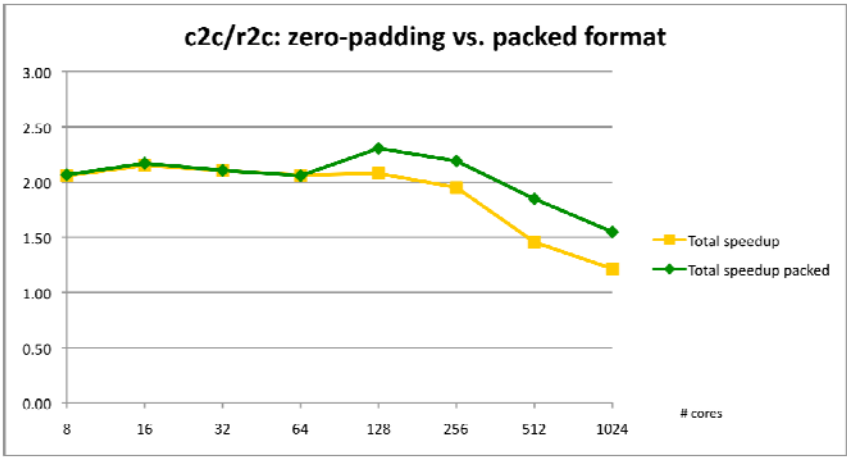


Fig. 41 Weak scaling speedup factors achieved on HPC-FF for the complete 2D FFT algorithm with (yellow) and without (green) the half-complex packed format representation on the poloidal direction on a grid count of 2048 *x* 512 *x* N_{cart}, in the poloidal, radial and toroidal directions, respectively.

The observed degradation is related to the latency times inherent to any inter-core communication. Two factors contribute to this problem. To understand them lets first recall that, as explained in Sec.9.2.3, both (i) the number of pairs of sub-blocks ("pencils") which are exchanged simultaneously in parallel and (ii) the number of times this exchange operation has to be sequentially performed to achieve the full distributed transpose increase linearly with N_{cart}. The former naturally implies an increase in the network congestion (collisions) with N_{cart}, which enhances the effective network latency. The latter implies an accumulation of latency proportional to N_{cart}. Moreover, since the size of the sub-blocks being transposed decreases accordingly (they're given by N_{chi}/N_{cart} *x* N_s *x* N_{phi}/N_{cart}), for higher core-counts the latency times becomes comparable to the time spent on the actual transfer, leading to scaling saturation. This was confirmed with measurements of the speedup factors for the parallel transpose on weak scaling tests with different matrix sub-block sizes. Smaller sub-blocks reached sooner the scaling saturation, whereas the opposite was observed with larger sub-block sizes.

At the time these tests where made, the HELIOS machine in Aomori was not yet available for production. This topic shall be revisited later in the remaining text. For the time being, the results presented so far serve as motivation to experiment with alternative distributed transpose algorithms, which might be less prone to network latency issues. This is the material covered in the next few sections.

9.5. Distributed transpose: XOR/MPI_Sendrecv vs. MPI_Alltoall

The network latency impact on the transpose performance measured on HPC-FF motivates the use of alternative methods to perform the same task. The most obvious one invokes the MPI_Alltoall directive, which establishes the all-to-all communication pattern and performs the message passing automatically. To be able to use it in the

simplest way, the sub-blocks to be exchanged across cores need to be stored continuously in memory. This is obviously not the case for the problem in hand, where the corresponding sub-blocks are 3D “pencils”. Two possibilities are available. Either use temporary buffer arrays, where the data for each sub-block is stored continuously before being communicated, or create a taylored data type that specifies the memory access pattern to the sub-blocks in a continuous manner. Both have been implemented (based on the source code kindly provided by Paolo Angelino) and tested.

The first solution is the simplest. All that is needed is to create two temporary 4D buffer arrays to store the several sub-blocks continuously before being sent and after being received by the MPI_Alltoall directive, respectively. The second solution generalizes the previous method to avoid explicit usage of temporary storage buffers. This can be achieved with the most flexible MPI derived data type, namely, MPI_Create_data_struct. Passing it as an argument to the MPI_Alltoall allows reading/writing directly the original 3D input/output sub-blocks, with the eventual needed data buffering being handled “under the hood” by the MPI library. This is therefore expected to be more efficient, although it will be seen in the following paragraphs that this is not always necessarily the case.

The very same principles can be applied to the XOR/MPI_Sendrecv method, which also uses temporary storage buffers to hold a single input/output sub-block before/after it is communicated with the MPI_Sendrecv call. Here the overhead involved in copying the temporary buffers is, in principle, less significant than within the MPI_Alltoall method, since they are much smaller and therefore have better chances to fit in the cache memory. Nevertheless, avoiding this explicit task could also be beneficiary in this case, so this method was implemented and tested along side with the others.

The transpose methods explained before were tested on a fixed grid-count of N_{chi}=2048, N_s=512 and N_{phi}=1024, which is representative of NEMORB's spatial grid for a typical ITER simulation. A strong scaling study, obtained by distributing the problem on different numbers of cores, was made on HPC-FF, RZG-VIP and the meanwhile available IFERC-HELIOS. On the last, two different MPI library implementations were used, namely, the default BullX MPI and Intel MPI. The results are gathered in Fig. 42, where each point on the graph represents the time it took to perform the transpose of the 3D grid (between the poloidal and toroidal directions) followed by the corresponding back transpose, repeated a thousand times. The best case out of at least five runs is plotted.

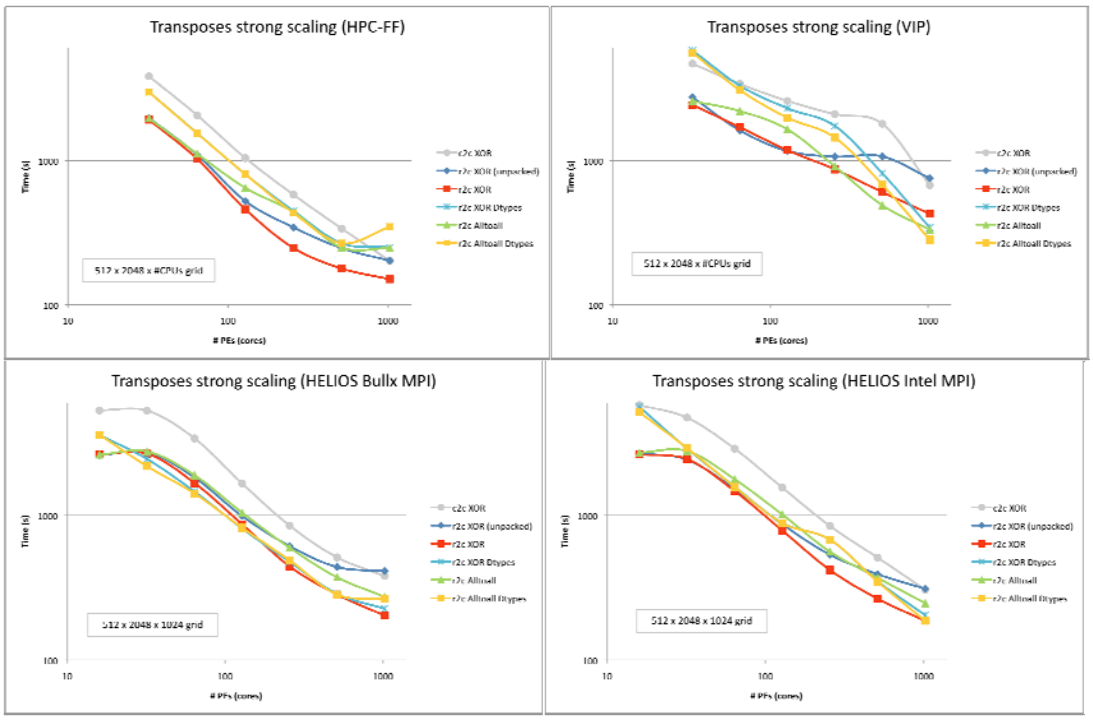


Fig. 42 Strong scaling on a grid-count of nchi=2048, ns=512 and nphi=1024 for the different transpose methods described in the main text. The results were obtained on HELIOS/IFERC using Bullx MPI v1.1.14.3 (top) and Intel MPI v4.0.3 (bottom).

The XOR/MPI_Sendrecv method with temporary storage buffers was used in three variants. The first of them acts upon the original Hermitian redundant complex-to-complex (c2c) Fourier transformed data. It yields the grey curves labeled “c2c XOR”, which give the reference data, since they correspond to the original method. The second variant is given by the blue curves labelled “r2c XOR (unpacked)”. They correspond to the Hermitian reduced data yield by the real-to-complex (r2c) poloidal FFT without the packed format, which requires zero-padding (Sec.9.4). The third variant remedies this by adding the half-complex packed format. It yields the red curves, labeled “XOR r2c”. The light-blue curves labeled “XOR Dtype” use an MPI derived data type to avoid the explicit temporary storage buffers. The remaining curves correspond to the two MPI_Alltoall variants applied to the Hermitian reduced data. The green curves labeled “r2c Alltoall” use temporary storage buffers, whereas the yellow curves labeled “r2c Alltoall Dtypes” use instead MPI derived data types.

The most important conclusion to be drawn is that, barring the RZG-VIP case of IBM Power6 hardware, the most efficient method seems to be the half-complex packed XOR/MPI_Sendrecv already used in the previous section (red curves). The speedup measurements made relative to the original full-complex algorithm yielded values of about 1.5, 1.7 and 2.0 for HPC-FF, RZG-VIP and HELIOS, respectively. Additionally, among the cases that use the MPI_Alltoall directive, it is the derived type variant (yellow curves) that performs best, as was expected. This method even yields the best results on RZG-VIP (a speedup of 2.4 was reached). Conversely, for reasons which are not transparent, at the time this report was written, the opposite happens on HPC-FF, where this method is even slower than the original full-complex transpose (grey curve). Together with the less performant derived data type variant of the XOR/MPI_Sendrecv algorithm on this machine, they seem to indicate that using derived data types within the ParTec MPI distribution available in the HPC-FF is not the most efficient way to go. As such, whether or not using derived data types makes sense from the performance point of view depends strongly on the MPI implementation available on a given machine. As to the question of the latency effect seen before, which motivated the use of different transpose algorithms in the first

place, it seems that it has a stronger impact on HPC-FF (curves start to flat out for higher numbers of cores) than on the other two machines.

Another important point is that, loosely speaking, the general shape of the strong scaling curves on HPC-FF and HELIOS is rather similar, especially when compared to the RZG-VIP's counterparts. From the machine architecture point of view, this is the expected behavior. Nevertheless, in absolute terms, the fastest results were obtained on HPC-FF, which was not expected, considering the similarity of the architectures and that HELIOS has more recent hardware. In reality, a more detailed analysis of the issue revealed that this difference is related initialization costs for the all-to-all communication pattern. This renders the interpretation of such cross machine comparisons impaired, and for that reason, a detailed study of the issue was conducted in parallel. This is the material presented in Sec.9.8. For now it suffices to state that the communication initialization costs are properly accounted for in what remains of the scaling studies shown in this report.

9.6. Additional transposes: FFTW and MKL libraries

This section starts with the general conclusion of the previous one, namely, that the MPI_Alltoall directive is not superior to the XOR/MPI_Sendrecv counterpart for the transpose problems under consideration. The follow-up investigations attempt to further find better alternatives to perform this task. Two additional transpose algorithms are added to the set already tested before. These come from the FFTW library and the Intel MKL implementation of the ScaLAPACK library. For testing purposes, a simpler 2D version of the transpose was implemented and compared to the 2D version of two of the methods implemented before, namely, the XOR/MPI_Sendrecv and the MPI_Alltoall, both without using derived data types (red and green curves in Fig. 42). This choice is based on the overall performance of the former and the close relation of the latter to the FFTW3.3 transpose algorithm.

The test cases correspond to a matrix size of $N_x x N_y = 32768^2$, parallelized over the second dimension. So, on a given number of cores N_{cart}, each core holds $N_x \times N_y/N_{cart}$ of the input data and $Ny \times N_x/N_{cart}$ of the transposed data. This choice yields a similar effective data-size to the nominal 3D ITER-case of the previous sections. The figures obtained on HELIOS and HPC-FF on 1024 cores are listed in Table 4. They correspond to the best results out of at least three simulations.

	XOR	All-to-All	FFTW3.3	MKL	MKL (impi)
HELIOS	14.4s	25.2s	46.1s	839.6s	1232.8s
HPC-FF	12.9s	41.2s	25.6s	1166.9s	–

Table 4 Time spent by the different transpose algorithms to perform 100 pairs of transpose and back-transpose of a 32768^2 matrix on 1024 cores, excluding communication initialization costs. On HELIOS, the Intel Fortran compiler v12.1.1 and the libraries FFTW v3.3 and MKL v10.3.7 were used with BullxMPI v1.1.14 for all results, except the last column one, for which the Intel MPI v4.0.3 was used. On HPC-FF, the Intel Fortran compiler v12.1.1 and the libraries the MKL v10.2.5 and Partec MPI v5.0.26-1 were used.

It can be straightforwardly seen that, neither FFTW's nor MKL's PBLAS implementations of the all-to-all communication are competitive with the previous algorithms in hand. The latter even proved to be more than two orders of magnitude slower than its counterparts. A strong scaling study of this method revealed its poor scaling properties, which explains the figure in Table 4. The former, which uses the Fortran ISO C bindings, proved hard to have working with complex numbers, even though it should be possible using "tuples", as stated in the manual. This together with the not very encouraging performance results obtained with real numbers implied that no further time investment was made in this topic. For essentially the same reason, no big effort was put into developing the 3D counterparts of the new

transpose algorithms, which would mimic the actual NEMORB's grid-count. Moreover, the MKL case, being done using matrix descriptors within BLACS that expect 2D matrices, makes it very hard (perhaps even impossible) to have an efficient 3D version of the transpose. In this case, one has to perform each 2D transpose separately, and repeat the process for each grid-point in the third dimension. This is obviously very inefficient compared to the remaining methods, which carry the whole third dimension when the 2D transpose is calculated. The MKL 3D tests made on HELIOS with NEMORB's nominal ITER grid-count (N_{chi} *x* N_s *x* N_{phi} = 512 *x* 2048 *x 1*024) revealed a slow down of more than three orders of magnitude compared to the other methods, as one would expect from the previous considerations.

9.7. Optimization of the XOR/MPI_Sendrecv transpose

Having tried several different alternative algorithms to the distributed transpose, it became clear that the best choice to perform such task is the XOR/MPI_Sendrcv method. Moreover, this choice further offers optimization possibilities related to its "hand-coded" algorithm. Unlike the methods relying on "canned" libraries, this one allows for changes to the communication pattern. The following two sections explore this possibility by following different order in the sequential communication steps involved (Sec.9.7.1) and by invoking the zeros yield by NEMORB's Fourier filters to inhibit part of the communication (Sec.9.7.2).

9.7.1. XOR exchange pattern sequence

The XOR/MPI_Sendrcv algorithm follows a communication pattern matrix row-by-row to perform the transpose in N_{cart} sequential steps within a DO-loop. Each step (row) provides a set of pairs of cores that exchange specific sub-blocks of data in a non-overlapping fashion. Since such steps of communication are independent of each other, there is no restriction on the order of which they are executed.

An extra array with dimension N_{cart} was created to store the order over which the rows of the XOR exchange pattern matrix are to be followed. Four different cases were used, namely, the standard sequential increasing order ("Reference"), the reverse order sequence ("Reverse"), the ("Alternating") sequence order given by {1, N_{cart}, 2, N_{cart}-1, …, N_{cart}/2, N_{cart}/2+1} and finally the random sequence order, obtained using a Knuth shuffle algorithm. Following the same practice as before, the communication initialization costs have been removed from the values shown in Table 5, which shows the distribution of elapsed times for the different communication-pattern sequences.

	Reference	Reverse	Alternating	Knuth shuffle
HELIOS	207.7 ± 5.9s	205.5 ± 7.8s	213.5 ± 3.3s	245.1 ± 1.5s

Table 5 Elapsed times of the complete 2D FFT algorithm (packed half-complex XOR/MPI_Sendrcv), followed by its inverse equivalent to revert back from Fourier to configuration space, executed 1000 times on NEMORB's ITER-sized spatial grid count (N_{chi}=2048, N_s=512 and N_{phi}=1024). Each column corresponds to a different communication-pattern sequence. Eight simulations for each sequence were performed and the average values are shown.

The best results were obtained for the case where the DO-loop proceeds in reverse order, although these were only marginally faster than the standard case figures with the difference well inside the error bars. The alternating sequence yields slower execution times but it is the random order ("Knuth shuffle") which gives the worst results, even subtracting the overhead related to the random shuffling of the sequence, which is done only once, before the first transpose is executed. These findings agree with the expectation that ordered memory access patterns pay-off

compared to erratic ones, which are prone to cache misses. Similarly, the alternating sequence seems to have more cache misses than the standard and reverse cases that maximise the continuity of the sub-block memory access pattern inside the transpose DO-loop. Based on these results, the “Reference” sequence was kept as the default. The marginal gain obtained (<1%) with the reverse order sequence does not justify the necessary changes, especially considering how this would obscure the subject of the next section.

9.7.2. Square Fourier filter: partial matrix transposition

This section outlines the exploitation of the Fourier filtering, which forces a large fraction of the transformed data matrix to vanish. Naturally, these zeros need not be transposed, and if they aren’t, this should in principle alleviate the communication burden of the algorithm.

The basic filter used in NEMORB is a diagonal one, which retains only a relatively narrow band of poloidal modes around each toroidal mode in the system. Physically this corresponds to retaining only modes that are relatively well field-aligned. All the modes outside this band are set to zero, since they do not take part in the dynamics and moreover could contribute to numerical noise. The problem is that this filter, being diagonal, can only be applied after the Fourier transforms in both angles are calculated, and not before. Indeed, after the poloidal (local) Fourier transform is performed, the resulting (poloidal) spectrum needs to be carried out as a whole into the toroidal Fourier transform, since the toroidal direction still contains configuration space data. This means that the full transpose must be carried out before the field aligned filter can be applied. The same holds for the back transpose that is done after the filter is applied. Such operation must occur only after the inverse toroidal Fourier transform is applied, which converts the narrowed toroidal spectra into a filled-up matrix corresponding to its toroidal configuration space representation. A workaround was attempted while trying to extend the convolution theorem treatment already used in NEMORB [1] to this part of the problem. Nevertheless, the same conclusion was reached, namely, one needs to carry out the full matrix transposes simply because the filter mixes both angular dimensions in a non-separable form.

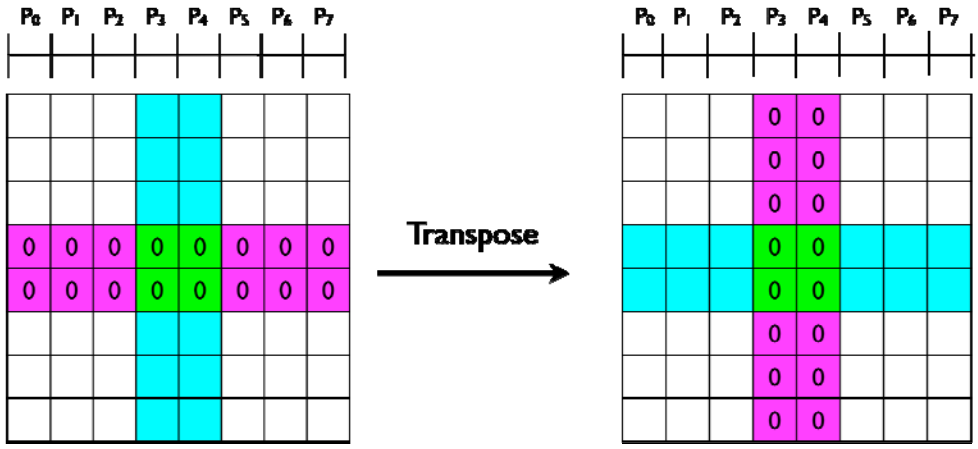


Fig. 43 Illustration of the reduce data transposition required after the “4-node per mode” low-pass filter is applied to the poloidal Fourier transformed data. The white areas are transposed using original bi-directional message exchange (MPI_Sendrecv). The cyan area sends the data to the magenta area without receiving the corresponding zero-elements (uni-directional message passing). The zeros in the green area need not to be communicated at all.

On the other hand, besides the field-aligned filter, a square filter is also applied to the data to ensure a lower Nyquist frequency cut (at least four nodal points per wavelength, instead of two). Unlike before, it is possible to separate the projections of the square filter in the poloidal and toroidal directions. Hence, one can apply the filter's poloidal projection to the Fourier transformed poloidal data before it is

exchanged across cores, therefore avoiding communicating the resulting zeros, as is illustrated in Fig. 43, neglecting any Hermitian redundancy for the moment for the sake of simplicity. The band set to zero in the middle corresponds to the higher frequency modes and is specified by the poloidal index interval [N_{chi}/4+1, 3N_{chi}/4]. The white regions maintain the XOR communication pattern unchanged, with the messages being exchange bi-directionally between the corresponding cores involved. The magenta and cyan regions involve only uni-direction message passing. The cores with non-zero elements (cyan) send them to their counterpart cores, which in turn have zero-elements (magenta). So, the former need not to receive any data from the latter. Finally, the core area (green) requires no communication whatsoever. Obviously, the same principle can be directly applied to the toroidal part of the square filter, even though this is not done here.

If implemented, the uni-directional communication parts of the partial transpose algorithm (magenta/cyan areas) would reduce the message size only, but not their number. Therefore, they are only expected to improve significantly the performance of the algorithm if it were network bandwidth-bound. Since however, the results obtained so far point to network latency and congestion as the main factors limiting the scalability of the algorithm, most of the performance gain is expected to come from the green area of the matrix, where no communication happens. Implementing this part effectively reduces the number of concurrent messages exchanged simultaneously, which is what is needed.

	Reference	Partial (filter)
HELIOS	208.0 ± 2.0s	197.3 ± 3.5s
HPC-FF	209.3 ± 16.6s	182.3 ± 9.2s

Table 6 Elapsed times for seven and three simulations made on HELIOS and HPC-FF, respectively, with full and “filtered” partial transposes. The speedups are in the range 5-10%.

The plot in Table 6 compares the cost of the new NEMORB's 2D filter algorithm using the previous considerations (“Partial”) and the same algorithm without doing so (“Reference”). An improvement in performance in the range 5%–10% is achieved. This further enhances the speedup factors shown in Sec.9.5. On HELIOS the improved speedup factor reached about 2.2. On HPC-FF, the degradation observed previously was also reduced, with the speedup factor increasing from below 1.5 to slightly above 1.7.

The final remark relates to the filter choice presented here, which was motivated by the default poloidal component of the square filter used in NEMORB. In principle the algorithm also allows different poloidal filter frequency values, although this part of the code was not tested extensively. Moreover, it should be noted that changing this values affects the performance. Obviously, narrower filters yield less zeros and therefore less communication is inhibited. Finally, because of its Hermitian redundancy, the algorithm imposes that the poloidal filter must be symmetric on the full (positive and negative) poloidal mode number domain. No different positive and negative frequency cuts are allowed.

The material directly related to the optimization of NEMORB's 2D Fourier filtering is concluded with the results present here. The remaining section addresses the issue of all-to-all communication patterns initialization cost on the HELIOS machine. This constitutes a side study made in order to understand the performance figures that were being obtained on this facility.

9.8. Initialization of all-to-all patterns: HPC-FF vs. HELIOS

As mentioned in Sec.9.5, the poorer results obtained on HELIOS compared to the ones obtained on the HPC-FF machine motivated more detailed investigations on

how the all-to-all communication takes place on both machines. Specifically, the main goal was to check how much time was spent in the preparation of the MPI communication pattern needed for the transposes. For that purpose, the same problem size of the previous sections was used (nominal NEMORB's ITER grid-count) on 1024 cores and the number of calls to the transpose algorithm made was varied in powers of 10, from 1 to 1000. Fig. 44 shows the corresponding figures on both HELIOS and HPC-FF. Two settings for the MPI library were used, namely, static (on demand=0) and dynamic (on demand=1) connection allocation. In the former, the establishment of the MPI connections occurs upfront, when the MPI_Init is called. Conversely, the latter does this step only the first time it is needed, which within this algorithm is when the first pair of transposes is called.

On HELIOS, one clearly sees that Bullx MPI has the highest all-to-all initialization cost of them all (leftmost red data points) and the opposite applies for the Intel MPI (leftmost yellow data points). On HPC-FF, both MPI libraries available there, namely, ParTec MPI (v5.0.26-1) and Intel MPI (v3.2.2.006), seem to be in between the results obtained on HELIOS, in terms of MPI initialization costs (leftmost green and blue data points). The observation that Bullx MPI on HELIOS yields considerably higher communication allocation times than the remaining led to the decision to remove this part of the cost from the scaling studies presented before. Otherwise, they would masquerade them, unless very large number of times steps were used. This can be seen from the right-most points in all curves that yield the asymptotic behaviour. There the differences caused by the initial offset fade away (note the logarithmic scale).

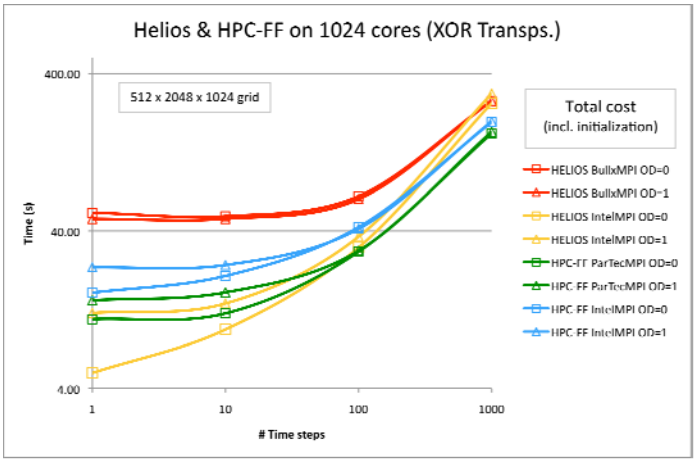


Fig. 44 Comparison of the total XOR/MPI_Sendrecv transpose time cost, including the time spend in the MPI initialization of the algorithm, on HELIOS with both Bullx MPI (v1.1.14.3) and Intel MPI (v4.0.3) and on HPC-FF with ParTec MPI (v5.0.26-1) and Intel MPI (v3.2.2.006). Both static and dynamic MPI connection allocations are shown for every case.

The importance of the communication initialization issue on Bullx MPI led to its continuation in the BLIGHTHO project. This is a topic of general interest which affects all codes using all-to-all communication patterns on large numbers of cores, not only in the context of the HELIOS machine, but as a general scalling issue for next generation of computers.

9.9. Conclusions

This report summarizes the work done on the NEMOFFT project on the optimization of NEMORB's bi-dimensional Fourier filtering algorithm. The final figures are the speedup factors achieved with the modifications made. Namely, the new algorithm is about 2.2 and 1.7 times faster than the original, on HELIOS and HPC-FF, respectively.

The optimization involved the implementation of the Hermitian redundancy on the NEMORB's Fourier transforms. This enabled the reduction of the number of FLOPs involved by a factor of two. Further using a clever storage technique for the Hermitian-reduced data (half-complex-packed-format) reduced also the data-set to be transposed across cores by the same amount, which led to the expected speedup factor of the order of two on HELIOS with nominal grid-counts representative of a typical ITER simulation. On HPC-FF, the degradation observed in this factor (from 2 to 1.5) was found to be related to network congestion/latency limitations. This motivated the implementation of alternative transpose methods. Nevertheless, the performance measurements made on those, which included using the MPI_Alltoall directive (with and without derived data types) and its FFTW3.3 and MKL ScaLAPCK counterparts, as well as an alternative algorithm based on the MPI_Gather/Scatter directives, revealed that the original "hand-coded" XOR/MPI_Sendrecv algorithm gave, in general, the best results.

Another reason taken into account for recommending the XOR/MPI_Sendrecv as the default method is related to its flexibility. This allowed to exploit NEMORB's low pass filtering, used to ensure enough resolution on all physically allowed modes in the system, to further reduce the data-set to be exchanged across cores. Indeed, *a priori* knowledge of the matrix elements that are set to zero by the filter enabled parts of the matrix to be excluded from the data exchange step, such that only a partial transposition was performed. This further increased the gain obtained to the final figures stated in the first paragraph of this section.

The last point refers to a side-study motivated by the excessive cost of all-to-all communication initialization on HELIOS with Bullx MPI. The conclusions obtained therein were taken into account when interpreting the various scaling measurements made throughout the work. The relevance of this issue led to further investigations made by M. Haefele in the framework of the BLIGHTHO project, in close contact to with Bull.

9.10. References

[1] B. F. McMillan et al., *Computer Physics Communications* **181**, 715–719 (2010).