EUROfusion

EUROFUSION WPISA-REP(16) 16105

R Hatzky et al.

HLST Core Team Report 2011

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 2011

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
ASCOT-11	Nitya Hariharan	finished
EUTERPE	Nicolay Hammer	finished
GYNVIZ	Matthieu Haefele	running
ITM-ADIOS	Michele Martone	running
KINSOLD	Kab Seok Kang	finished
MGTRI	Kab Seok Kang	running
NEMOFFT	Tiago Ribeiro	running
ZOFLIN	Nitya Hariharan	finished

Table 1 Projects mapped to the HLST core team members.

Roman Hatzky has contributed in particular to the projects EUTERPE and ZOFLIN. Furthermore, he was occupied in management and dissemination tasks, e.g. the development of the HLST web site, due to his position as core team leader.

Matthieu Haefele worked mostly on the GYNVIZ project and contributed to the EUTERPE project by improving the restart mechanism on varying numbers of cores.

The aim of the GYNVIZ project is 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.
Three main components can be identified. The first one consists of a uniform data format. Last year, it had been decided to use the XDMF format for that purpose because of the wide spectrum of data types it can express and its flexible design. However, the XDMF software in its original form did not provide all functionalities to make it fully functional in the context of GYNVIZ. Thus, a collaboration with the XDMF team in the U.S. was set up in order to improve parts of the implementation according to our concerns. Unfortunately, this collaboration failed in the middle of this year due to a total lack of response from the XDMF developer side. We decided to abandon the XDMF technology from the GYNVIZ project. As a consequence, the initial intention of bringing a common and standard format to a range of different codes had to be given up.
The second component consists of a set of post-processing software whose development is nearly completed. The main part is intended to turn 3D time-varying datasets into 4D compressed ones in order to explore them with 4D visualization. The 4D compression and visualization technologies were transferred from the EUFORIA project. The post-processing tool is genuinely developed within the GYNVIZ project. The 4D visualization functionality has reached now a mature production state. Some last production tests with real data are still on going. In the remaining project time we plan to provide for all GYNVIZ users dedicated support to train them on how to use these tools.
Finally, the third component consists of the network and computing infrastructures hosted by Rechenzentrum Garching (RZG) and Jülich Supercomputer Center (JSC). Initially, it was planned to use the DEISA file system to provide direct data transfer but as the DEISA project ended in April 2011 and JSC was not willing to support it anymore, we decided to switch to the *bbcp* technology for the transfer. A wrapper is provided to help the user in using the appropriate optimization options.

In addition, Matthieu Haefele contributed with a training session of a total of 13.5 hours to a one week summer school in Autrans, France. The focus was on post-processing aspects as e.g. the HDF5 library and parallel-IO. This training session is an ongoing effort of training activities Matthieu Haefele has accomplished as a member of HLST. It will proceed with the individual training of the GYNVIZ users. Thus some of the material being prepared for the summer school will be reused for this purpose.

Nicolay Hammer finished the EUTERPE project by providing the HLST report "Performance Tuning Using Vectorization", which has been published as IPP report 5/126. He left the HLST core team at 31. March 2011.

Nitya Hariharan worked on the ASCOT-11 and ZOFLIN projects.

The ASCOT-11 project was intended to give further support on the completed ASCOT-10 project. Ian Bush provided the HLST team a Fortran module under the Lesser GNU Public License (LGPL) that allows easy access to Shared Memory Segments (SMS). The new module was tested along with the spline routines from Numerical Recipes and the outcome was successful. Finally, we passed it onto the ASCOT group.

The ZOFLIN project was focused on the enhancement of the single processor performance and the scalability of the ZOFLIN code which is targeted on the nonlinear interaction of zonal flows (ZF) among each other. First, we have updated the code to be Fortran 2003 compliant, which allows the code developers to follow a more standard method of programming. This also helps to have a more portable code. The use of tools like Forcheck, Automake and Marmot now ensures that certain types of programming errors, neither in the application nor in the Makefile do not occur anymore.
Looking for improvements of the single processor performance, we found the efficiency of the code, with respect to the FLOP rate, to be quite high on single (2.3 GFLOPS) and multiple processors (1.6 GFLOPS per core on 64 cores). Achieving higher FLOP rates seemed to be impossible because of the memory-bound nature of the ZOFLIN code. However, usage of Fortran 77 style of programming, with fixed arrays sizes makes it easier for the compiler to optimize access to these arrays. Finding scope for optimization for such codes is not trivial since the code is quite efficient to begin with.
The code makes use of a large number of C macros which are expanded by the pre-processor. Macro usage makes it much more difficult to analyze the program as the non-preprocessed code and the preprocessed code are very different and hard for a human to read and understand. Nevertheless we have tried various changes to make the macros to use the cache more efficiently. However, we were not able to get any significant gains in run time with these changes.
We analyzed the time spent in communication in the code and found the *MPI_Waitall* to consume around 40% of the time spent in MPI calls. By combining the data that was sent across cores as part of the boundary exchange, we were able to reduce the number of calls to the asynchronous routines *MPI_Isend* and *MPI_Irecv*, but did not achieve much gain in the run time. From the run times and the FLOP rate, as given by our performance diagnostics, it is obvious that the code is very well load balanced and there is at most only a 1% difference between the highest and lowest run times.
We found ZOFLIN to perform quite well on HPC-FF, and even with all the changes mentioned above, have been able to make only minor improvements to the code. Thus the proposed 50% improvement in the run time made by the project coordinator was a too ambitious goal.

Nitya Hariharan left the HLST core team at 31. September 2011.

Kab Seok Kang worked on the KinSOL2D and MGTRI projects.

The KinSOL2D project was focused on the efficient implementation of either a multigrid solver or a Krylov subspace solver using a multigrid preconditioner. The scaling properties of such a solver are of key importance as it is planned to extend the Particle-in-Cell (PIC) code BIT1 from 1D3V to 2D3V plasma simulations of the Scrape-Off-Layer (SOL). To keep the excellent scaling properties of the BIT1 it is mandatory for the new 2D Poisson solver to have comparable scaling properties.
In contrast to standard problems, the domain contains both an internal conducting structure with a Dirichlet boundary condition and an inner empty space with a Neumann boundary condition. As long as these internal structures are aligned with the grids of the different levels of the multigrid V-cycle there seems to be just a small negative effect on the convergence rate for the multigrid method both as a solver and as an efficient preconditioner. However, if the internal structures are not aligned, the convergence rate can be significantly reduced. In the worst case the multigrid solver will not converge at all. But even then, the multigrid method can still be used as an efficient preconditioner for the Generalized Minimal Residual Method (GMRES) iterative solver, which will always converge. In this case, there is only little influence of the size of the inner empty space and inner conducting structure.
For the solvers tested here either the multigrid solver with local Gauss-Seidel smoother (MGGS) or the GMRES method with a multigrid preconditioner and a local Gauss-Seidel smoother (GMGS) gave the best results. For our test cases good strong and semi-weak scaling properties up to more than a hundred cores could be achieved. Both, the strong and semi-weak scaling properties became better, the larger the test cases were. This is plausible as the communication costs undergo a relative decrease. In addition, the iterative multigrid method has a very low memory consumption compared to direct solvers.
For details please see the HLST report *"*The multigrid method for an elliptic problem on a rectangular domain with an internal conducting structure and an inner empty space*"* which has been published as IPP report 5/128.

The MGTRI project focuses on the gyrofluid GEMT code which will contain the MHD equilibrium solver GKMHD to evolve the Grad-Shafranov MHD equilibrium. Presently GKMHD is not parallelized. Hence, a major effort in making the code parallel is to implement a parallelized multigrid solver on a triangular mesh.
We have implemented different multigrid solvers for a Poisson problem on a structured triangular grid on a regular hexagonal domain. In this context, it was crucial to derive an appropriate communication pattern between the subdomains to exchange the information necessary for the ghost nodes. To check the correctness of the implementation several tests have been performed. In addition, performance and scaling tests have shown that the preconditioned conjugate gradient method with a multigrid preconditioner with Gauss-Seidel smoother (CGGS) is the most efficient method under consideration. For the given problem sizes of $50 \cdot 10^6$ Degrees of Freedom (DoF) and $200 \cdot 10^6$ DoF the numerical results revealed almost perfect strong scaling up to 384 cores. In addition, the multigrid method as a solver and as a preconditioner yielded a very good semi-weak scaling property up to 1536 cores which improves for larger test cases.
In the remaining project time we plan to perform tests with more realistic data after having consulted the project coordinator, Bruce D. Scott.

Michele Martone worked on the ITM-ADIOS project. The project's goal is to evaluate the usage of the ADIOS parallel I/O library on the HPC-FF machine.

Initially, the ADIOS library (an unreleased version following 1.3) was installed, and its test and example programs were successfully run. A development of an ADIOS enabled I/O benchmark ("IAB", in short) was initiated. It has been decided to develop the IAB in C (just as the ADIOS library), because the ADIOS Fortran interface is not as expressive, and more error prone (no Fortran module files available). The IAB is intended to serve both as a testbed for available features (when learning to use them) and as a parallel I/O throughput benchmark. With the IAB it is possible to invoke ADIOS with different parameters (e.g.: number of MPI tasks, amount of data to be written, number of destination files to write to in parallel, etc.). However, "staging" features are not available to the general user. Instead, they require additional software which is not in the public domain.

The first activity pursued was a quantification of the potential ADIOS usage benefits on the HPC-FF machine. For that, an assessment of the maximum bandwidth achievable when writing from a serial program was done first. Using the C language I/O from a serial program, but tuning appropriately the (user side) LUSTRE filesystem “striping” parameters, it has been possible to obtain at most around 1 GB/s. Using ADIOS in a parallel environment it was possible to obtain at most 14 GB/s; that is around 75% of the system capacity. In particular, array data from 4096 MPI tasks was written to a subset of 64 files. Special care had to be taken to achieve such a high performance. For instance, when in the above reported experiment, only a single file was written, performance dropped by around 45%; when writing to 4096 sub-files, the loss was around 70%. Nevertheless, the measured speedup around 15 between “best cases” of serial (without ADIOS) and parallel (with ADIOS) output on HPC-FF is very encouraging, and suggests that the overhead of porting applications to ADIOS may be justifiable.

Further investigations will characterize the role of the different LUSTRE/ADIOS parameters at hand in order to establish general best use practices. Besides using the special purpose benchmark, adaptation of two ITM codes (GEMZA and NEMORB) to ADIOS is under way, with the twofold goal of obtaining working examples in the Fortran language, and aiming to attain an I/O performance comparable to the best IAB cases. Additionally, investigations in different usage cases (e.g. asymmetry in the amount of data on the different processors); correctness aspects of ADIOS (e.g.: memory management) and robustness (e.g. to file system failures) will be pursued.

Tiago Ribeiro worked on the NEMOFFT project. The main purpose of the project is to remove, or at least alleviate, a parallel scalability bottleneck of the global gyrokinetic ORB5 code, in its current electromagnetic version NEMORB. This code solves the Poisson equation and Ampère’s law in Fourier space and relies on filtering to refine the physical quantities. A parallel two-dimensional (2D) Fourier algorithm, consisting of two 1D Fourier transforms (FFT) interleaved with a distributed transpose, is used. This algorithm requires large amounts of grid data to be transposed across processors and naturally impairs the code’s parallel scalability.

In order to ameliorate this bottleneck, NEMORB’s 2D fast FFT algorithm was modified to use the Hermitian redundancy inherent to its purely real input data. This allowed to reduce to roughly one-half the size of the first Fourier transformed direction, from N complex Fourier modes kept for a N-size real dataset, to keeping only $N/2+1$ of these modes. This resulted in more efficient computation of the 1D FFTs by a factor of 2. Nevertheless, because the Hermitian-reduced number of Fourier modes is an odd number and the number of processors is an even number, padding extra data elements with zeros is necessary for the transpose algorithm to work. If for small number of processors the number of zero-padded elements is small, thus allowing for speedups factors of 2 also in the cross-processor transpose

operation due the matrix size reduction, for high number of MPI tasks (of the order of toroidal grid nodes for a typical grid size) it becomes significant and that gain is lost.

To address this issue another property of the Hermitian redundancy was used, namely, that for even N, which is the case for NEMORB, the zero (F_0) and Nyquist ($F_{N/2+1}$) Fourier modes are purely real. Therefore, without any loss, the real part of $F_{N/2+1}$ can be stored in the imaginary part of F_0 in what is called a half-complex-packed-format. Converting to this storage format before performing the transpose avoids the need for zero-padding and results in smaller amounts of data to be transposed across-processors, even for high number of MPI tasks. The performance measurements yielded speedup factors of about 2 on HPC-FF for the overall 2D FFT algorithm. They further revealed that, depending on the size of the problem (grid-count), a degradation of the speedup starts to occur when the communication latency becomes significant compared to the total transpose communication cost. This happens whenever the number of MPI tasks becomes too large for small grid-counts. Further improvements to overcome such limitations are already planned. Using different transpose algorithms as well as a hybrid MPI/shared memory segments parallelization scheme is foreseen.

1.2. Further tasks and activities of the core team

1.2.1. Dissemination

Hammer, N. J.: Performance Tuning Using Vectorization, IPP report 5/126, Max Planck Society eDoc Server, [online], 2011.

Kang, K.S.: Multigrid solvers for a Scrape-Off layer domain, *IPP Theory Meeting*, 5th – 9th December 2011, Liebenberg, Germany.

Martone, M.: The library 'librsb' (Sparse BLAS Implementation for Shared Memory Parallel Computers), *RZG seminar*, 24th October 2011, Garching, Germany.

Martone, M.: Parallel I/O for fusion research applications, *IPP Theory Meeting*, 5th – 9th December 2011, Liebenberg, Germany.

Kang, K.S.: The multigrid method for an elliptic problem on a rectangular domain with an internal conducting structure and an inner empty space, IPP report 5/128, Max Planck Society eDoc Server, [online], 2011.

1.2.2. Training

Haefele, M.: Workshop "Masse de données : I/O, format de fichier, visualisation et archivage",13th January 2011, Lyon, France.

Haefele, M.: Ecoles d'été "Masse de données: structuration, visualisation", 26 – 30 Septembre 2011, Autrans (France); contributed with a training session of a total of 13.5 hours.

Hammer, N.J.: Performance Tuning Using Vectorization, *GOTiT e-Seminar*, 22nd February 2011, Garching, Germany.

Hammer, N.J.: Performance Tuning Using Vectorization, *RZG Seminar*, 21st February 2011, Garching, Germany.

1.2.3. Internal training

The HLST core team has attended:

- The HLST meeting at IPP, 8th March 2011 and 18th October 2011, Garching, Germany.
- Arbeitsgemeinschaft Simulation (ASIM) workshop, 22th – 24th March 2011, Garching, Germany.

Kab Seok Kang has attended:

- IPP Summer University on Plasma Physics and Fusion research, 26th – 30th September 2011, Greifswald, Germany.

1.2.4. Workshops & conferences

Hatzky, R., Mishchenko, A., Könies, A., Bottino, A., Kleiber, R., and Borchardt, M.: The numerics behind electromagnetic gyrokinetic particle-in-cell simulation – the cancellation problem resolved, *14th European Fusion Theory Conference*, 26th – 29th Sep. 2011, Frascati.

1.2.5. Publications

Kleiber, R. and Hatzky, R.: A partly matrix-free solver for the gyrokinetic field equation in three-dimensional geometry. *Computer Physics Communications*, **183**: p. 305–308, 2012.

Kleiber, R., Hatzky, R., Könies, A., Kauffmann, K., and Helander, P.: An improved control-variate scheme for particle-in-cell simulations with collisions. *Computer Physics Communications*, **182**: 1005 – 1012, 2011.

Mishchenko, A., Könies, A., and Hatzky, R.: Global particle-in-cell simulations of plasma pressure effects on Alfvén modes. *Physics of Plasmas*, **18**: Art. No.012504, 2011.

Bottino, A., Vernay, T., Scott, B.D., Brunner, S., Hatzky, R., Jolliet, S., McMillan, B.F., Tran, T.M., Villard, L.: Global particle-in-cell simulations of microturbulence in tokamaks: finite-beta effects and collisions. *Proceedings of the EPS 38th conference on plasma physics, Strasbourg; Plasma Phys. Control. Fusion*, **53**: Art. No. 124027, 2011.

1.3. Recommendations for the year 2012

The call for the use of High Level Support Team resources will close on 31st January, 2012. 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. Final report on HLST project ASCOT-11

We had some issues with the copyright and the license of the FIPC module that we got from Ian Bush [1]. We contacted the concerned person at Daresbury Laboratory, referred by Ian Bush, informing him about our wish to continue using the module. Unfortunately, we did not get any reply from them. However, Ian Bush has been working on a new version of the module, which is available under the Lesser GNU Public License (LGPL). This was provided to us and we passed it onto the ASCOT group.

The new module was tested along with the spline routines from Numerical Recipes and the tests were successful. Other sanity tests such as use of correct data types in the routines creating the Shared Memory Segments (SMS), creation and freeing of segments etc. were also successful. The module supports up to seven dimensional SMS.

The ASCOT group is still in the process of a major restructuring of their code. As a result we have not had any queries from them specific to the FIPC module. They will be using the library for production runs in the future. We have, meanwhile, completed and submitted a document on the FIPC module to Ian Bush. He will use this document to make the module available, on SourceForge, for public use.

2.1. References

[1] Shared Memory Segments, HLST Core team report 2010.

3. Supplementary report on HLST project EUTERPE

3.1. Restart files improvement

EUTERPE is a gyrokinetic code currently developed at IPP Greifswald. Its original restart file mechanism required to restart the code on exactly the same number of cores as the previous simulation that generated the restart files. Even if that is reasonable in most cases, it happens that restarting on a different number of cores can be of interest when available batch queues sizes change. The purpose of this work is to make the restart mechanism more flexible in order to be able to restart the code on a different number of cores.

When the allocated execution time for the simulation is coming to an end, the simulation writes restart files. These files consist of the necessary information in order to be able to restart the simulation at the present timestep and go further in time within the next job. Since the simulation is a parallel application, data is distributed over several MPI processes. So each process writes its own local data into a file using a POSIX I/O call.

The reading in of the restart files consists of two phases: header and data reading. The main program calls the header reading phase first in order to get the data size contained in each file. Then it allocates the necessary memory to hold these data and finally calls the data reading routine, which actually reads the files and fills the allocated memory. The previous implementation was straightforward. Since the code restarted always with the same number of cores as in a previous simulation, each core had to open its own file and read both its header and the data. In order to allow a flexible number of cores for restart, these mechanism had to be redesigned.

To simplify the restructuring of the reading phase, the header reading mechanism has been serialized. The process of rank 0 reads now the header of all the files, computes the size information for each core and distributes this information using MPI communication routines. As the headers are a small amount of data, the impact of the serialization on the overall performance can be neglected. The data reading phase has been kept parallel for performance reasons. In the following, we assume that the number of clones in the simulation is increased or reduced by a factor which is a power of two. Now, according to the number of cores used by the simulation that has generated the restart files, two cases can arise.

Number of cores has been reduced

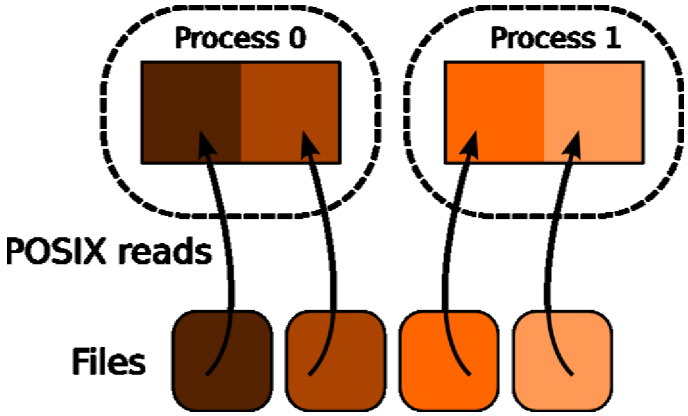


Fig. 1 Restart principle with a reduced number of cores

Fig. 1 shows the principle when the number of cores has been reduced or keeps the same. There, each MPI process has to read the data of one or more files. If the number of cores is reduced, the total amount of available memory is reduced as well. A memory limitation can occur in this case. It is up to the user to check whether the amount of memory needed by the previous simulation can fit into the memory available in the restarted simulation. Then, each file is consecutively opened and its contents appended to the single memory chunk of the process. As the merged data needs to be sorted for performance reasons, a sorting routine is called afterwards.

Number of cores has been increased

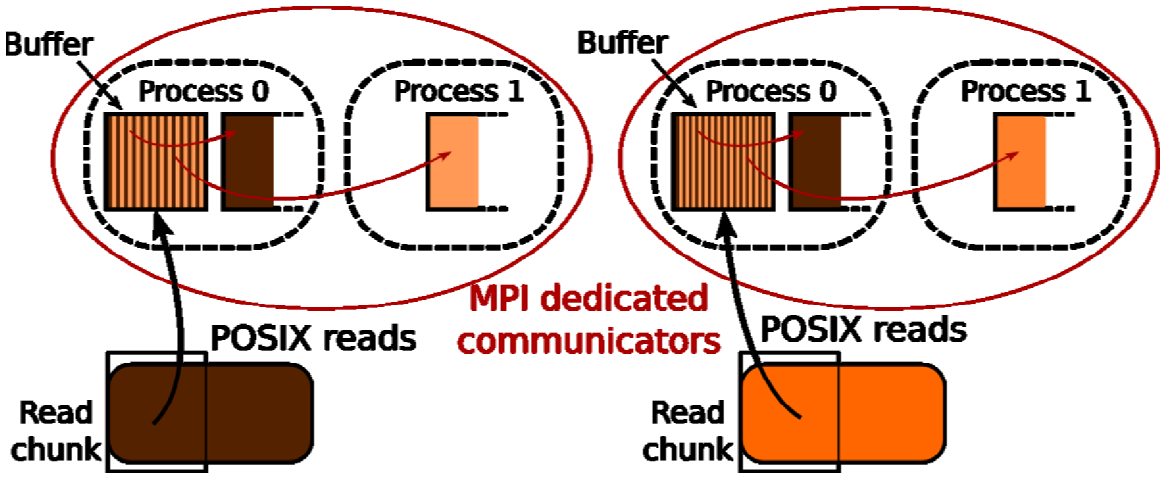


Fig. 2 Restart principle with an increased number of cores

Fig. 2 shows the case when the number of cores has been increased. Here, each file has to be split into two or more parts. Dedicated MPI communicators are created for that purpose, one for each file. Within these communicators, processes of rank 0 allocate a buffer, read the file and distribute it to the processes that belong to the communicator. Since a sorting algorithm was applied on the data before writing the restart files, the data in the files are already sorted. The computational cost associated to a particular particle depends on the position in the file. The particles related to the data stored at the beginning of the file have a higher associated computational cost. So in order to keep a computational work balance among the processes, the buffer is not split in contiguous chunks but instead a stripe pattern is applied. In Fig. 2, each file has to be split into two parts. One particle out of two is

sent to one process, the complementary particle to the other process (see striped structure in the buffers of Fig. 2). In the general case, each file has to be split into *N* parts, one value out of *N* is sent to each process. This distribution algorithm keeps the data sorted and thus guarantees a load balance between the MPI processes.

In principle, a memory shortage can occur, due to the additional allocated temporary buffers needed to read in the files. Indeed, if the previous simulation used the maximum amount of available memory, and if the file is read in one piece, then the additional temporary buffer would need already the whole memory and no memory would be left for the data structures. To overcome this issue, each file has to be read by chunks, thus limiting the memory needed for the buffer. The size of the chunks is a parameter in the *part* module of EUTERPE. It has to be a power of two and is currently set to 2^{23} Bytes = 8 MByte. This value is small enough in order to avoid a memory shortage and to guarantee a high-performance read/write of large files.

In summary, the EUTERPE restart file mechanism has been made more flexible. The code can restart on a different number of cores as long as the ratio of the old number and the new number of cores is a power of two. The size of the chunks in which the data are read and written is the only additional parameter, and it has been introduced in order to overcome possible memory shortage.

3.2. Performance improvement

Unfortunately, the vectorization of the dominant routines in the EUTERPE code did not lead to the expected performance improvement on HPC-FF. Hence, it was necessary to investigate if there was, some other way for obtaining a performance improvement. In the past, the EUTERPE code has already been subject to performance optimization as a project of the DEISA initiative. At that time a significant performance increase by a factor of two was achieved. Thus, it was difficult to find a further major performance bottleneck. Instead it became clear that only the sum of several small improvements could give us a further significant improvement. In the following we will give an overview of our achievements:

1. A detailed performance analysis revealed that the code was memory bound, i.e. the memory access speed limited the increase of the FLOP rate. It showed that especially the calls to the look-up tables for the equilibrium quantities of the magnetic field were quite costly. As a consequence, now the relevant equilibrium quantities needed in a certain subroutine are copied in special work arrays at the initialization. Hence the input data amount is reduced just to the necessary and the locality of memory accesses is now better.
2. Instead of calling separate routines for each equilibrium quantity, all equilibrium quantities at the particle position are calculated within one sweep from the input data.
3. IF statements have been moved from the inside of loops to the outside by duplicating the loop structure.
4. Instead of precomputing certain B-spline quantities and storing them into look-up tables it is more efficient to recompute them when needed.
5. The diagnostic output data are not calculated anymore in separate one-dimensional arrays. Instead, a single two-dimensional array is used.
6. The FFT's from the Intel MKL seem to be slightly faster than the FFTW library routines.
7. A temporary array is now used for the charge- and current-assignment which has the index order (k,j,i) instead of (i,j,k), which speeds up the assignment process.
8. The charge- and current-assignment for ions and fast ions is done in one sweep instead of using separate calls. Accordingly, a temporary array with an extra dimension is used.
9. Instead of using assumed shape arrays in FORTRAN 90 it is more efficient to use explicit-shape arrays if there is some information available about the size of the

array at compile time. This becomes even more efficient if the innermost indices are declared as constants.

10. Unrolling innermost loops by hand can help if several nested loops are used.

The execution time has been reduced by nearly 30%. As a result the FLOP rate is now in some test cases more than 2.8 GFLOPS/core which is 24% of the peak performance of HPC-FF. This is an excellent result for an application on a RISC architecture.

In addition, a scan of EUTERPE with the newest FORCHECK v14.1 has been done. Some inconsistencies have been corrected. The performance changes triggered a compiler bug of the IBM XLF11.1.0.7 on IBM BlueGene/P which had to be identified as such. A problem report has been submitted to IBM.

Finally, to further improve the data locality we would like to recommend to use REAL*4 equilibrium data instead of REAL*8 as it is now the case. This would increase the effective size of the cache architecture by a factor of two. According tests have to show if there is any negative impact on the total precision of the simulated data.

4. Report on HLST project GYNVIZ

The aim of the project is to unify and to provide support for the whole hardware and software chain that results from the turbulence codes' output to the interactive and remote visualization software. Fig. 3 presents a global picture over the proposed project.

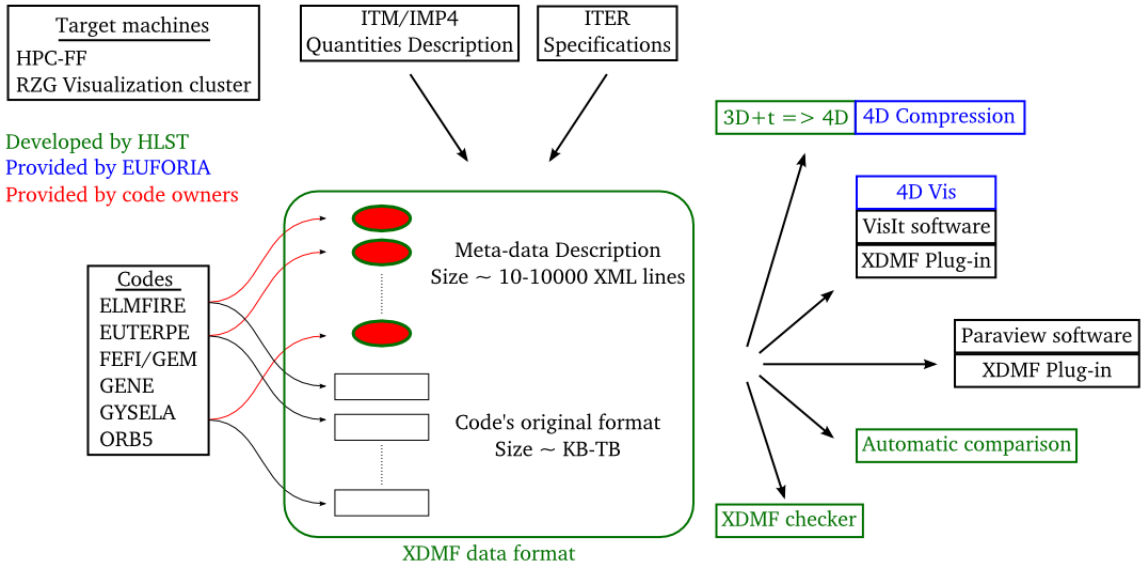


Fig. 3 Overview of the overall project

4.1. Reduction of the project scope

At the very beginning of the project, we had to make a choice among two possibilities concerning the data format: to design a new dedicated format from scratch or to use an existing one. 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 for the extent of a fixed time frame, software maintenance is a big issue once the project has ended. But on the other hand, to design our own format keeps us independent from 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.

The review of different existing file formats showed that XDMF had the major advantage not to imply large modifications in the IO procedures of the individual turbulence codes. But it showed also that XDMF had its weaknesses. Thus, we got in contact with the XDMF development team and after an encouraging discussion we started to collaborate in March 2010. Our intention was to write a XDMF checker to help users with the migration of their current format to XDMF. In addition, we had in mind to become an alpha tester for the new XDMF library. Our intention was that the XDMF team would improve, develop and maintain the XDMF library even after the end of the GYNVIZ project.

Unfortunately, this collaboration failed in the middle of this year 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 anymore. Finally, at the

beginning of this summer we decided not to wait any longer for the XDMF official release with the consequence of abandoning the XDMF technology in 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 be further extended 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 spent effort, the loss is tolerable. Only the XDMF checker, developed exclusively by us for XDMF, took a significant amount of development (approximately 5 weeks) which became now useless. 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.

4.2. GYNVIZ tool development

4.2.1. 3D+t → 4D conversion tool: gyncompress4d

This 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 comes from the need to handle a large variety of domain decompositions of 4D data. Namely, 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 chose to implement 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. $N = 2^n+1$, 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.

The entry point of such a tool is the description of the underlying domain decomposition. Originally, the XDMF language should have been used for that purpose. Instead a dedicated ASCII file will replace 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/tmp/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/tmp/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 will be built in memory. Indeed, a large 4D data 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 single precision floats. The library exports directly the 4D compressed data once the compression algorithm routine terminates.

The development of the *gyncompress4d* conversion tool is now finished and tested. Some last production tests on real data coming from the different turbulence codes are still ongoing.

4.2.2. 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. The plug-in has also been successfully ported to the current up-to-date version of VisIt (2.2.2). 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. The licensing issues have also been solved. The EUFORIA software we are using is now distributed under the open source Cecill-B license.

4.2.3. Data transfer

As the turbulence codes run on HPC-FF, the data is located on HPC-FF. In order to be visualized on the visualization cluster at Rechenzentrum Garching (RZG) the compressed data have to be transferred from Jülich Supercomputer Center (JSC) to RZG. The dedicated 10 Gb/s DEISA network is used for that purpose. Initially, it was planned to use directly the DEISA file system to provide the transfer but as the DEISA project ended in April this year and JSC was not willing to support it anymore, 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. The little wrapper around *bbcp* named *gyntransfer* is provided to help the user in using appropriate optimization options.

4.3. Conclusions and future work

The GYNVIZ project is now approaching its end. Unfortunately, the scope of the project had to be narrowed 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 focus has been shifted into bringing the 4D visualization functionality to a mature production state which has been obtained. Some last production tests with realistic datasets are still ongoing. In the remaining project time we plan to provide all GYNVIZ users with dedicated support in order to train them on how to use our tools.

5. Report on HLST project ITM-ADIOS

5.1. Introduction

Lengthy, computationally intensive plasma physics simulations require periodic saving of intermediate results. There can be several motivations for this: prevention of data loss due to accidental hardware or software failures, the desire to have the ability to monitor/process the results of an ongoing simulation (diagnostics) or to allow a simulation to evolve beyond the time span allowed by the batch system.

However, a consequence of an ever increasing ratio between computing power to Input/Output (I/O) subsystems throughput is that a relevant fraction of (wall clock) time is consumed by "waiting" for I/O operations to complete. Adoption of "parallel I/O" techniques may reduce this time by increasing the I/O throughput, and thus also the efficiency in resources usage. Also techniques overlapping I/O and computation, like "staging" techniques, are known to improve I/O efficiency (usually at the cost of using additional hardware).

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, with the aim of obtaining a better I/O efficiency in plasma physics simulations performed on the HPC-FF computer. A benchmark suite with “kernel I/O” programs will be provided to test the capabilities of the ADIOS library. An assessment will be done in competition with classical parallel I/O strategies as provided e.g. by the “Message Passing Interface” (MPI) library, in combination with the classical “Portable Operating System Interface” (POSIX) Application Programming Interface (API) usage (see the HLST Annual Report 2010 document [1, p.12] for studies performed so far on this topic by Matthieu Haefele).

If ADIOS testing shall give us promising performance results, its integration into one or more well established simulation codes will be tackled. However, also other aspects of the ADIOS library are of interest as e.g. the user friendliness of the installation process and usage. Fault tolerance (for data integrity) and the overall health of the project itself are additional topics of our investigation.

The following sections will illustrate with some degree of detail the activity performed so far, while the last section will give a perspective on what we see as next steps in this project.

5.2. The ADIOS Parallel I/O library

ADIOS [2] is the outcome of a cross-institutional effort, primarily led by the Oak Ridge National Laboratory (ORNL) and the Georgia Institute of Technology (GIT). The most appealing aspect of ADIOS is that it can be easily integrated with MPI-enabled parallel applications. ADIOS is meant to offer parallel I/O capabilities to MPI-parallel programs having only serial I/O; thus increasing the overall throughput in disk I/O operations. For this purpose, ADIOS can read/write data files using different parallel file format libraries in a transparent manner.

A notable application of ADIOS (according to [3, p.29] and [6]) has been the management 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 preliminary impressions with the library are positive: the source code archives are distributed together with a collection of sample programs which exemplify the library usage in the most common ways. The user manual ([3], about 100 pages) covers the various aspects of ADIOS usage. Besides the API and the detailed

description of the XML configuration file it offers an extensive commentary of 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, it is detrimental to the public user that not all of the functionality being described in the manual is publicly available. 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. ADIOS only requires dependencies to one further library, the "Mini-XML" [5] 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, which makes it usable with the “shell modules” system. Configuring and building phases of the ADIOS library itself did not pose any problems. As there are several options available at installation level, a future reinstallation of the ADIOS library could become appropriate to achieve improved performance or enable additional features.

After installation, we have been successful in executing example and test programs distributed in the library sources archive. We describe our further ADIOS related activities in the next sections.

5.3. Setting up an ADIOS benchmarking environment

As our first ADIOS related activity, we have started developing a flexible benchmark program (we will refer to it as "IAB", short for "ITM-ADIOS Benchmark") in order to get accustomed with the different ADIOS features available to us. Since we were using a pre-release ADIOS version we have received from the authors, we took care of using the API in the style recommended by them. IAB consists of a "core" benchmark program written in a compiled language and a number of scripts. The scripts are used to drive the core program to run various experiments on HPC-FF, collect performance data, and post-process results.

We have chosen to implement the benchmark program core in the C language, rather than the traditional numerical computation programming language Fortran. There are two main reasons behind this choice. First, because the available 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. We hope that ADIOS developers will make Fortran modules available for this purpose in the future.

So far, we have been able to implement (and gain confidence with) the following ADIOS features: single or multi dimensional global/local arrays, writing to a number of files different than the number of MPI tasks, ADIOS buffer control and specification of program data without the XML-file interface.

The "IAB" benchmarking suite is of fundamental importance to our investigation, as it is being adapted to simulate different ADIOS usage scenarios, under different external conditions (mainly, filesystem related), as well as collecting performance statistics in a handy manner. We are continuously modifying the benchmark as we progress with our knowledge of ADIOS, especially when testing/benchmarking ADIOS based solutions using features requested by the ITM users we are in contact with. We expect to continue IAB's development in pair with our investigations.

5.4. HPC-FF File system characterization

In order to proceed with properly parallel I/O on the parallel file system we have on HPC-FF, we felt it was necessary 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” filesystem is based on LUSTRE; therefore a serial program interacting with it still benefits from the higher throughput due to having a parallel filesystem. Additionally, LUSTRE gives the user the possibility of specifying the "striping semantics" of a file, thus influencing its I/O throughput.

In this context, we have developed (or used) a number of trivial serial programs using the strictly serial I/O API of Fortran and C. Additionally, we developed companion scripts for running these programs, collecting their performance data and producing graphs, out of a variety of experiments on them. These programs are instructed to perform I/O of the same data (in terms of both size and contents), but with a different approach regarding LUSTRE-independent parameters as write block size, number of blocks, filesystem synchronization options, and language. This type of experiment is necessary, as the semantics of basic I/O constructs in C and Fortran languages differ; prominently, the Fortran runtime library buffers data before effective I/O, while C is unbuffered, and thus closer to the operating systems interface.

From our preliminary experiments running these programs, we notice that Fortran's buffering mechanism masks considerably the program's I/O calls to the I/O nodes: this has the outcome of preventing us from exercising control over when to perform effective I/O (by effective, we mean when the operating system syscalls are invoked by Fortran's runtime library). Its outcome (in terms of performance) can be good or bad, depending on the particular case, but it generally mitigates extremes. On the contrary, C's standard API I/O semantics does not specify buffering, and the effective I/O pattern influences performance considerably. We will investigate into methods for exercising more control over buffering and "block writing" in Fortran, be it using available language constructs, or non-standard compilers extensions.

By studying these baseline cases, we already found that writing a non-trivial amount of data to disk (e.g.: a hundred MBytes), the gap between the fastest and the slowest approach can be an order of magnitude high.

As an example, see Fig.4, illustrating the different I/O throughput by invoking the Unix "dd" program, by varying LUSTRE parameters only. We have chosen "dd" because it invokes C's I/O routines (and thus, the operating system's I/O calls) in a manner which can be precisely controlled. In this experiment, we drive the program into writing a specified amount of data to a single file, but with different LUSTRE parameters. In addition, we run "dd" with a strict data synchronization option; thus, preventing the file system caching on the operating system side.

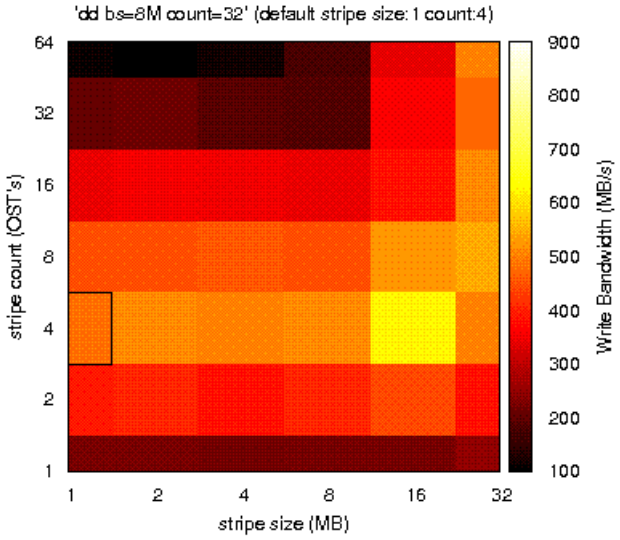


Fig.4 Throughput while writing a file using “dd“ on HPC-FF ‘s $WORK filesystem, for different LUSTRE parameterizations. Notice the default one (4 for “*stripe count*“ and 1 MB for “*stripe size*”) leads to a sub-optimal performance, while the best one has been achieved changing the stripe size parameter to 16 MB.

With this experiment we have attained (at most) around 800 MB/s, and we regard this value as (or near to) a reasonable best case using serial I/O. Without the synchronization option, we have been able to achieve around 1 GB/s. It will be interesting to compare these values against the best attainable via truly parallel I/O (that is, via a program with multiple writing processes). Such an estimate may be of paramount importance when facing the decision whether to take or not the burden of changing the I/O style of an existing application code from a serial one, to a parallel one.

5.5. ADIOS on HPC-FF

Using the ADIOS benchmark program described in Sec. 5.3, we worked towards reaching as much I/O speed as we could with a moderate effort (no thorough investigation). This experiment has been performed under a normal user account using the batch scheduling system and half of the machine’s processors. Under specific conditions, and using the maximally allowed number of MPI tasks (4096), we have been able to reach approx. 14 GB/s. According to HPC-FF's system administrators, this is almost 75% of what the system was capable of obtaining under ideal conditions during acceptance testing. We regard this result as very encouraging and don't expect to get more than around 20 GB/s.

Fig. 5 shows the aforementioned "best run". We notice that the overall best performance here was achieved when writing the data from all of the MPI tasks to only 64 files. Due to the complexity of the system at hand, we are not aware of the exact reason for this particular value. However, we shall investigate into the repeatability of these experiments, as well as locating such "optimal" parameterization for other setups also (e.g.: different number of overall MPI tasks, or different total amount of written data).

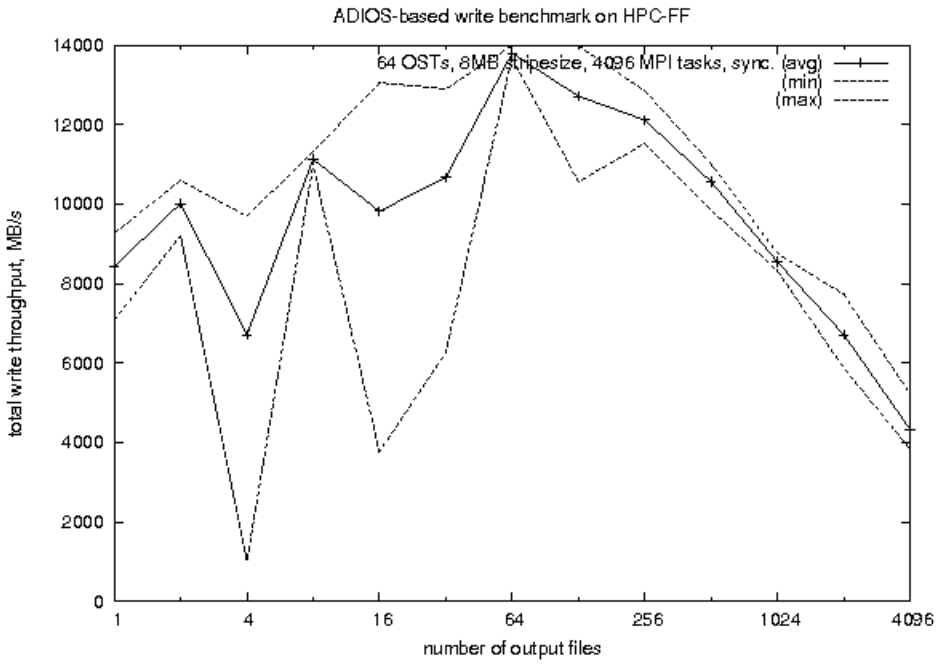


Fig. 5 Throughput while writing array data to an increasing number of files, while keeping fixed parameters as number of parallel MPI tasks, array data amount per task, LUSTRE filesystem parameters, and so on.

However, this result is not by any means definitive: it marks only a step in the course of our investigations. A very important aspect yet to study is whether the same performance can be matched/emulated by integrating ADIOS in the ITM codes at hand (e.g.: Bruce Scott's GEMZA, Alberto Bottino's NEMORB): the I/O needs of these programs may differ slightly, thus leading to adoption of a different solution, and with that, potentially a different maximal performance. We plan such an integration, as it will not only show these programs interaction with ADIOS, but will also serve as a starting point with the ADIOS API for the code authors themselves.

Other aspects of our future investigation in the ADIOS library may regard the correctness, robustness, tolerance to bad programming practices or system errors. It would also be interesting to determine whether other approaches (e.g. a bare MPI-IO approach, or using the SIONlib library [4]) may prove comparably transparent and portable.

5.6. Contact/Collaboration with ITM code developers

A sub-goal of this project is to effectively transfer the knowledge we acquired during our investigations to interested parties within the plasma physics community. Therefore, regular contact with code developers is being kept. In this way, they are able to follow the different aspects of our investigation and have the chance to comment on them; in turn, we are using their comments/enquiries to actively shape our investigation to their needs. An advantage of this approach is that it will not be necessary to have a specific, intensive "knowledge transfer" session, as expertise is being shared on a regular basis. Moreover, transferring to them ADIOS-related "hands on" knowledge as early as possible shall lead to an earlier discovery of eventual incompatibilities/problems in our current "best practices" in ADIOS usage. For these reasons, we are in the course of instructing code developers into autonomous use of the IAB benchmark code on HPC-FF. Moreover, we established a mailing list for I/O topics on HPC-FF which is now used when confrontation of results/practices is needed.

5.7. Conclusions, roadmap for further work

Our investigation was conducted by the question of what the potential ADIOS benefit would be on the HPC-FF machine. To answer this question, an assessment of the maximum bandwidth achievable when writing from a serial program was done. Using the C language I/O from a serial program, but tuning appropriately the (user side) LUSTRE filesystem “striping” parameters, we have been able to obtain at most around 1GB/s.

Using ADIOS in a parallel environment it was possible to obtain at most 14 GB/s when writing; that is around 75% of the system capacity. In detail, array data from 4096 MPI tasks was written to a subset of 64 files. When using less files, we have seen performance penalties of 40%. When using as many as 4096 files, the penalty was around 70%. Nevertheless, the speedup by a factor of approx. 15 between “best cases” of serial and parallel ADIOS output is very encouraging on HPC-FF and it seems to justify the overhead of porting applications to ADIOS.

As there are several factors influencing I/O performance, we have yet to characterize their role in subsequent experiments, and thus identify “best practices”, both with and without ADIOS. As a complementary activity, we would like to evaluate alternative technologies (e.g. SIONlib [4], "bare" MPI-IO [8], etc.) allowing us to achieve similar performance results, as well as satisfying other constraints (e.g. interoperability, ease of use, generality, portability, etc.).

We expect that in the next months we will get into a closer cooperation with our users, as they will be increasingly able to experiment by themselves with the technologies we are investigating into (we are adapting their GEMZA and NEMORB computational codes to work with ADIOS); this, while relying on us for troubleshooting and best practices.

To document an ADIOS-based equivalent of Matthieu Haefele's experimentation with MPI-IO usage on HPC-FF (see [7] or [1, p.12] for his report) is also in our broader intentions.

Since availability of the next-generation machine for fusion computations (IFERC) is expected beginning 2012, and because of the similarity of this machine's architecture to HPC-FF's, we will try to make our findings general enough to be extended/usable to the new platform as well.

5.8. References

[1] EFDA High Level Support Team; *HLST Core Team Report 2010*; https://www.efda-hlst.eu:445/internal/reports/annual-hlst-report-2010/reports-ofthe-projects-2010/hlst-core-team-report-2010/view
[2] *ADIOS Project Home Page;* http://www.olcf.ornl.gov/center-projects/adios/
[3] S.Hodson, S.Klasky, Q.Liu, J.Lofstead, N.Podhorszki, F.Zheng, M.Wolf, T Kordenbrock, H Abbasi, N.Samatova; *ADIOS 1.3 User Manual*; http://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.3.pdf
[4] *SIONlib library home page* http://www2.fz-juelich.de/jsc/sionlib/
[5] *Mini-XML Project Home Page;* http://www.minixml.org/
[6] 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
[7] Matthieu Haefele; *Comparison of different Methods for Performing Parallel I/O*; http://edoc.mpg.de/get.epl?fid=72188&did=498606&ver=0
[8] *MPI-I/O documentation* http://www.mpi-forum.org/docs/mpi22-report/node261.htm

6. Final report on HLST project KinSOL2D

6.1. Introduction

The Particle-in-Cell (PIC) code BIT1 is restricted so far to 1D3V (1-dim real space + 3-dim velocity space) plasma and 2D3V neutral particle modeling with a reasonable scaling beyond a thousand cores. The ongoing work is focused on the enhancement of the code in real space to 2D3V plasma simulations of the Scrape-Off-Layer (SOL). The increase of the dimensionality of the code to 2D or even 3D seems to be straight forward. However, the Poisson solver in 2D has been identified as a bottleneck for scaling; furthermore, there was even no discretization available at the beginning of the project. It is mandatory that this part of the code should scale to very high core numbers in order to maintain the good scaling property of the whole code. So the work plan is to develop a good scaling Poisson solver in 2D. Possible candidates as solvers are a multigrid solver, or depending on the type of the matrix, a preconditioned Conjugated Gradient Method (CGM) and Generalized Minimal Residual Method (GMRES), respectively. A combination of both is also thinkable where the multigrid method is used as a preconditioner for either the CGM or the GMRES method. In contrast, a parallel direct solver doesn't seem to be a good choice as it usually doesn't scale as efficiently to large numbers of cores and its memory consumption is much larger compared to iterative methods.

6.2. Assessment of potential solvers

The GMRES method has to be used for non-symmetric or non-positive definite systems which can arise e.g. through the boundary condition treatment. In general, the preconditioned system of a symmetric system is not symmetric for the same inner product. However under certain conditions such a system can be symmetric in a different inner product (*A*-inner product or energy inner product) and the less costly CGM method can be used.

The multigrid method is a well-known, fast and efficient algorithm to solve many classes of problems including the linear elliptic, nonlinear elliptic, parabolic, hyperbolic, Navier-Stokes equation, and Magnetohydrodynamics (MHD). Although complex to implement, researchers in many areas think of it as an essential algorithm and apply it to their codes because the complexity of the multigrid method is only $N\ log(N)$, where N is the degrees of freedom (DoF).

To implement and analyze the multigrid method, we have to consider two main parts of the multigrid algorithm: the intergrid transfer operators and the smoothing operator, separately. The intergrid transfer operators depend on the discretization scheme and are highly related with the discretization of the matrix. The smoothing operator will be implemented according to the matrix-vector multiplication. So we have to determine the appropriate discretization method which includes the generation of the matrix and an efficient implementation of the matrix-vector multiplication.

If a multigrid solver converges, it usually does so very fast, which is the case on most problems. However, the multigrid method as a solver does not guarantee convergence. In contrast, iterative Krylov subspace methods, which include the CGM and GMRES, guarantee convergence and can be further improved by preconditioners to speedup the convergence rate. The multigrid method is also well-known to act as a very efficient preconditioner. The preconditioned CGM can be used only for symmetric and positive definite problems and has to use the A-norm instead of the L^2-norm. The preconditioned GMRES works for non-symmetric or non-positive definite problems, but needs more working memory. To reduce the working memory in GMRES, we can use Restart GMRES which does not strictly guarantee the convergence, but converges for most problems.

6.3. The model problem

We consider the second order elliptic partial differential equation with the coefficient ε(x,y) being defined on a rectangular domain with an internal conducting structure as in Fig. 6. As boundary condition, we have to consider for the outer wall the Dirichlet zero boundary condition, $\Phi_w(x,y,t) = 0$, the Neumann zero boundary condition for the inner empty surface, $E_n(x,y,t) = 0$ and the Dirichlet boundary condition for the internal conductor, $\Phi_c(x,y,t) = \Phi_c(t)$.

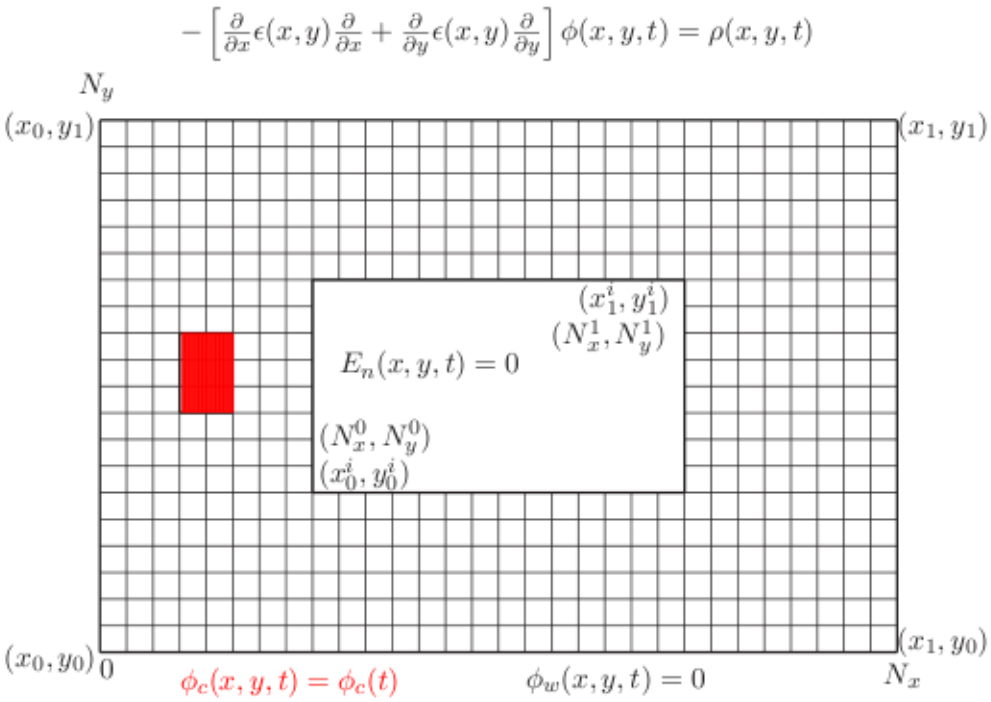


Fig. 6 Rectangular domain with an internal conducting structure [■: internal structure (conductor)]

We use the finite difference method as discretization method. In addition, we use a structured mesh with ε(x,y) being defined on each cell boundary. The coefficient ε(x,y) is time dependent and will change for each time step of the simulation.

As the model problem is quite complex, it is advantageous to split it first into subproblems which can be solved separately. After successful treatment of the subproblems they can be used as building blocks to assemble an algorithm for the full problem.

6.4. The multigrid software framework

To prevent starting each new multigrid project from scratch we have started to develop a multigrid software framework. So far a preconditioned CG, a preconditioned GMRES and a multigrid solver have been implemented. In addition, the multigrid method can be used as an efficient preconditioner. The discretization method has an impact only on the matrix-vector multiplication and the construction of the matrix. Special versions of the vector multiplication for finite volumes, finite elements and finite differences have been provided. In this framework, parallelism is introduced through the matrix-vector multiplication operation. Corresponding tests have been done to prove the correctness of the implementation. For the discretization with finite differences considered here with a Neumann boundary condition, the generated matrix is non symmetric, so one cannot use the CG method. Instead, either the preconditioned GMRES or the multigrid method has to be used.

The implementation of the parallel matrix-vector multiplication and the smoothing operators, in our case Jacobi and Gauss-Seidel, is highly depending on the domain handling of the parallelization concept. In our case we discretize the whole rectangular domain in each direction with a uniform mesh and divide the rectangular domain in n_x by n_y small rectangular subdomains which are handled by one core each. If a rectangle subdomain is in the inner empty space, we do not need to handle this domain and thus do not assign a core to such a subdomain. Hence, we can request fewer cores from the batch system which is quantized in multiples of 8. As a result, the number of idle cores will always be below 7. In the future this number can be further reduced but at the moment it seems to be justified by the fact that it is only a small fraction of the total number of cores. For the internal conducting structure no exception is made so that all subdomains are assigned to a core independent of wether they include parts of the internal conducting structure or not.

6.5. Test with internal conducting structure

The internal conducting structure is an obstacle for the multigrid method in two ways. On the one hand, the corners of the internal conducting structure and/or inner empty space are a challenging problem. And on the other hand, an alignment of the inner structures with the grids of the multigrid V-cycle can not always be guaranteed. To investigate this effect we assume that the conducting structure is arbitrarily shifted relative to the coarsest mesh of the multigrid method.

We test the correct discretization of the elliptic problem on a rectangular domain with an internal conducting structure and an inner empty space by comparing the converged numerical solution with the known, exact solution. To construct an exact solution for a zero Neumann boundary condition on the boundary of the inner empty space, we choose the following sine function f(x,y):

$$\frac{1}{2\pi^2} \sin^2 \pi x \sin^2 \pi y$$

The Dirichlet boundary condition of the internal conductor is given by the values f(x,y) at the boundary of the conductor. For simplicity, we chose for the mesh sizes $h = h_x = h_y$. The corresponding L^2 errors are listed in Table 2. It can be clearly seen that the error converges by second order ($O(h^2)$) when the grid resolution is increased by a factor of two.

level	L^2 error	h	Error ratio
7	0.00005786	0.031250000	–
8	0.00001440	0.015625000	4.0182
9	0.00000359	0.007812500	4.0082
10	0.00000090	0.003906250	4.0039
11	0.00000022	0.001953125	3.9996
12	0.00000006	0.000976562	3.9984

Table 2 The discretization L^2 error and the error ratio between succeeding refinements.

Next we test the multigrid solver and the preconditioned GMRES (PGMRES) method in combination with the multigrid preconditioner. We have implemented the first order intergrid transfer operator and tested its functionality. As smoothing operators we use the Jacobi iteration and the local Gauss-Seidel iteration. These two smoothing operators are relatively simple and well analyzed. The Jacobi iteration does not have a good performance, but does not depend on the number of cores. This is beneficial when testing the parallelized multigrid method because the results do not depend on

the number of cores being used. In contrast the local Gauss-Seidel iteration has a good performance, but depends slightly on the number of cores.

As a test case, we chose a 4 by 4 rectangular domain with a 2 by 2 inner empty space and a fixed finest mesh with $h = h_x = h_y = 0.001953125$ on the finest level 11 and different numbers of coarse levels for the V-cycle. We always chose the inner empty space in such a way that it is aligned with the coarsest grids. In contrast the internal conducting structure can have an arbitrary shift relative to the coarsest mesh.

We sketch two different configurations of the internal conductor area in Fig. 7, one does not match with the coarsest mesh (left) and the other one matches the coarsest mesh (right). The error reduction factors of the multigrid method are listed in Table 3 according to the selected smoothing operator and the number of levels of the V-cycle. The smaller the reduction factor is, the faster the initial residual diminishes and the faster the iterative multigrid algorithm converges. The solution on the coarsest mesh is computed by using the GMRES method.

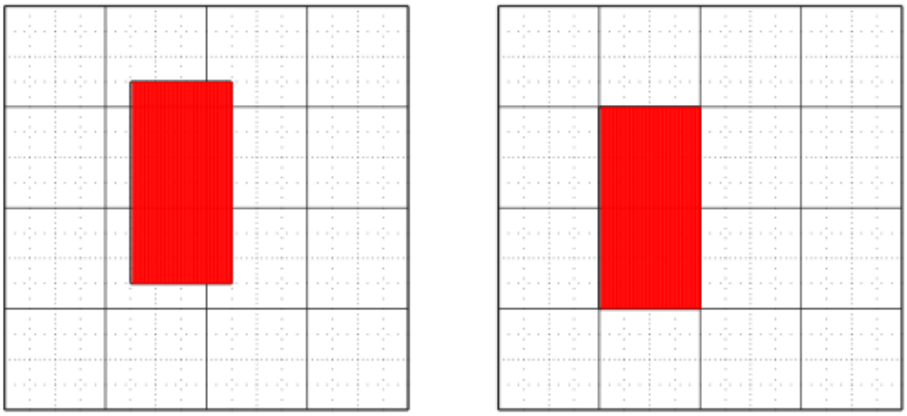


Fig. 7 The positioning of the internal conducting area [■: internal structure (conductor), line: coarsest mesh, dotted line: finest mesh]

	Jacobi iteration		Gauss-Seidel	
levels	MG	PGMRES	MG	PGMRES
3	0.434595	0.153167	0.321513	0.096483
4	0.350831	0.145256	0.295646	0.090466
5	0.750798	0.197123	0.510491	0.129561
6	**	0.207959	0.835621	0.137452

(a) Unmatched internal conductor area. Note, "**" marks a non converged result.

	Jacobi iteration		Gauss-Seidel	
levels	MG	PGMRES	MG	PGMRES
4	0.268014	0.090191	0.131479	0.042970
5	0.290220	0.097606	0.144441	0.049281
6	0.307084	0.109197	0.157332	0.050775
7	0.323019	0.112501	0.168422	0.058537
8	0.318262	0.121409	0.160810	0.066247

(b) Matched internal conductor area.

Table 3 The average error reduction factor of the multigrid method as a solver and multigrid preconditioned GMRES for a different number of levels starting from the finest level 11.

The results show that the multigrid method as a solver and the PGMRES method with a multigrid preconditioner have a very good performance if the internal conducting structure matches with the coarsest mesh. This means that the corners of the conducting structure and inner empty space seem to be no problem for convergence as long as these structures are aligned. Otherwise the convergence rate is significantly reduced. This becomes plausible when one looks at how the internal conducting structure and also the inner empty space are mapped onto the coarser grids of the multigrid V-cycle. From a certain coarser level on, the internal conducting structure is mapped only onto one node point, i.e., its structure can not be resolved any longer by the grid. This seems to be no problem as long as the structure is symmetrically positioned around such a node point. However, if the structure is not aligned with the grid, the information of its misalignment gets lost and it will not be correctly treated on the grids of the coarser levels. In the worst case, this can lead to a non converging multigrid solver result. However, in such a case the multigrid method can be still used as a preconditioner to speedup the convergence rate of e.g. the PGMRES method.

Next, it is interesting to see how the GMRES method behaves without using a preconditioner. For this purpose we compare the solution times of the multigrid solver with the preconditioned PGMRES solver and the GMRES solver for two different mesh spacings of $h_x = 0.0078125$ and $h_y = 0.00390625$ on a parallel 24 core run on HPC-FF. In this comparison we consider the case that the internal conducting structure matches with the coarsest mesh. For the multigrid method we use the local Gauss-Seidel smoother together with a total number of six levels. The corresponding solving times are listed in Table 4 together with the DoF. It can be clearly seen how the multigrid preconditioner speeds up the GMRES method by a factor of 405 for a problem size of 778,323 DoF.

h	# DoF	MG	PGMRES	GMRES
0.0078125	194281	0.449	0.246	17.6
0.00390625	778323	0.901	0.535	216.8

Table 4 The solving time in seconds for a multigrid solver, a GMRES solver with a multigrid solver as preconditioner and a GMRES solver on 24 cores on HPC-FF at JSC.

6.6. Test for a Neumann boundary condition on the inner empty space

The Neumann boundary condition of the inner empty space is quite different compared to the Dirichlet boundary condition of the internal conducting structure. Hence, we have to address the question again about the impact of the proper alignment of the inner empty space compared to the coarsest grid.

Depending on the choice of the finest level on which the problem should be solved, the boundary node points at the inner empty space are not necessarily aligned anymore with the node points of certain coarser grid levels. We illustrate the problem for the 1-dimensional case shown in Fig. 8. The optimal case is shown in (a) where the boundary point *P* is aligned with the grid. This is in contrast to case (b) where no alignment could be achieved. For case (a) a first order implementation of the Neumann boundary condition is already sufficient. However, for case (b) this leads to a very poor convergence rate so we further test a second order implementation which is consistent with the approximation order of our finite difference scheme (see Table 2).

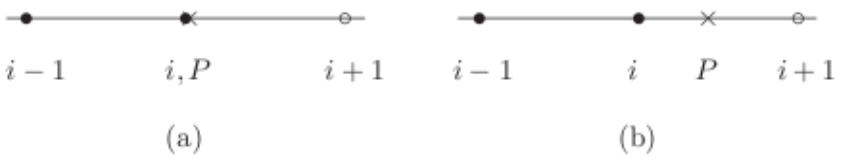
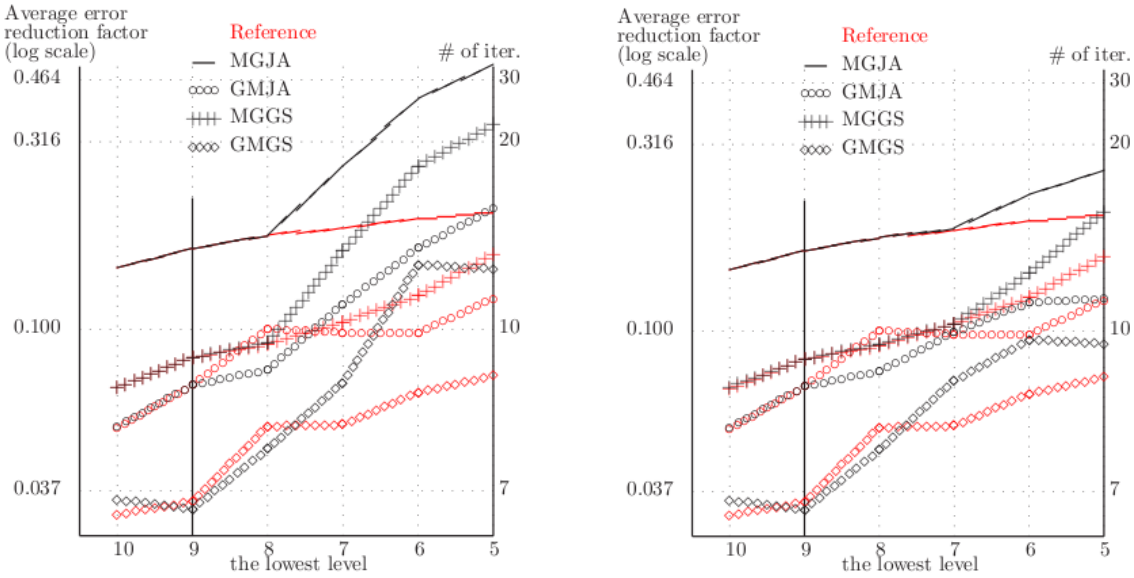


Fig. 8 Notation for the finite difference scheme on the Neumann boundary condition. (●: node points, ×: real boundary points, ○: node points in the empty interior region)

Our model problem for testing the multigrid scheme consists of 10 levels. For level 9 we choose a square domain (0, 4) × (0, 4) with the inner empty space (0.9921875, 3.0078125) × (0.9921875, 3.0078125) which is aligned with the grid. Accordingly the inner empty space will be also aligned with the grid of the highest level 10 but not with the grids of the lower levels 8 and smaller. In addition, we define a reference case with an inner empty space (1, 3) × (1, 3) on level 9 which is aligned with the grid on all levels down to the lowest level 5.



(a) First order scheme (b) Second order scheme

Fig. 9 The average residual error reduction factors and the corresponding number of iterations of the multigrid method as a solver and as a preconditioner for the PGMRES method with Jacobi and Gauss-Seidel smoothers according to the lowest levels for a finest level of 12. Aligned/non-aligned cases in red/black.

In Fig. 9 we show the residual error reduction factor averaged over all iterations and the required number of iterations for different schemes involving the multigrid method both as a solver and a preconditioner. As a termination criterion for the multigrid algorithm we define a reduction of the initial residual error of the finest level by a factor of 10^{-10}. The results are shown as a function of the lowest level on which the problem is solved with the GMRES method. In total we make an assessment of four schemes: a multigrid solver with a Jacobi smoother (MGJA) (solid line), a multigrid solver with a Gauss-Seidel smoother (MGGS) (crosses), the PGMRES method with a multigrid preconditioner using the Jacobi smoother (GMJA) (circles), and the PGMRES method with a multigrid preconditioner using the Gauss-Seidel smoother (GMGS) (diamonds). The results for the aligned/non-aligned cases are plotted in red/black.

As expected the performance of the Jacobi smoother is not as good as the performance of the Gauss-Seidel smoother. Also it is clearly shown that the averaged residual error reduction factor of the PGMRES is smaller than that for the multigrid solver. We also see that the larger the total number of levels becomes, the

larger the averaged residual error reduction factor gets. This is due to the fact that the problem is solved (to high precision) on the lowest level. So the smaller the number of levels in the multigrid V-cycle becomes, the faster the multigrid method converges. However, for the results of the first order implementation of the Neumann condition depicted in Fig. 9 (a) the averaged residual error reduction factor for the non-aligned case degrades significantly compared to the aligned case if the lowest level becomes smaller than the level 8. Instead the second order implementation of the Neumann condition gives better results as can be seen in Fig. 9 (b). Here the averaged residual error reduction factor starts to degrade when the lowest level becomes smaller than 7. In addition, the degradation for the non-aligned case seems to be not as pronounced compared to the aligned case with decreasing lowest levels.

6.7. Test with the full problem

Finally we test the full problem with both the inner empty space and the internal conducting structure. In contrast to the previous section we add an internal conducting structure which is aligned with the grid of level 8 in the *x*-direction and with the grid of level 6 in the *y*-direction. We distinct the following three cases: a full square domain of (0, 4) × (0, 4) without an inner empty space (Full), a square domain with an inner empty space of (1, 3) × (1, 3) (Ha1), and a square domain with an inner empty space of (0.5, 3.5) × (0.5, 3.5) (Hah).

Analogously to Fig. 9 we show in Fig. 10 the average residual error reduction factor as a function of the coarsest level for a finest level of 12, i.e., h = 0.0009765625. Again we have chosen the four methods of the previous section using the multigrid algorithm as a solver or a preconditioner, respectively. In addition we have marked the different geometrical cases with colours: Full in black, Ha1 in red and Hah in blue.

The results in Fig. 10 show that the size of the inner empty space has only a small influence on the average residual error reduction factor. Especially, there is hardly any difference for the multigrid solver with a Jacobi smoother (MGJA) (solid line). However, the execution time for the different geometrical cases can vary significantly as the size of the domain is quite different.

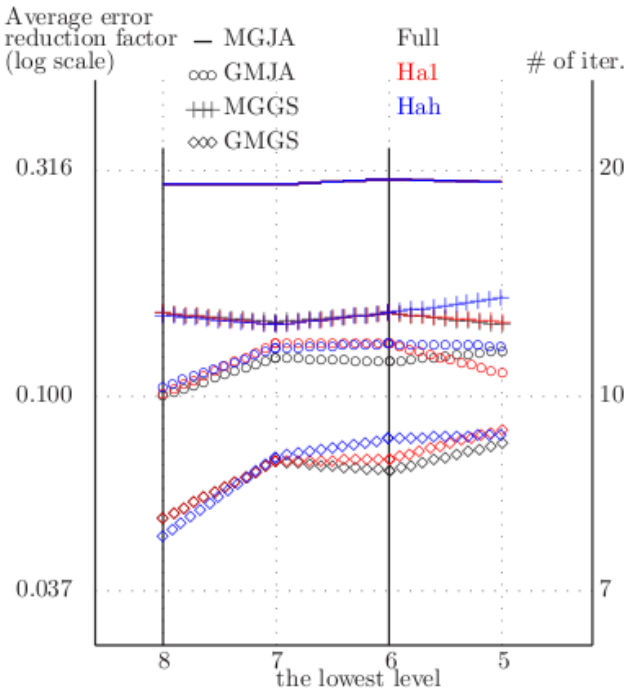


Fig. 10 The average residual error reduction factors and the number of iterations of the multigrid method as a solver and as a preconditioner for the PGMRES method with Jacobi and Gauss-Seidel smoothers according to the lowest levels for a finest level of 12. Results for three different geometrical configurations of the domain are shown: Full (black), Ha1 (red) and Hah (blue).

6.8. Scaling properties

Finally, we investigate the scaling properties of the solver according to the number of cores allocated on HPC-FF. Thus, it becomes necessary to measure execution times which in turn depend directly on the stop criterion for the iterative solvers. Hence, results are considered as converged when the initial residual error has diminished by a factor of 10^{-9}. To improve statistics we solve each problem 200 times and calculate the average solution time.

First, we perform a strong scaling by running a fixed problem size on an increasing number of cores. As test problems, we choose the domain Ha1 from the previous section but without the internal conducting structure and the domain Ha1c8 with the internal conducting structure (0.484375, 0.515625) x (1.9375, 2.0625). In addition, we select for each domain three different combinations of the finest level and the total number of levels denoted by {finest level} – {number of levels} in Fig. 11 and Fig. 12. The minimum number of cores which we use to solve each problem is limited by the memory requirements, i.e., the problem size of the finest level characterized by the DoF. The maximum number of cores is limited by the DoF on the lowest level as it is mandatory to have at least one grid point per core.

Each domain is divided into $n=n_x*n_y$ subdomains, i.e. n_x in *x*-direction and n_y in *y*-direction. We increase the number of subdomains n by alternate doubling the number of cores in each direction, i.e., start with n_x*n_y then $(2n_x)*n_y$, $(2n_x)*(2n_y)$, etc. However, if some cores are mapped onto the inner empty space they are forced to idle. This means that we start with 4 cores; then double to 8 cores; then to 16 cores, but this time using only 12 cores. This is feasible because as 4 cores are mapped onto the inner empty space and hence are not needed. From 12 cores on we double again the number of cores by refining the mesh either in *x*- or *y*-direction but still excluding the cores being mapped onto the inner empty space.

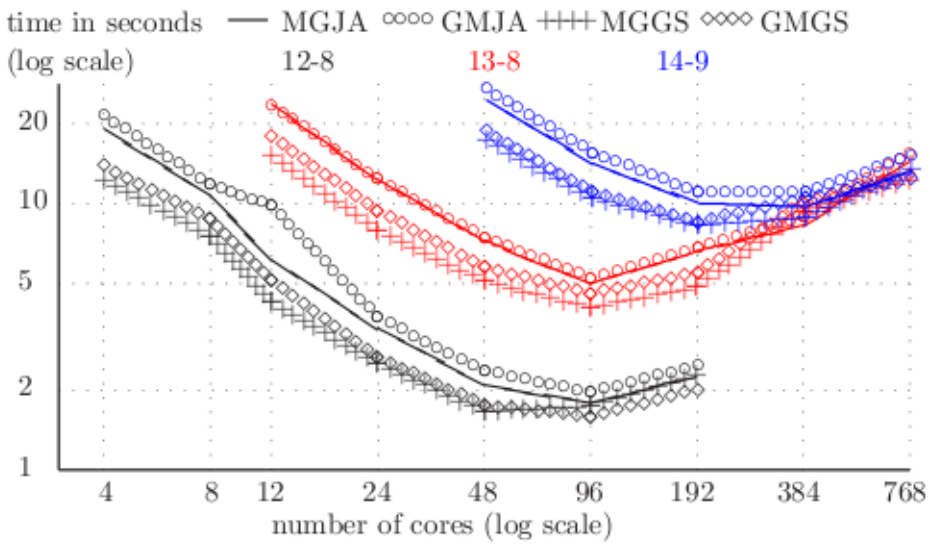


Fig. 11 The solution times in seconds of the multigrid method as a solver and as a preconditioner for the PGMRES method with Jacobi and Gauss-Seidel smoothers as a function of the number of cores on the Ha1 domain.

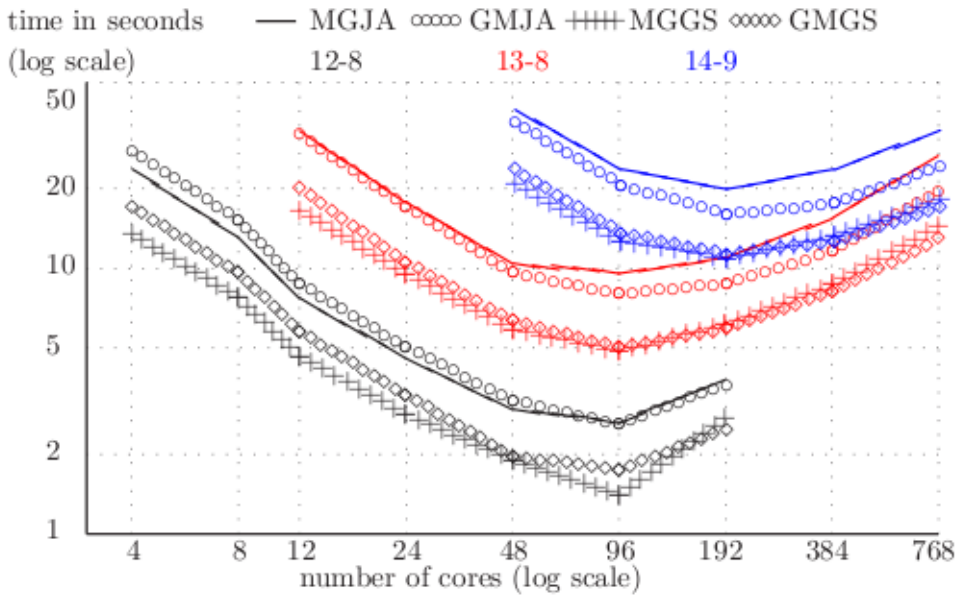
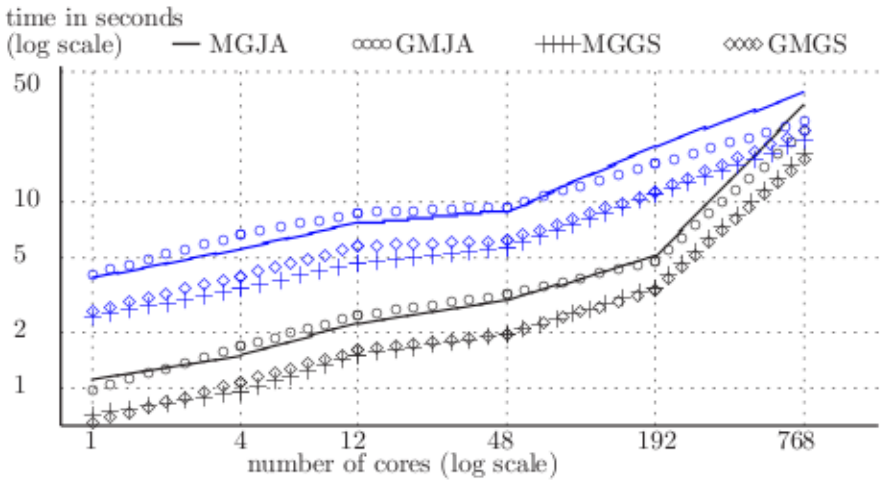


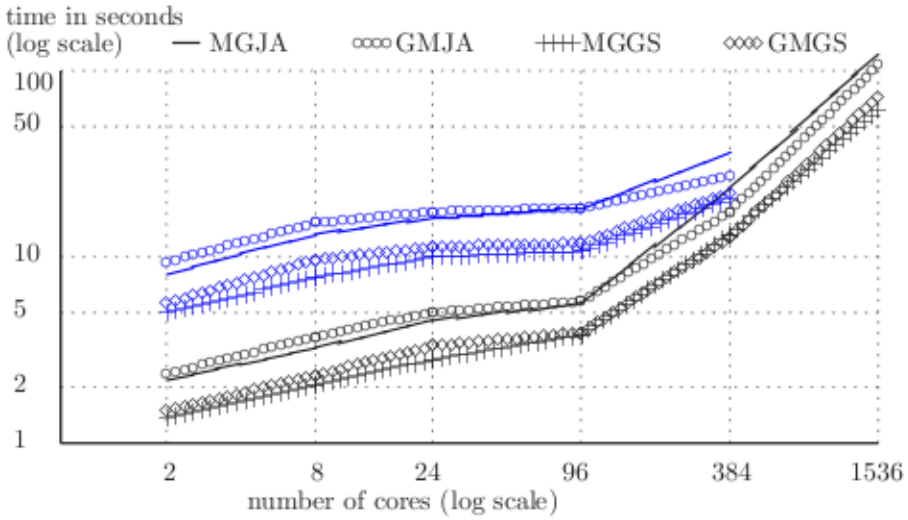
Fig. 12 The solution times in seconds of the multigrid method as a solver and as a preconditioner for the PGMRES method with Jacobi and Gauss-Seidel smoothers as a function of the number of cores on the Ha1c8 domain.

As test problems, for the Ha1 and Ha1c8 domain we choose for the V-cycle the following levels and degrees of freedom: case 12-8 with 12,578,816 DoF and 12,574,559 DoF (black), case 13-8 with 50,323,456 DoF and 50,306,751 DoF (red), and for case 14-9 with 201,310,208 DoF and 201,244,031 DoF (blue), respectively. Due to the internal conducting structure all the cases of the Ha1c8 domain have a smaller numbers of DoF. Each problem is solved either with a multigrid solver with a Jacobi smoother (MGJA) (solid line), a multigrid solver with a Gauss-Seidel smoother (MGGS) (crosses), the PGMRES method with a multigrid preconditioner using the Jacobi smoother (GMJA) (circles), or the PGMRES method with a multigrid preconditioner using the Gauss-Seidel smoother (GMGS) (diamonds). The corresponding execution times of the solving process are depicted in Fig. 11 for the domain Ha1 and in Fig. 12 for the domain Ha1c8.

If we compare both figures, it becomes obvious that the scaling results for both domains are quite similar. The execution time is longer for higher DoF that have to be solved. This is correlated with an improvement of the scaling property for larger problem sizes. Hence, the 14-9 case has the best scaling property and scales nicely up to 96 cores on HPC-FF. If we compare the scaling of the different solvers we can see that they scale similarly. However, the multigrid solver with a Gauss-Seidel smoother (MGGS) has usually the shortest execution time.



(a) 262144 DoF (black) and 1048576 DoF (blue)



(b) 523264 DoF (black) and 2095104 DoF (blue)

Fig. 13 The solution times as a function of the number of cores to solve the Ha1c8 problem with fixed numbers of DoF per core (weak scaling).

In addition, we also perform a semi-weak scaling for the Ha1c8 domain, i.e., the problem size per core is fixed while the number of cores increases. Hence, we choose four different problem sizes per core which consist respectively of 262,144 DoF, 523,264 DoF, 1,048,576 DoF, and 2,095,104 DoF on the finest level. To be precise, the number of DoF per core is not always exactly the same as some cores share parts of the inner conducting structure and thus have to handle less DoF. As the inner conducting structure is relatively small, this should affect only a small number of cores. Hence, the above given DoF reflect the dominating case where the number of DoF for each core is not influenced by the inner conducting structure.

To keep the work per core as constant as possible, we invoke an additional level in the multigrid V-cycle when quadrupling the number of cores. As a result, the mesh of the finest level will increase by a factor of two in each direction so that the DoF of the finest mesh on each core stays constant. The execution time of such calculations is shown in Fig. 13 for the Ha1c8 domain as a function of the number of cores. Unfortunately, the execution time is always increasing with the number of cores although a perfect weak scaling would imply a constant execution time. Part of the problem comes from the fact that we did not construct a proper weak scaling for the multigrid algorithm. We actually solve problems with a different resolution which results in an additional grid level. So we face here the principle problem of how to construct a proper weak scaling case for a multigrid solver.

Nevertheless for very large core numbers we can see that the execution time for the solution increases significantly. So here, the communication overhead becomes prohibitive and impairs the scalability of the multigrid algorithm in the number of cores. However, the larger the number of DoF per core is, the better it is for the weak scaling as the communication costs undergo a relative decrease. So for the largest case of 2094081 DoF per core (about 800 million DoF in total) a relatively good scaling up 384 cores can be achieved.

6.9. Conclusions

We implemented and tested a multigrid solver with a Jacobi smoother, a multigrid solver with a Gauss-Seidel smoother, the PGMRES method with a multigrid preconditioner using the Jacobi smoother, and the PGMRES method with a multigrid preconditioner using the Gauss-Seidel smoother for a certain set of elliptic PDEs on a rectangular domain.

In contrast to standard test problems this problem is already quite complex as the domain contains both an internal conducting structure with a Dirichlet boundary condition and an inner empty space with a Neumann boundary condition. As long as these internal structures are aligned with the grids of the different levels of the multigrid V-cycle there seems to be a negligible negative effect on the convergence rate for the multigrid method both as a solver and as an efficient preconditioner. For the solvers tested here either the multigrid solver with a Gauss-Seidel smoother (MGGS) or the GMRES method with a multigrid preconditioner and a Gauss-Seidel smoother (GMGS) gave the best results.

For the multigrid method one benefits from both the fast execution time which can be orders of magnitudes faster than ordinary iterative solvers and the very good scaling property of the multigrid method which can go up to more than a hundred cores for our test cases. In addition, the iterative multigrid method has a very low memory requirement, when compared to direct solvers. Thus, the multigrid method is suitable for large problems on massively parallel machines like HPC-FF.

However, if the internal structures are not aligned, then the convergence rate can be significantly reduced. In the worst case the multigrid solver will not converge anymore. But even then, the multigrid method can still be used as an efficient preconditioner for the GMRES iterative solver.

For further details please see the technical report *"*The multigrid method for an elliptic problem on a rectangular domain with an internal conducting structure and an inner empty space*"* which has been published as IPP report 5/128.

7. Report on HLST project MGTRI

7.1. Introduction

The physics code project GEMT is intended to generate a new code by combining two existing codes: GEMZ and GKMHD. GEMZ is a gyrofluid code based on a conformal, logically Cartesian grid. GKMHD is an MHD equilibrium solver which is intended to evolve the Grad-Shafranov MHD equilibrium with a 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 method of solution 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, Navier-Stokes equation, and Magnetohydrodynamics (MHD). Although the multigrid method is complex to implement, researchers in many areas think of it as an essential algorithm and apply it to their codes because the number of operations of the multigrid method depends on the degrees of freedom times the number of levels (*log* of the degrees of freedom).

To implement and analyze the multigrid method, we have to consider two main parts of the multigrid algorithm, the intergrid transfer operators and the smoothing operator, separately. The intergrid transfer operators depend on the discretization scheme and are highly related with the discretization of the matrix. The smoothing operator will be implemented according to the matrix-vector multiplication. The multigrid method on triangular meshes was studied by many researchers and reached a mature state. However, the implementation was 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.

7.2. 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 information 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, …, $6 \cdot 4^n$ as e.g. can be seen in Fig. 14. For each core, we classify real and ghost nodes which are positioned on the cores as shown in Fig. 15**.**

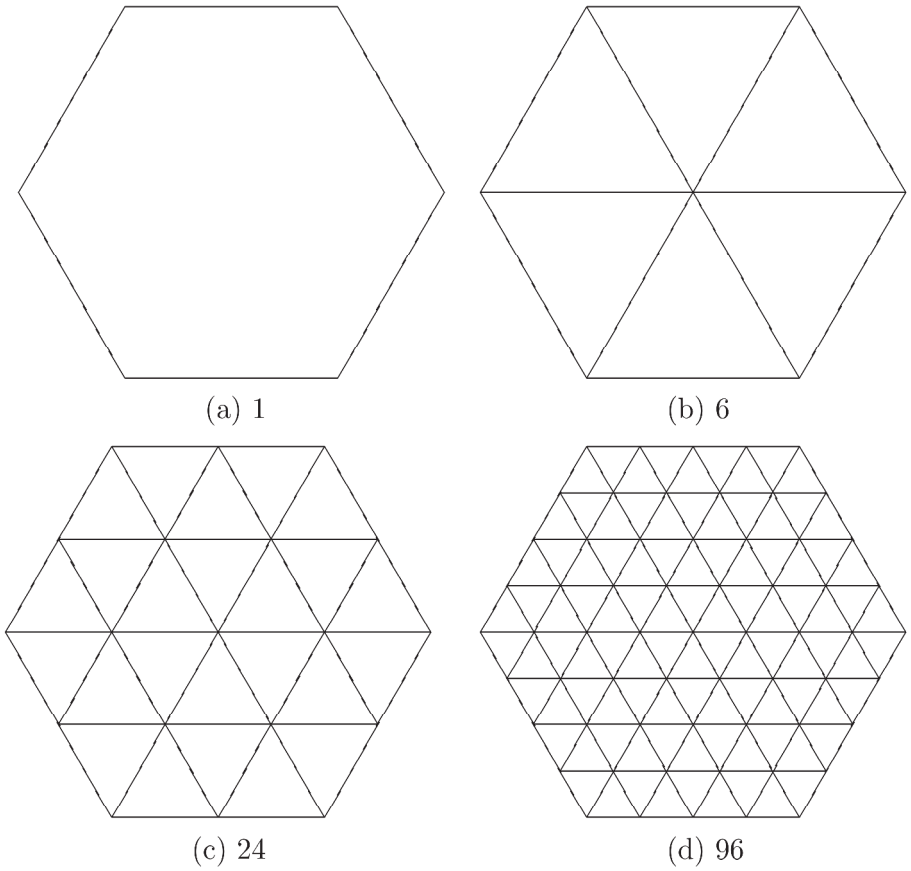
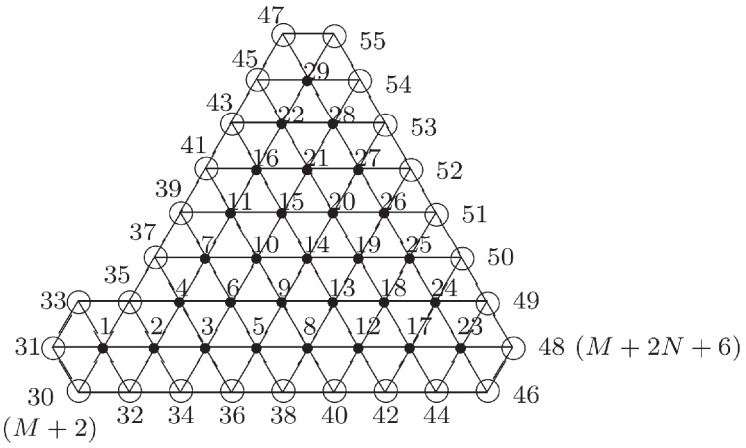


Fig. 14 The subdomains according to the number of cores: (a) one core, (b) six cores, (c) 24 cores and (d) 96 cores.

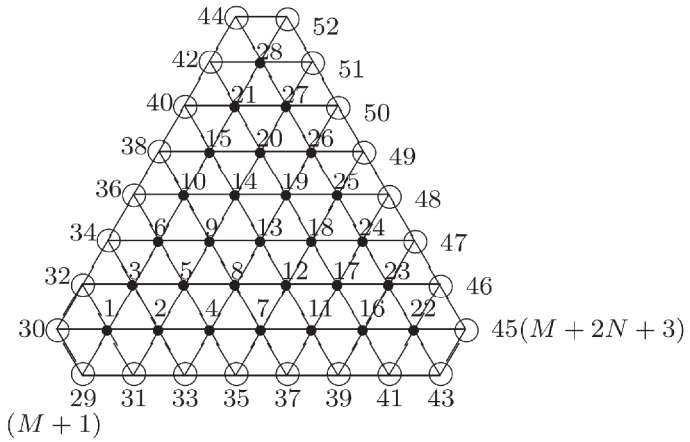
The real nodes are part of the subdomain which is assigned to a specific core. They are handled and updated by the core. Instead, the ghost nodes are originally part of exterior subdomains being located on other cores but nevertheless their values are needed for calculations done on the core. Hence, 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 the position of the subdomain as shown in Fig. 15. 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.

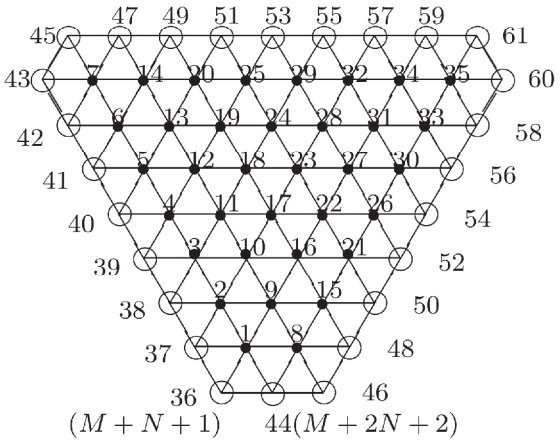
In each communication step every core gets the value of the ghost nodes from the other cores. This process is divided into five sub-steps. It is the dominating overhead of the parallelization and thus a key issue for the performance of the code. The adaptation of other parts of the multigrid algorithm in its serial form is not as critical and should be straightforward.



(a) Sub-domain type I: 0, 6, 9, 12, 15, 18, 21, 24, ...



(b) Sub-domain type II: 1, 2, 3, 4, 5, 11, 14, 17, 20, 23, 28, ...



(c) Sub-domain type III: 7, 10, 13, 16, 19, 22, 25, 27, ...

Fig. 15 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).

7.3. Model problem and discretization scheme

We 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. 16 (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;

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

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

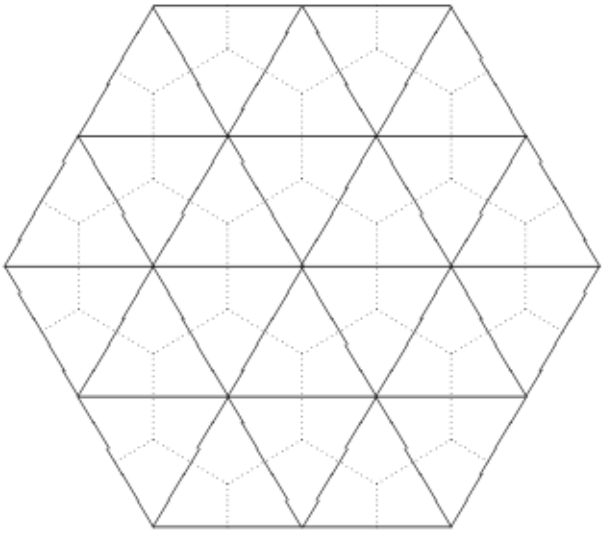


Fig. 16 The primal triangulation (solid lines) and control volumes (dotted lines).

To define the finite volume method, we construct the control volumes of the regular triangular meshes as in Fig. 16 (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, we do not have to distinct between both methods for our consideration.

The multigrid method for finite elements on triangular meshes is well developed and has been analyzed by many researchers. Accordingly, we will use the standard linear interpolation and restriction operators as intergrid transfer operators. In the following, we will restrict ourselves on the execution time of solving the discretized system only.

7.4. Scaling properties

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}. We test four different solvers, the multigrid solver with Jacobi smoother (MGJA), the precondtioned conjugate gradient method with a multigrid preconditioner with Jacobi smoother (CGJA), the multigrid solver with Gauss-Seidel smoother (MGGS), the precondtioned conjugate gradient method with a multigrid preconditioner with Gauss-Seidel smoother (CGGS). As a reference, we also investigate the scaling property of the parallel matrix-vector multiplication which is the kernel in typical parallel iterative solvers as e.g. the Conjugated Gradient (CG) and Gauss Seidel method. In order to ease the comparison with the slopes of the other curves, we multiply the execution time of the parallel matrix-vector multiplication by an arbitrary factor of 50.

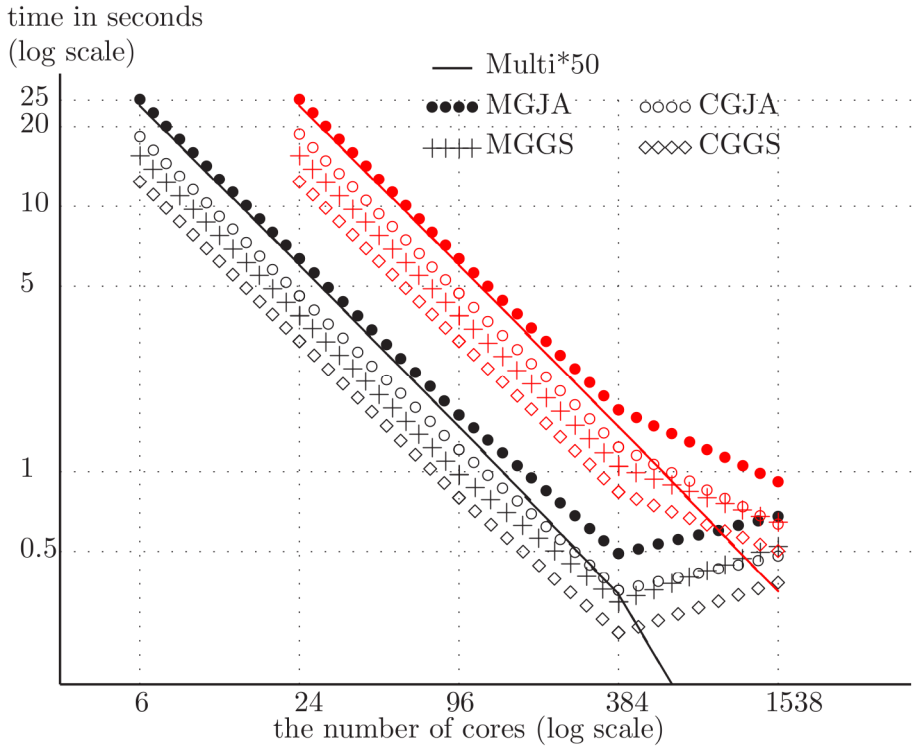


Fig. 17 The solution times in seconds of the multigrid method as a solver and as a preconditioner for the PCG method with Jacobi and Gauss-Seidel smoothers as a function of the number of cores for domains with $50 \cdot 10^6$ DoF (black) and $200 \cdot 10^6$ DoF (red). In addition, the execution time of the parallel matrix-vector multiplication multiplied by a factor of 50 (solid line) is plotted.

For a given right-hand side we report the scaling properties of the different solvers depending on the number of cores involved in the parallelization process. Under consideration are two problems of different size characterized by their degrees of freedom (DoF). We distinguish between the solution times expressed in Fig. 17 and the speedup in Fig. 18. From Fig. 17 we can deduce that the precondtioned conjugate gradient method with a multigrid preconditioner with Gauss-Seidel smoother (CGGS) is the most efficient one. The numerical results show almost perfect strong scaling up to 384 cores as can be seen in Fig. 18. As expected due to a relative reduction of communication overhead, the problem with the most DoF has a better strong scaling property. For the matrix-vector multiplication case, the speedup has a super-linear property which is related to cache effects.

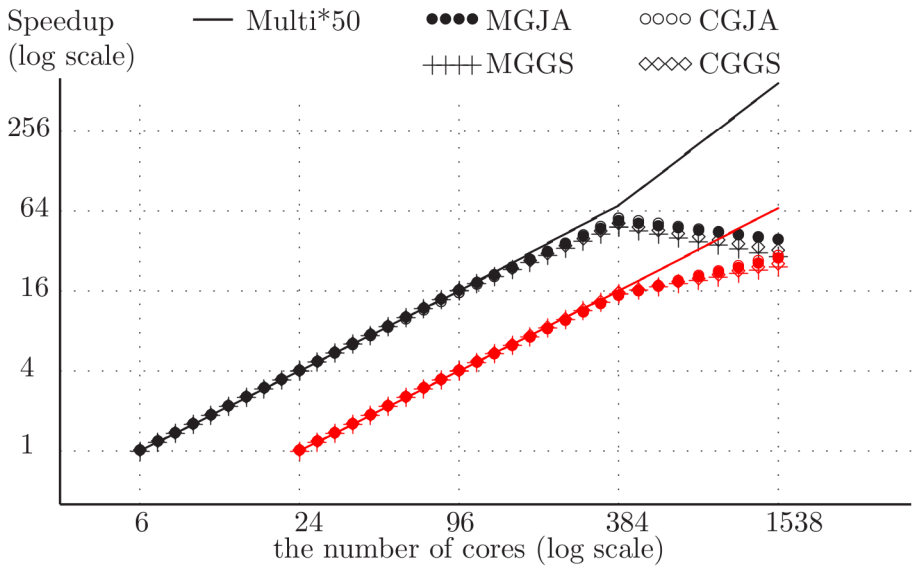


Fig. 18 The speedup of the multigrid method as a solver and as a preconditioner for the PCG method with Jacobi and Gauss-Seidel smoothers as a function of the number of cores for domains with $50 \cdot 10^6$ DoF (black) and $200 \cdot 10^6$ DoF (red). In addition, the speedup of the parallel matrix-vector multiplication multiplied by a factor of 50 (solid line) is plotted.

For the multigrid algorithm it is nearly impossible to fix the number of operations per core while increasing the total problem size. Hence, we consider a semi-weak scaling by fixing the DoF of the finest level on each core. Instead, for the matrix vector multiplication a weak scaling can be determined. We test three different numbers of DoF of the finest level on each core, i.e. $500 \cdot 10^3$ DoF, $2 \cdot 10^6$ DoF, and $8 \cdot 10^6$ DoF. The corresponding solution times of the solvers are depicted in Fig. 19. It can be clearly seen that the matrix-vector multiplications have a perfect weak scaling property and the multigrid method as a solver and as a preconditioner has a very good semi-weak scaling property up to 1536 cores. Especially for larger test cases the semi-weak scaling property improves considerably.

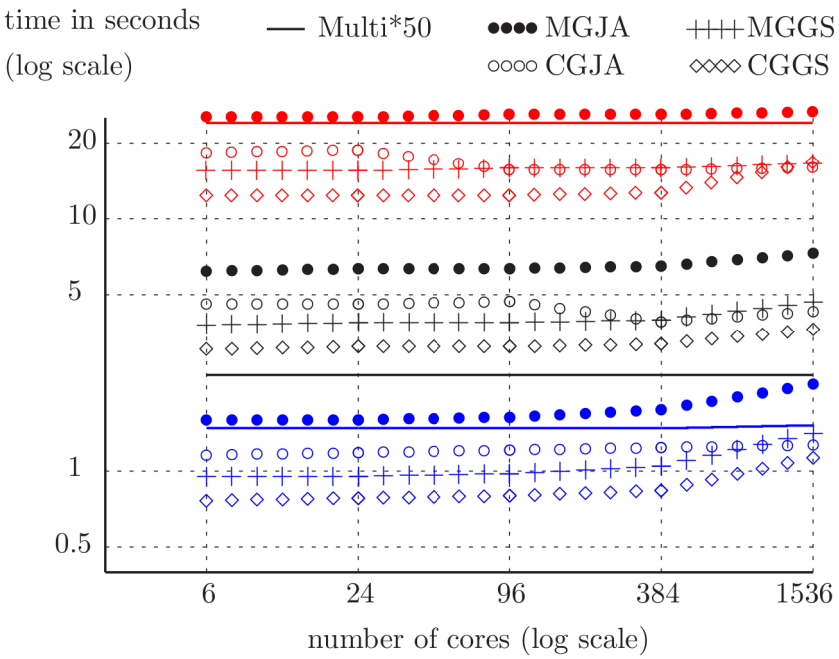


Fig. 19 The solution times in seconds of the multigrid method as a solver and as a preconditioner for the PCG method with Jacobi and Gauss-Seidel smoothers as a function of the number of cores. Three different cases with different fixed DoF per core are depicted: $500 \cdot 10^3$ DoF (blue), $2 \cdot 10^6$ DoF (black), and $8 \cdot 10^6$ DoF (red).

7.5. Conclusions and future work

We have implemented different multigrid solvers for a Poisson problem on a structured triangular grid on a regular hexagonal domain. In this context, it is crucial to derive an appropriate communication pattern between the subdomains to exchange the information necessary for the ghost nodes. To check the correctness of the implementation several tests have been performed. In addition, performance and scaling tests show that the preconditioned conjugate gradient method with a multigrid preconditioner with Gauss-Seidel smoother (CGGS) is the most efficient method under consideration. For the given problem sizes of $50 \cdot 10^6$ DoF and $200 \cdot 10^6$ DoF the numerical results reveal almost perfect strong scaling up to 384 cores. In addition, the multigrid method as a solver and as a preconditioner has a very good semi-weak scaling property up to 1536 cores which improves for larger test cases.

In the remaining project time we plan to perform tests with more realistic data after having consulted the project coordinator, Bruce D. Scott.

8. Report on HLST project NEMOFFT

8.1. Introduction

The main purpose of the project is to remove, or at least alleviate, a parallel scalability bottleneck of the global gyrokinetic ORB5 code, in its current electromagnetic version NEMORB. This code is very HPC resource demanding, especially for large sized-simulations, and relies on efficient parallel algorithms that require filtering to refine the physical quantities. The filtering is done via two-dimensional Fourier transforms and requires large amounts of grid data to be transposed across processors. Obviously, such cross-processor communication impairs the code's parallel scalability, and in practice it currently renders ITER-sized plasma simulations unfeasible.

In principle, there are two possible ways to deal with this issue. Namely, the first is to change the spatial grid to be globally aligned with the equilibrium magnetic field, thus eliminating the need for Fourier filtering; the second way is to improve the way the parallel data transpose and Fast Fourier Transforms (FFTs) are done. While the former would completely eliminate the Fourier transform 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 on the original code. On the other hand, the latter, even though potentially less effective, can be implemented as an external library, with minimal changes to the code required. Further, 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 directly benefit from the second approach. Therefore, this project will focus exclusively on this approach.

8.2. NEMORB's 2D Fourier transform algorithm

NEMORB's three-dimensional (3D) spatial grid comprises the radial, poloidal and toroidal directions, discretized with *Ns*, *Nchi* and *Nphi* grid-nodes, respectively. The domain is further decomposed into *Nvp_cart* sub-domains over the toroidal direction and distributed across the same number of processors, typically as many as toroidal grid-nodes – Fig. 20. The reason why the toroidal direction is used for the domain decomposition instead of the poloidal one is the minimization of inter-process communication. Since in a tokamak the toroidal direction is typically very close to the parallel direction, NEMORB approximates the gyro-radii, which occur in the plane perpendicular to the magnetic field, with their projection on the poloidal plane. Hence, all gyro-radii are local at a given toroidal location. If the domain decomposition was in the poloidal direction instead, this would no longer be the case as there would necessarily be some gyro-radii shared by two adjacent domains, which would greatly increase the amount of communication between them.

NEMORB solves the Poisson equation in the poloidal and toroidal angular Fourier space and relies on filtering to refine the physical quantities. A two-dimensional (2D) fast FFT in those angles is therefore implied. In general, data locality is of key importance for multi-dimensional FFTs and has a major impact to its parallelization and data distribution concept. An *N*-dimensional FFT can be obtained from *N* one-dimensional (1D) FFTs computed sequentially, one for each of the *N* dimensions. Since 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 processor doing the computation. For the case of NEMORB, the toroidal decomposition of its spatial domain sets the poloidal direction as the first one to be (1D) Fourier transformed. First, the input data, which is real, is converted into a complex array of the same size with zero imaginary part and then a complex to complex FFT is performed. The result is stored in a 2D complex matrix (*Nchi* x *Nphi*) which further needs to be Fourier transformed in the remaining (toroidal) direction.

Since the corresponding data is distributed it must first be made local to each processor. This is done with a matrix transpose that swaps the toroidal and poloidal data. It involves an all-to-all communication pattern implemented via a bitwise exchange algorithm written by Trach-Minh Tran. The data local to each processor is sub-divided into as many blocks as there are toroidal sub-domains (processors), and a XOR condition establishes the exchange pattern of the blocks between the processors. The inter-processor data exchange proceeds sequentially for each processor rank inside a DO-loop using MPI_Sendrecv calls. With the data transposed, the toroidal direction becomes local to each processor and the poloidal data distributed. The second 1D FFT, in the toroidal direction, can now be calculated without any communication cost. Fig. 21 schematizes this 2D FFT algorithm.

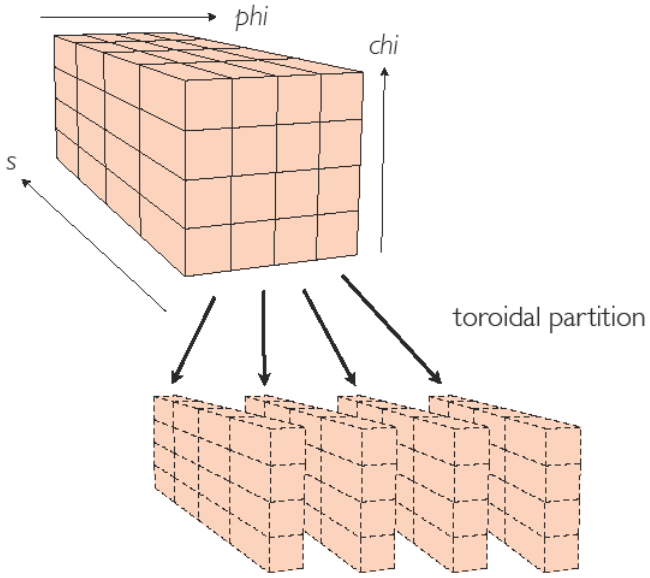


Fig. 20 Illustration of the spatial grid parallelization of NEMORB, with the sub-domains distributed in the toroidal direction.

It is noteworthy that the block sub-division used in the transpose algorithm necessarily implies that there can not be more toroidal sub-domains (*Nvp_cart*) than poloidal grid-nodes (*Nchi*). On the other hand, the domain decomposition implies that there must be at least one toroidal grid-node per sub-domain, so *Nvp_cart* can not be bigger than *Nphi* either. Therefore, the smaller angular grid-count sets the maximum allowed number of sub-domains, which, one should realize, does not disallow cases with *Nchi* < *Nphi*. More precisely, declaring *a* and *b* to be strictly positive integers, for a given number *Nvp_cart* of sub-domains (processors), the grid-count has to be set according to

$$\frac{Nchi}{a} = Nvp\_cart = \frac{Nphi}{b} \quad .$$

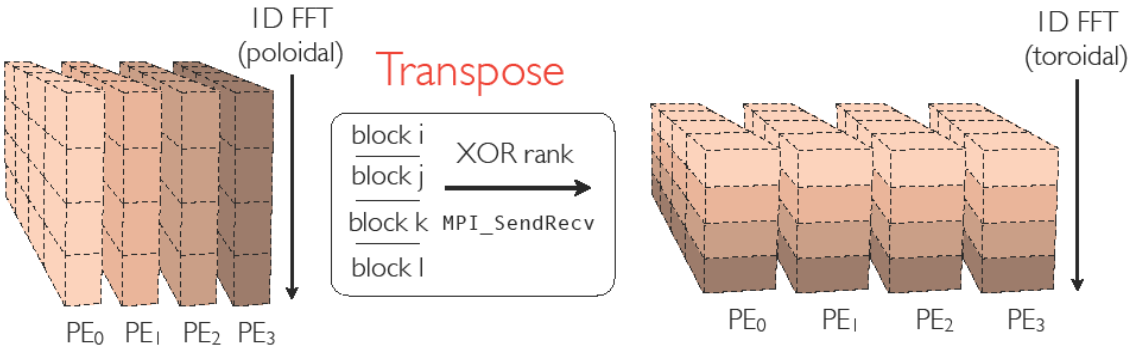


Fig. 21 Illustration of the distributed 2D FFT algorithm consisting of two 1D FFTs interleaved with a distributed transpose is used.

Table 5 shows the performance measurement of this algorithm on a typical ITER-sized spatial grid (512 x 2048 x 1024) distributed across 1024 processors on HPC-FF, when the algorithm was executed 500 times to improve statistics. One sees that the distributed matrix transpose (highlighted in red) represents a significant part (65%) of the whole time, which is mostly due to inter-processor communication (note the lower MFLOPS compared to the 1D FFTs). This percentage can increase for smaller exchange block array sizes, but this shall be discussed in more detail in Sec. 8.4.

		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 5 Performance measurement of NEMORB's 2D FFT algorithm on a typical ITER-sized spatial grid (512 x 2048 x 1024) distributed over 1024 processors on HPC-FF. `FTcol1` and `FTco2` correspond to the poloidal and toroidal FFTs, respectively, `Reord` is the index-swapping-order operation (see Sec.8.5) and `Normal` is the FFT normalization.

8.3. FFT of real data: Hermitian redundancy

The Fourier transform of purely real discrete *Nchi*-sized data exhibits the following symmetry

$$F^*_{Nchi-m} = F_m$$

where the asterisk represents the complex conjugate and the *m*-index is the mode number. Such Hermitian redundancy implies that the FFT can be unequivocally represented with only *Nchi*/2+1 independent complex values. Taking this property into account allows the FFT calculation of real data in a more efficient manner compared the full complex to complex transform, both in terms of memory and CPU usage.

In NEMORB, the 2D data (poloidal and toroidal) to be Fourier transformed is purely real. The Hermitian redundancy can be used in the first 1D FFTs which are performed in the poloidal direction. The FFTs in the second direction (toroidal) act upon the poloidally Fourier transformed data, which is complex. Therefore, the full complex transform has to be carried in this direction that no longer exhibits the Hermitian redundancy. Still, there is a theoretical factor of two for the potential speedup due to the cut by roughly two in the original size of the matrix to be transformed.

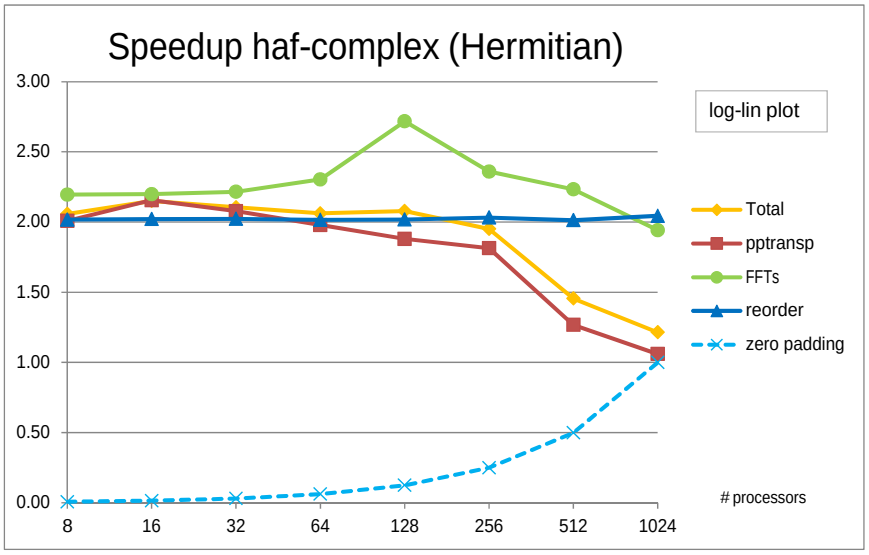


Fig. 22 Speedup factors achieved for each component of the 2D FFT algorithm when using the half-complex representation for the poloidal direction. Note that this is not a pure weak-scaling study since the size of the problem in the poloidal direction is held fixed.

There are two possibilities to use the Hermitian redundancy in the poloidal direction. Either (i) store two poloidal (real) arrays as the real and imaginary parts of a same-sized complex array, compute the complex FFT and use the symmetry relations above to separate the two transforms, or (ii) store one poloidal array in a half-sized complex array in an appropriate manner (even indexes go the real part, and odd indexes go to the imaginary part), compute the complex FFT and use the symmetry relations above to separate the two transforms. In terms of floating point operations, they are both equivalent, but they affect differently the distributed transpose. The former computes two poloidal arrays simultaneously, so it effectively reduces the matrix size to be transposed from the original *Nchi* x *Nphi* to roughly *Nchi* x *Nphi*/2. The latter stores the first 1D FFT in a half-sized complex array (half-complex format), so the matrix being transposed is now roughly of size *Nchi*/2 x *Nphi*. While the former approach is more straightforward to code, it needs at least two poloidal arrays to be local at each processor, preventing the cases with *Nvp_cart* = *Nphi*, which is undesirable. The second method does not suffer from this issue, so it was chosen. Besides, since it is readily available in standard FFT libraries (e.g. FFTW and Intel MKL), there is no need to implement it by hand. One simply calls the “real to complex” transforms of such libraries.

The plot in Fig. 22 shows the speedup factors (ratio between elapsed times) achieved with this method compared to the original “2D complex to complex FFTs” on grid-counts of 512 x 2048 x *Nvp_cart* (number of processors), in the radial, poloidal and toroidal directions, respectively. The several components involved in the 2D FFT algorithm are separated. The two main ones are the FFT calculation (green) and the parallel transpose (red). The blue line shall be discussed in more detail in a subsequent section and can be ignored for the moment. The yellow line represents the overall speedup factor, and it follows closely the red line since, as shown in Table 5, the transpose represents a significant part of the whole computational cost. Up to 256 processors a factor of two is gained, but beyond that a degradation is observed. The reason is that this method requires zero-padding in the poloidal direction to be compatible with the parallel transpose algorithm. The latter requires the poloidal direction of the matrix being transposed to be divided into *Nvp_cart* blocks. Therefore, its number of rows must be a multiple of the number of processors (*Nvp_cart*), and when that is not the case, extra rows of zeros must be added. Knowing that the number of rows is given by *Nchi*/2+1 due to the Hermitian redundancy, for small *Nvp_cart* such condition is easy to fulfill. As *Nvp_cart* gets

bigger, more and more zero-padding is necessary, as the light-blue curve shows. It measures the ratio between the number of extra zero rows added and the original *Nchi*/2+1 size. In the limiting case of *Nchi* = *Nvp_cart* = 1024, one needs to add 1023 extra rows of zeros to the initial *Nchi*/2+1 = 1025 rows, for a total of 2048 rows. This is the same size as the original matrix obtained neglecting the Hermitian redundancy. Hence, for this case there should be no speedup of the transpose algorithm, as the red curve confirms. However, it is important to realize that the zero padding discussed before is simply due to the extra element on top of the remaining *Nchi*/2 elements, which together make the Hermitian reduced complex representation. In fact, without this extra array element, there would be no need for zero-padding up to *Nvp_cart* = 1024 for a poloidal grid-count of *Nchi* = 2048. This is the basis behind the work-around for this issue, which is described in the next section.

8.4. Compact FFT format to avoid zero-padding

As seen before, the Hermitian redundancy states that the Fourier transform of a *Nchi*-sized real dataset has only *Nchi*/2+1 independent complex values. A direct consequence is that, for even *Nchi*, which is the case for NEMORB, the first (F_0) and last ($F_{Nchi/2+1}$) Fourier modes, which correspond to the zero and Nyquist sampling frequencies, are purely real. Therefore, without any loss, the real part of $F_{Nchi/2+1}$ can be stored in the imaginary part of F_0. Calling this the “FFT packed-format”, it allows to reduce the number of rows (poloidal Fourier modes) by one, from *Nchi*/2+1 to Nchi/2. Using again the example from the previous section, up to *Nvp_cart* = 1024 with *Nchi* = 2048, there is no need to have any zero-padding since *Nchi*/2 is always a multiple of *Nvp_cart*. The scaling study of Fig. 22 is repeated in Fig. 23 using the packed-format technique.

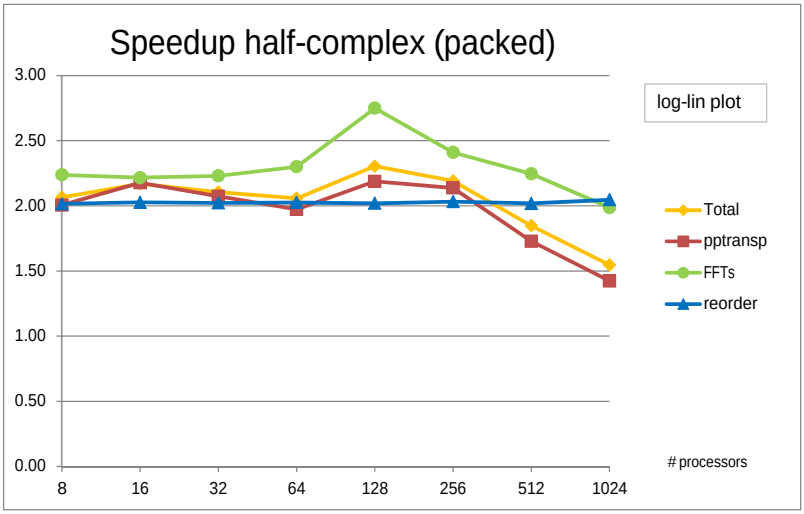


Fig. 23 Speedup factors achieved for each component of the 2D FFT algorithm using the half-complex packed format representation for the poloidal direction.

All the curves, except the transpose and total ones, stay unchanged as expected since there were no changes to those parts of the algorithm. The transpose speedup is always the same or higher than before, even for the cases that originally required very small zero-padding percentages. This means that extra cost of copying to convert to packed-format FFT is negligible. For higher number of processors (*Nvp_cart*), which before required 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 highest two *Nvp_cart* values, there is a degradation of the speedup, which can not be accounted by the matrix size, as there is no zero-padding whatsoever.

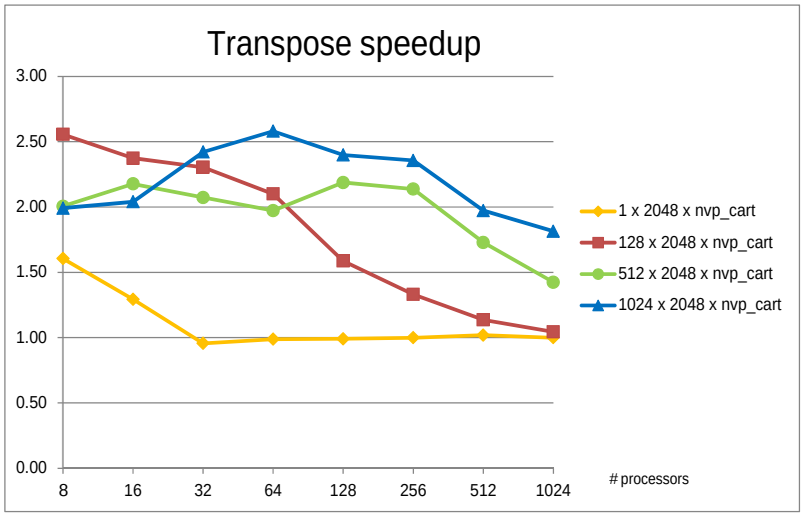


Fig. 24 Speedup scaling with the radial (ns) grid size, when using the FFT half-complex packed-format for the poloidal direction.

Such degradation is related to the latency times inherent to any inter-processor communication. Since communication is performed sequentially for each block with an MPI_Sendrcv call, the larger the *Nvp_cart*, the more latency time gets accumulated. As the size of the blocks being transposed decreases with *Nvp_cart*, the latency times becomes comparable to the time spent on the actual transfer. To test this idea, Fig. 24 shows the speedup factors for the parallel transpose scaling test with different matrix block sizes. The angular grid-count is kept unchanged (same as in Fig. 22 and Fig. 23) but the radial grid count varied between 1, 128, 512 and 1024. It is clear that, for the smallest matrix block sizes being transposed (yellow curve), the speedup degradation occurs very soon, indicating that the communication latency times dominates even the Hermitian redundancy reduction of the number of matrix rows. Accordingly, increasing the matrix block sizes delays the speedup degradation. The biggest matrix block size case (blue curve) yields >1.8 transpose speedup with *Nvp_cart* = 1024, for the same poloidal and toroidal grid-counts as in Fig. 23, which here corresponds to the green curve. To explain the maxima observed in some of these curves, it should first be noted that they (like the ones in Fig. 22 and Fig. 23) are not representing absolute quantities. They represent the ratio between the elapsed times of two methods, which in turn always rise with increasing number of processors. How fast they rise depends on two factors, namely, the increase in matrix size (*Nphi* is set by *Nvp_cart*) and the accumulation of communication latency time. While the former accounts for the speedup differences for lower *Nvp_cart* the latter plays a more important role for higher *Nvp_cart*.

A comment on the special case *Nvp_cart* = *Nchi* is in order. Naturally it requires zero-padding when the Hermitian redundancy is used, even with the packed-format representation. The reason is simply that, *Nchi*/2 is smaller than *Nvp_cart* and to use the parallel transpose algorithm, *Nvp_cart* - *Nchi/2* = *Nchi/2* rows of zeros need to be added. Obviously this leads to little speedup with respect to the original complex-to-complex FFT formulation. The speedup would come solely from the more efficient calculation of the FFTs, but the transpose size, which is what raises most execution time, would remain the same. Another possibility to further extend this speedup boundary and still have a gain on the limiting case *Nvp_cart* = *Nchi* would be to modify the transpose DO-loop to communicate, in sequence, the *Nchi*/2 real elements followed the *Nchi*/2 imaginary elements, for a total of *Nvp_cart* messages to be passed, with no need for zero-padding. Of course, from what was discussed in Fig. 24, this method would be more affected by communication latency (due to the half-sized matrix block being transposed), so this has not been implemented in practice. Finally, even if the zero-padding checking condition that has been

implemented in the code relaxes the constraint imposed by the original transpose formulation and allows for cases with *Nvp_cart* > *Nchi*, they should not be used in practice. The code should be able to run, but very inefficiently, with lots of wasted resources on transposing zeros.

8.5. Index order swapping: local transpose

The blue curves in both Fig. 22 and Fig. 23 refer to the swapping index operation that is required in NEMORB before the 2D FFTs are calculated. The index order in the main code is (*s*, *chi*, *psi*) but the Fourier transform modules require (*chi*, *s*, *phi*). This amounts in practice to performing a local transpose between indexes 1 and 3 on the code's 3D data array. Originally this was done via two nested DO-loops, and the speedup measured in the referred plots was achieved using Intel MKL local transpose. To make such decision, several tests were made on HPC-FF comparing these two index-swapping methods with the intrinsic Fortran transpose. An example is given in Fig. 25 for the speedup of using the MKL transpose routine MKL_DOMATCOPY over the nested DO-loops on real data matrices with sizes between 500 x 500 and 2000 x 2000.

For these matrix sizes, which are in the range of the ones used in NEMORB for an ITER-sized simulation, the MKL transpose is always faster. This explains the gain obtained in the re-ordering curves in Fig. 22 and Fig. 23. Although 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 will perform. This is intuitive since for smaller matrices the overhead of using the MKL algorithms dominates.

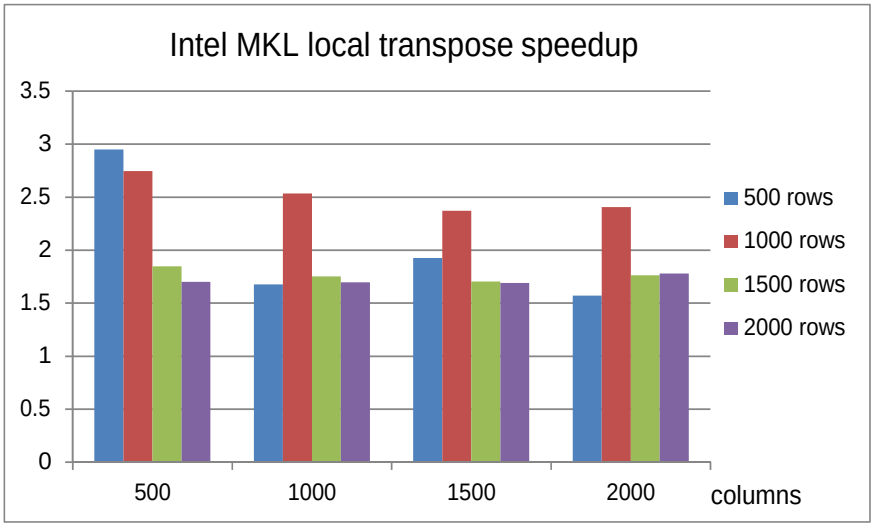


Fig. 25 Intel MKL local transpose speedup relative to a DO-loop transpose.

8.6. Discussion and planned roadmap

The 2D FFT algorithm of NEMORB was modified to use the Hermitian redundancy of its real input data's Fourier transforms. Storing the Fourier modes in a half-complex-packed-format allowed for speedup times of the order of 2. The communication latency starts to become an issue for higher numbers of processors when the matrix transposing blocks get small enough, which degrades the speedup. For these cases, an alternative MPI_alltoall-based transpose using memory non-contiguous MPI derived types might give better results and such a test is planned for the future.

The recent results reported in [1] which show it is advantageous to do the domain decomposition with particle decomposition (clones) as the first communicator. Since

the charge assignment operation involves global sums over the clones, if all the clones of a sub-domain are distributed within each compute node (in HPC-FF up to 8 clones are possible, in IFERC twice as much is possible) this reduces significantly the amount of inter-node communication. Making the sub-domain communicator the second one being distributed brings no disadvantages because the number of sub-domains of each clone is usually too large to be located on an individual node anyway. Hence, the 2D FFTs that work on the sub-domain communicator will always require an all-to-all communication pattern due to the distributed transpose that involves inter-node communication.

As to the planned roadmap, now that the transpose cost was optimized by using the Hermitian symmetry properties of the data, the next steps involve testing other distributed matrix transpose algorithms. The MPI_alltoall with non-contiguous MPI types' transpose was already mentioned, but the FFTW3.3 and Intel MKL PBLAS transposes will also be tested. Ultimately, the transpose speedup degradation imposed by the communication latency indicates that the final solution will most probably have to make use of data-locality within each node. The plan is to use a hybrid MPI/Shared Memory Segments (SMS) parallelization scheme. The idea is to have a bigger block of the whole matrix of data accessible to all processors within the node and then do the distributed transpose with MPI communication across nodes. This choice implies using the node-locality for the sub-domain communicator, rather than for the inter-clone communicator as mentioned in the previous paragraph. This approach should nevertheless be tested.

8.7. References

[1] P. Angelino *et al*., private communication, CRPP 2011

9. Report on HLST project ZOFLIN

The targets of the ZOFLIN code are the nonlinear interaction of zonal flows (ZF) among each other, which sets up equilibrium flow states characterised by particular flow patterns and transport level, the impact of varying plasma geometry on the poloidal flows, and the effects of high gradients and fluctuation levels of the tokamak edge on poloidal flow oscillations, so called geodesic acoustic modes (GAM).

We started working on ZOFLIN from 1st, Feb 2011. The project is concerned with code optimization with emphasis on reordering the memory access during computation of difference operators, collection of communication requests and use of asynchronous communication during computation that would allow overlapping communication and computation.

We were given a representative test case to be run on 64 cores with most of the options that a production case would require, switched on. All our analysis and findings are with respect to this test case.

Before starting any optimization work on the ZOFLIN project, the code was analyzed with Forcheck; the program dependencies cross checked with Automake, profiled using the instrumentation library *perflib* and the conformability of the code checked to the MPI standard using Marmot. This was to ensure that we had a clean and easily manageable code and also to understand the hotspots in the code that we needed to concentrate on for our work.

9.1. Code analysis with Forcheck

Forcheck is a static FORTRAN program analyzing tool which allows developers to cross check their programs for any undetected bugs and also to ensure that the program conforms to FORTRAN standards. During analysis, we were able to identify parts of the ZOFLIN code that did not conform to the FORTRAN 2003 standard. These code segments were changed accordingly. A few illustrative examples, of the changes done, follow:

1. Many of the subroutines did not have an interface for them. This prevents any type checking during the subroutine calls and can lead to programming errors. For example, the code has calls to the C I/O routines, *fopen*, *fread*, *fwrite* and *fclose* that have been declared to be external routines. We have created an interface for these routines using the *iso_c_binding* module which, as part of the 2003 standard, allows us to interface FORTRAN and C routines.

   Adding interfaces to subroutines revealed other errors. For example, the intent of the variables passed from one subroutine to another was not consistent. A variable with intent ‘IN’, which means the variable cannot be modified, in one subroutine was being passed with an intent of ‘INOUT’, which means the variable can be modified, to another subroutine. This has now been changed to be consistent across the relevant subroutines.

   One advantage of introducing the interface using the *iso_c_binding* module is that it allows the code to be compiled and analyzed using Marmot, which is a tool to check whether the application conforms to MPI standards. We analyzed the original code version and the version with Forcheck related changes using Marmot and found that there were no errors in the calls to the MPI library or other possible run-time issues such as deadlocks etc.

2. Several variables in the programs were declared with the Fortran 77 convention; these have been changed to conform to the 2003 standard.

We did some tests with the changed code and compared the results with the original one to ensure correctness and consistency. The changes were approved by the Project coordinator. We also made other suggestions in order to make the code Fortran 2003 compliant. This included the use of Fortran and C pointers for FFT routines to circumvent the problem of using arrays simultaneously as Real and Complex type for use in the FFT routines.

9.2. Dependency check with Automake

Automake is a tool that can be used to check dependencies between programs written in FORTRAN or C to ensure that the program executable is built correctly.

We analyzed the programs in the ZOFLIN project and checked the dependencies reported by Automake. This was done for both the original version and the code version with our Forcheck related changes. We found that, for both the versions of the code, the dependencies listed by Automake and those being used during compilation were the same.

9.3. Profiling with perflib

perflib is an instrumentation library that can be used to profile each part of a program to find performance related information. To *perflib* requires linking the program to be instrumented with two libraries *libperf.a* and *libpfm.a*. The program also needs to be changed to use the *perflib* routines that carry out the instrumentation.

We analyzed both the original version and the version with our changes to ensure that there were no adverse effects of our changes on the code. The profiling was done on a single processor. We found only minor differences in the run times between the two versions. These were system related issues and not due to our changes to the code itself.

9.4. Changes tested with the code

We have done careful performance tests with the *perflib* to identify hotspots; i.e. code that can be optimized. The proposal made by the Project coordinator suggested a 50% improvement in the run time. This is not very trivial to achieve as the single processor performance on HPC-FF, for the original code, is around 2.3 GFLOPS which is about 20% of the peak performance (11.72 GFLOPS) on a single core. This in itself is quite good for a RISC processor and makes it difficult to improve the single processor performance any further, especially since the memory requirements in respect of bandwidth are also high. We discuss this further when mentioning the PSI_TPP environment variable.

We carried out some tests on HPC-FF to determine if the code was memory or cache bound. The Intel Nehalem processor on HPC-FF supports dynamic resource scaling which allows active cores to make use of any resources left free by the idling cores. This includes, for example, the L3 cache (8 MB – shared among 8 cores) and higher memory bandwidth per core (32 GB/s).

We ran the code on 64 cores on HPC-FF and then used the PSI_TPP environment variable to set the number of active cores within a node [1]. We found that, using four active cores per node, decreases the run time per core (53 seconds) compared to using eight active cores per node (62.9 seconds).

The code uses a large number of three-dimensional complex type arrays (up to thirteen of them within one of the loops in the hotspots). The prefetching algorithm might not be efficient enough to have the data from all these arrays readily available when required. Also, getting such a large amount of data from the L3 cache to the L2

cache (private for each core) can also contribute to the total run time. Our test with PSI_TPP proves that the reduced run time is due to the higher bandwidth available per active core and indicates that the code is either cache-bound or memory-bound. Finding scope for optimization for such codes is not trivial.

Another feature of the code is that it uses a Fortran 77 style of programming, where the arrays sizes are fixed depending on the test case and are known during compile time. This makes it easier for the compiler to optimize access to these arrays. The code is, as a consequence, quite efficient to begin with.

For optimization, it is necessary to find out the hotspots in the code that consume the maximum percentage of the run time. Table 6 gives a sample output of the *perflib* for *pm.1*, the main program, from the original code version, run on one processor. The region '*NLET'* encompassing the whole execution code, with a run time of 353.5 seconds, and only those routines that consume more than 5% of the run time, have been shown. The column marked *Exclusive* shows the amount of time spent in the corresponding subroutine excluding any sub-calls to the *perflib*. For example, the subroutines marked as '*S*' and '*f07bsf*' (both highlighted in red) consume about 31.8% (112.43 seconds) and 14.4% (50.98 seconds), respectively, of the total run time (353.5 seconds). We concentrated our optimization efforts on these code regions.

		Inclusive			Exclusive		
Subroutine	#calls	Time(s)	%	MFLOPS	Time(s)	%	MFLOPS
NLET	1	353.5	100	2270.16	0	0	10.55
EC	4002	113.0	32.0	2250.55	38.61	10.9	3864.27
EF	4001	61.19	17.3	2120.35	61.06	17.3	2124.47
S	4000	163.52	46.3	2324.66	112.43	31.8	2391.99
f07bsf	512000	50.98	14.4	2181.07	50.98	14.4	2181.07
DIAG	4000	21.93	6.2	1068.23	21.93	6.2	1068.23
Size of data segment used by the program: 112.33 MB							
Total program size: 556.72 MB Resident set size: 116.81 MB							

Table 6. Sample output from *perflib* for main program *pm.1*, run on one processor.

The *'S'* subroutine, from Table 1, makes use of a large number of C macros which are expanded by the pre-processor. Depending on the test case, we usually encounter nested macros. The use of these macros stems from the fact that the code reuses a large number of equations at different parts, for example, during matrix set-up, matrix solve etc., and uses different parameters to solve these equations. In order to make it easy for the code developer to change these equations, they are put in the form of macros and re-defined wherever required. This also reduces the chances of introducing errors while changing the equations. The difference operators, mentioned in the proposal, are also calculated with these macros.

One consequence of the macro usage is that it makes it much more difficult to analyze the program as the non-preprocessed code and the preprocessed code is very different and much harder for a human to read and understand. Also, we need to make any changes on the macro level; else they will not be accepted by the project coordinator. This makes it necessary to change the macros carefully.

Nevertheless we have tried various options to try and change the macros to use the cache more efficiently. We give some examples below:

1. Common sub-expressions were removed. The compiler, in general, should be able to eliminate them. However, the code seems to be sensitive to the order of floating-point operations and gave slightly varying numerical results.

2. *If* constructs in a loop are, in general, expensive as the compiler has to discard the results of any incorrect branch prediction. These were removed wherever possible.

3. The order of array indexing within a nested loop was changed to be inline with the loop order. Here also, we did not find much improvement in the run time.

4. One of the macros was changed to exclude repetitive calculations. This gave only a minor improvement in the total run time.

5. Copying the required data from the various 3D arrays to a temporary 4D array. This 4D array is then indexed into, depending on the data we need, and used while computing.

We also replaced the NAG implementation of a LAPACK matrix solver routine, *f07bsf*, with an MKL implementation, *zgbtrs*. However, we were not able to get any significant gains in run time with these changes. So, we decided to look at the MPI communication pattern of the code to determine if we could get any run time gains from changing the pattern of communication. However, we had some issues on HPC-FF that had to be fixed before we could proceed.

9.5. Issues with MPI_GATHERV

We have ParTec MPI installed on HPC-FF which is upgraded from time to time. After one such upgrade, we found that we could no longer run ZOFLIN on HPC-FF. The program terminated with an assertion error which was

Assertion failed in file helper_fns.c at line 335: 0
memcpy argument memory ranges overlap, dst_=0x2d58ee0 src_=0x2d58ee0 len_=4

We found that the new MPI version allowed additional checks to ensure that the buffers being used for sending and receiving did not overlap with each other. ZOFLIN had a call to *MPI_GATHERV* where the send and receive buffers were pointing to the same location in an array on MPI process with rank 0. This is a violation of the MPI standard which does not allow aliasing of buffers. We were able to fix the problem by changing the arguments in the call to *MPI_GATHERV*. We could then run ZOFLIN on HPC-FF without any issues.

9.6. MPI communication pattern and load balance

We proceeded with the analysis of the communication pattern among the cores to get an idea of how much time the application spends in communication, and how much in computation. To do so, we used the *Intel Trace Analyzer and Collector (ITAC)* [2] tool available on HPC-FF. Also, to get an idea of the load balance across cores, we used the *perflib* available on HPC-FF. The total run time and the MFLOP rate can be used to find out if the application is load balanced.

9.6.1. Intel Trace Analyzer and Collector

The *ITAC* tool has two parts, the *Collector* and the *Analyzer*. Using the *Collector* is fairly easy and only requires linking the application with the tool libraries. The *Collector* collects information on the time spent within the user application and that spent in MPI communication. This information is then written out to a *.stf* file which can then be viewed with the *Analyzer* in the form of a flat profile, graph etc.

Fig. 26 shows the flat profile of a run done on 64 cores on HPC-FF. The *TSelf* column gives the time spent in the given function, excluding the time spent in functions called from it. The *TTotal* column gives the time spent in a specific function, including functions called from it. The column *#Calls* gives the number of calls to a given function, *TSelf/Calls* is self explanatory. All the columns are the cumulative total for the total number of cores the job has been run on. From Fig. 26, we can find out the total time taken (on 64 cores) for all MPI calls. This is about 1170 seconds (sum of *TSelf* values for MPI calls) which is about 30% of the cumulative run time, 4007 seconds (*TTotal* column). It is clear that calls to the *MPI_Waitall* routine consumes the maximum amount of time compared to other MPI calls. The time spent in this call (469 seconds) is around 40% of the time spent in only MPI calls (1170 seconds). Hence we concentrated on reducing the time spent in this call.

One of the uses of the *MPI_Waitall* call in the code is to exchange the boundary data among the neighboring cores. This is done by waiting for the *MPI_Isend* and *MPI_IRecv* requests to complete and enables asynchronous communication. This exchange of data is done for different field variables in the code. We were able to combine the boundary data of some fields into an MPI derived data type and exchange it with the neighboring cores with a single *MPI_Waitall* call. However, this did not give any improvement to the overall run time. There was only a significant reduction in the number of calls to *MPI_Waitall, MPI_ISend* and *MPI_IRecv*. Fig. 27 shows the output of the Trace tool.

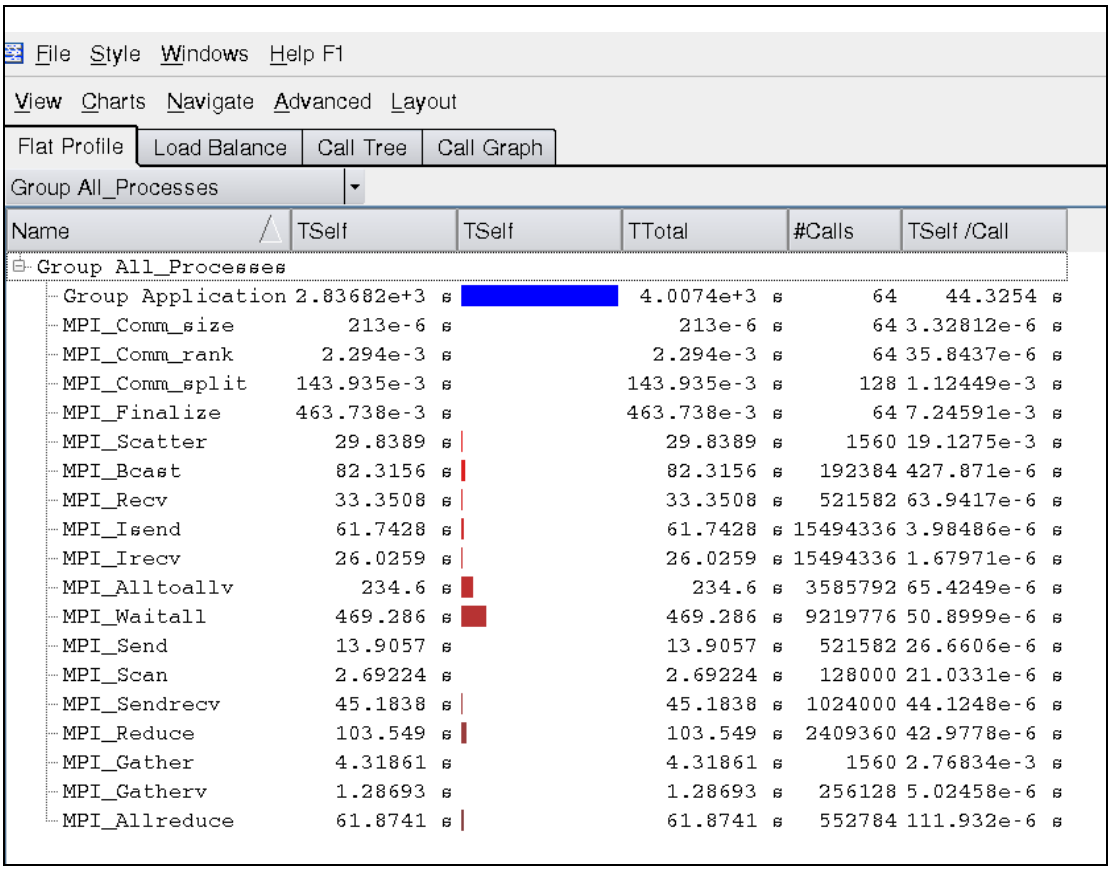
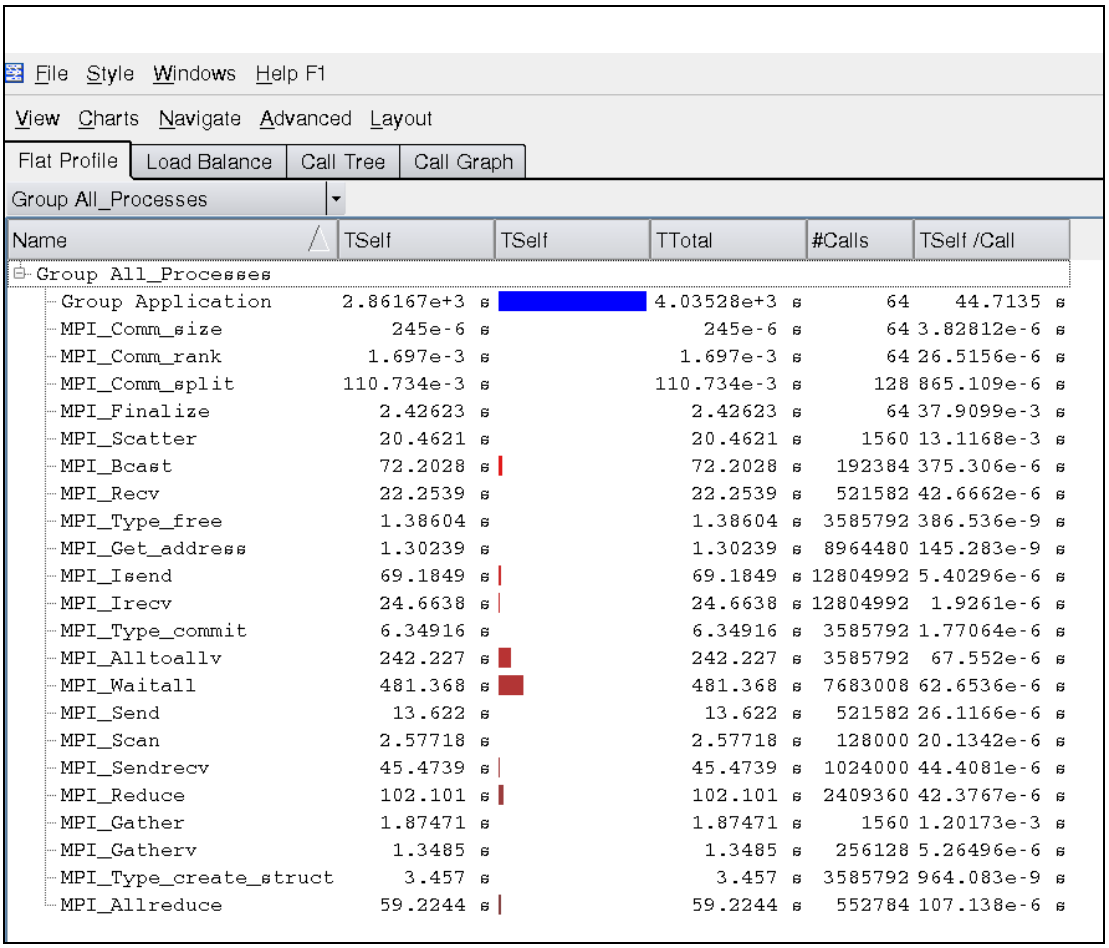


Fig. 26 Profile data displayed by the *Intel Trace Analyzer* for a 64 core run on HPC-FF.



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	2.86167e+3 s		4.03528e+3 s	64	44.7135 s
MPI_Comm_size	245e-6 s		245e-6 s	64	3.82812e-6 s
MPI_Comm_rank	1.697e-3 s		1.697e-3 s	64	26.5156e-6 s
MPI_Comm_split	110.734e-3 s		110.734e-3 s	128	865.109e-6 s
MPI_Finalize	2.42623 s		2.42623 s	64	37.9099e-3 s
MPI_Scatter	20.4621 s		20.4621 s	1560	13.1168e-3 s
MPI_Bcast	72.2028 s		72.2028 s	192384	375.306e-6 s
MPI_Recv	22.2539 s		22.2539 s	521582	42.6662e-6 s
MPI_Type_free	1.38604 s		1.38604 s	3585792	386.536e-9 s
MPI_Get_address	1.30239 s		1.30239 s	8964480	145.283e-9 s
MPI_Isend	69.1849 s		69.1849 s	12804992	5.40296e-6 s
MPI_Irecv	24.6638 s		24.6638 s	12804992	1.9261e-6 s
MPI_Type_commit	6.34916 s		6.34916 s	3585792	1.77064e-6 s
MPI_Alltoallv	242.227 s		242.227 s	3585792	67.552e-6 s
MPI_Waitall	481.368 s		481.368 s	7683008	62.6536e-6 s
MPI_Send	13.622 s		13.622 s	521582	26.1166e-6 s
MPI_Scan	2.57718 s		2.57718 s	128000	20.1342e-6 s
MPI_Sendrecv	45.4739 s		45.4739 s	1024000	44.4081e-6 s
MPI_Reduce	102.101 s		102.101 s	2409360	42.3767e-6 s
MPI_Gather	1.87471 s		1.87471 s	1560	1.20173e-3 s
MPI_Gatherv	1.3485 s		1.3485 s	256128	5.26496e-6 s
MPI_Type_create_struct	3.457 s		3.457 s	3585792	964.083e-9 s
MPI_Allreduce	59.2244 s		59.2244 s	552784	107.138e-6 s

Fig. 27 Run times on 64 cores on HPC-FF, with boundary exchange of data done for more than one field done at the same time. The number of calls to *MPI_Waitall*, *MPI_Isend* and *MPI_irecv* is now reduced.

9.6.2. Load Balance and MFLOP rate

We used the *perflib* library and did runs on 64 cores on HPC-FF to determine if the application was load balanced. The total run time together with the MFLOP rate on each core helped us determine whether the amount of work, in terms of computation, was the same across all cores. The runtime (seconds) and MFLOPS distribution across cores are given in Fig. 28 and Fig. 29. The FLOP rate was found to be around 1.6 GFLOPS per core when running on 64 cores. This was around 2.3 GFLOPS when running on a single core. The code is very well load balanced and there is at most only a 1% difference between the highest and lowest run times.

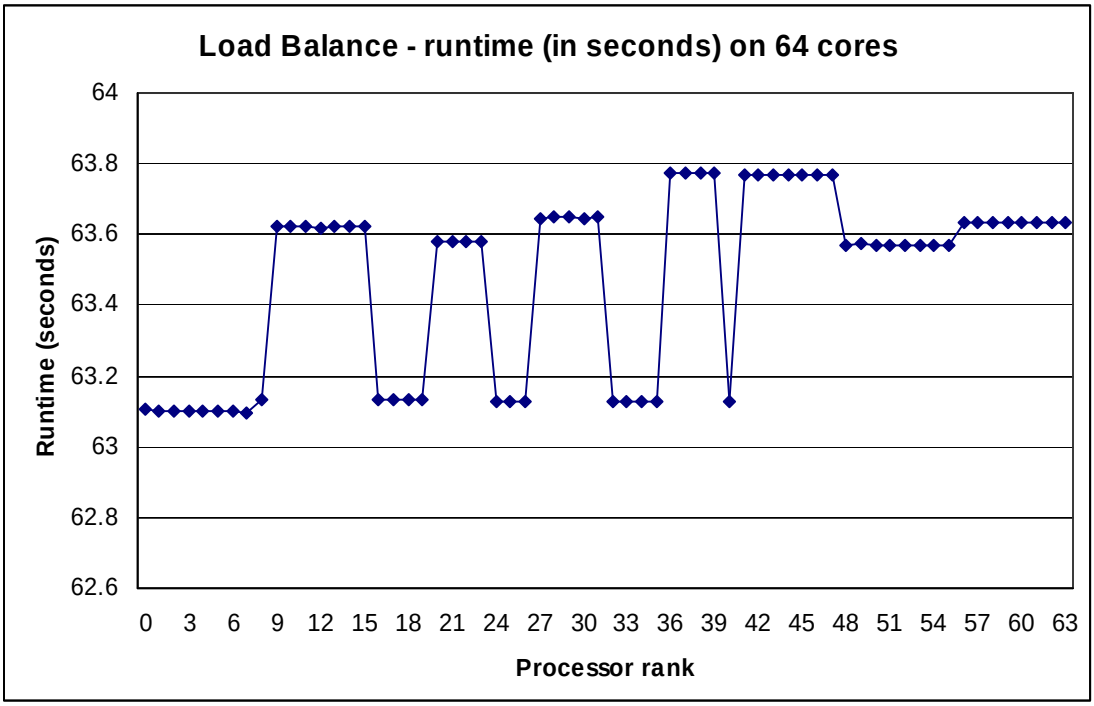


Fig. 28 Load Balance – run time in seconds across 64 cores on HPC_FF.

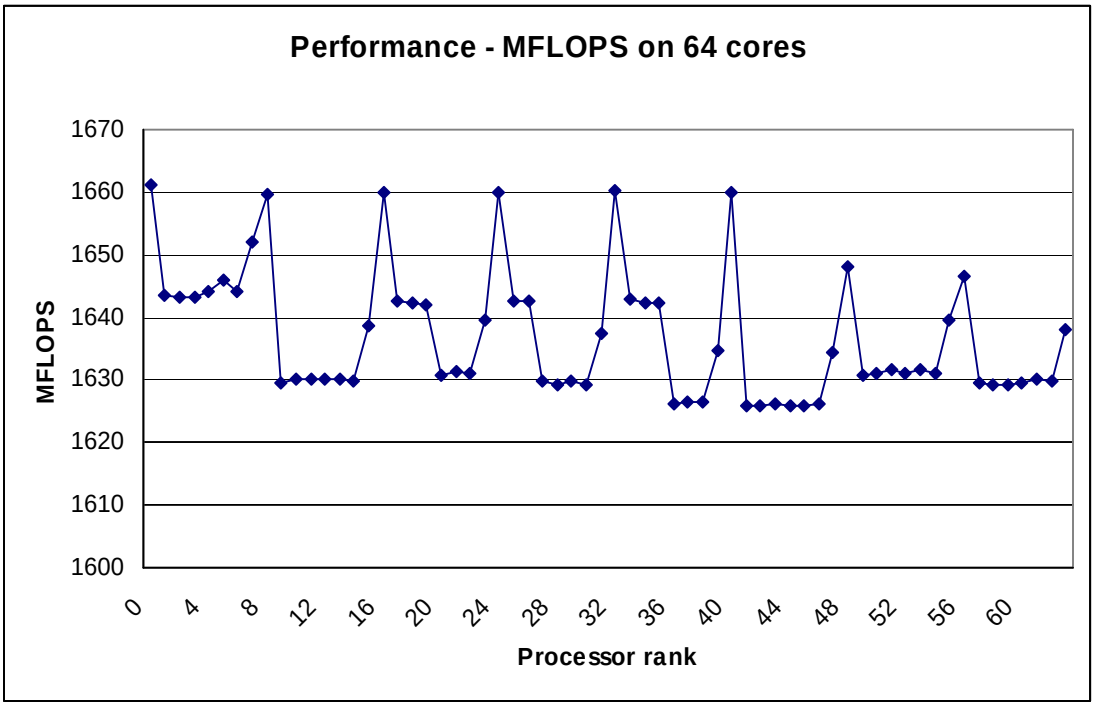


Fig. 29 MFLOPS distribution across 64 cores on HPC-FF.

9.7. Conclusions

We have adapted ZOFLIN to be Fortran 2003 compliant, which allows the code developers to follow a more standard method of programming. This also helps to have a more portable code. Use of tools like Forcheck, Automake and Marmot ensured that there are no programming errors or errors in the Makefile.

We used the *perflib* to determine the FLOP rate of different parts of the code. We found the efficiency of the code, with respect to the GFLOP rate to be quite high on single (2.3 GFLOPS) and multiple processors (1.6 GFLOPS per core on 64 cores). This suggested that optimizing such a code was a non-trivial task as the performance was already good.

Our tests on HPC-FF with the PSI_TPP environment variable showed an improvement in run time with increasing memory bandwidth per core. This proves that the code is either cache-bound or memory-bound. However, usage of Fortran 77 style of programming, with fixed arrays sizes makes it easier for the compiler to optimize access to these arrays. Finding scope for optimization for such codes is not trivial since the code is quite efficient to begin with.

The code makes use a large number of C macros which are expanded by the pre-processor. Macro usage makes it much more difficult to analyze the program as the non-preprocessed code and the preprocessed code are very different, and the latter is hard for a human to read and understand. Nevertheless we have tried various options in changing the macros to use the cache more efficiently. However, we were not able to get any significant gains in run time speed with these changes.

We analyzed the time spent in communication in the code and found the *MPI_Waitall* to consume around 40% of the time spent in MPI calls. By aggregating the data that was sent across cores as part of the boundary exchange, we were able to reduce the number of calls to the asynchronous routines *MPI_Isend* and *MPI_Irecv*, but did not achieve much gain in the run time. From the run times and the MFLOP rate, as given by *perflib*, it emerges that the code is very well load balanced, as there is at most only a 1% difference between the highest and lowest run times.

We found ZOFLIN to perform quite well on HPC-FF and even with all the changes mentioned above, have been able to make only minor improvements to the code. Thus the proposed goal of a 50% improvement in the run time stated by the project coordinator, K. Hallatschek, was too ambitious.

9.8. References

[1] HLST Core Team Annual Report 2010.
[2] Intel Trace Analyzer and Collector.
http://www2.fz-juelich.de/jsc/docs/vendordocs/itac/doc/Getting_Started.html