EUROFUSION WPISA-REP(19) 27193

R Hatzky et al.

HLST Core Team Report 2019

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 and 2019-2020 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

This document is intended for publication in the open literature. It is made available on the clear understanding that it may not be further circulated and extracts or references may not be published prior to publication of the original when applicable, or without the consent of the Publications Officer, EUROfusion Programme Management Unit, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK or e-mail Publications.Officer@euro-fusion.org

Enquiries about Copyright and reproduction should be addressed to the Publications Officer, EUROfusion Programme Management Unit, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK or e-mail Publications.Officer@euro-fusion.org

The contents of this preprint and all other EUROfusion Preprints, Reports and Conference Papers are available to view online free at http://www.euro-fusionscipub.org. This site has full search facilities and e-mail alert options. In the JET specific papers the diagrams contained within the PDFs on this site are hyperlinked

HLST Core Team Report 2019

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 and 2019–2020 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission

Contents

1. Executive Summary 4
1.1. Progress made by each core team member on allocated projects 4
1.2. Further tasks and activities of the core team 8
1.2.1. Training 8
1.2.2. Internal training 8
1.2.3. Workshops & conferences 9
1.2.4. Publications 9
1.2.5. Meetings 9
2. Final report on HLST project MAG 10
2.1. Introduction 10
2.2. The potential equations and their discretization 11
2.3. Basic structure of AMReX 13
2.3.1. AMR process 14
2.3.2. The embedded boundary (EB) approach 16
2.4. The MLMG class 17
2.4.1. New derived class MLCellABecSec of MLLinOp 20
2.4.2. Current AMReX limitations 22
2.5. Numerical validation 22
2.1. Conclusions and outlook 26
2.2. Reference 26
3. Final Report on HLST project CINCOMP4 28
3.1. The Marconi supercomputer architecture 28
3.2. Marconi performance stability test 28
3.3. Test of the huge virtual pages (HP) queue 30
3.4. Test of the selFIe software package 31
3.5. Test of the new ticket system 31
3.6. The JFRS-1 supercomputer 31
3.6.1. JFRS-1 system specification 31
3.6.2. STREAM benchmark 32
3.6.3. Intel MPI Benchmark (IMB) 33
3.6.4. Linpack SMP benchmark 33
3.6.5. Performance stability test 34
3.7. Marconi job waiting time in the queueing system 34
3.8. Beta testing of the Intel oneAPI model 35
3.9. Conclusions 35
3.10. References 36
4. Final Report on HLST project SPICE2 37
4.1. The SPICE codes 37

4.2. Status of the code ... 37
4.3. Distributed calculation of the E-field ... 37
4.4. Performance of the complete SPICE code ... 38
4.5. Bug check ... 39
4.6. Conclusions ... 40
4.7. References ... 40
5. Final Report on HLST project PICOPT2 ... 41
5.1. Introduction ... 41
5.2. Theoretical peak AVX512 performance ... 41
5.3. Performance analysis of EUTERPE ... 43
5.4. Revisiting the EUTERPE HLST project (EHP) ... 48
5.5. Structural Changes ... 52
5.6. Software prefetching ... 54
5.7. Removing redundant computations ... 57
5.8. OpenMP 4.0 SIMD clauses ... 59
5.9. General code changes ... 62
5.10. Conclusion ... 63
5.11. Summary ... 65
5.12. Bibliography ... 65
6. Report on HLST project LCTURB ... 67
6.1. Introduction ... 67
6.2. Performance of GENE-3D ... 67
6.3. Data Compression ... 69
6.4. Outlook and conclusions ... 70
6.5. Bibliography ... 71
7. Final report on HLST project OPT-DG ... 72
7.1. Introduction ... 72
7.2. Profiling & performance analysis of Fluxo ... 72
7.2.1. Single-core profiling ... 72
7.2.2. Vectorization analysis ... 72
7.3. Reduced Fluxo ... 73
7.4. Simplified code to assess explicit SIMD vectorization ... 74
7.5. Summary and outlook ... 74
7.6. References ... 75
8. Final report on HLST project REFMUL3+ ... 76
8.1. Introduction ... 76
8.2. Parallelisation of REFMUL3 ... 76
8.3. Parallel I/O ... 77
8.3.1. HDF5 library: single-file vs. multi-file ... 77

8.3.2. HDF5 performance assessment and tuning 78
8.3.3. Hybrid simulations: impact on I/O 81
8.3.4. Check-point and restart-file 82
8.4. PDI library 83
8.5. Other performance related topics 85
8.6. Conclusions and outlook 86
8.7. References 88
9. Report on HLST project JORSOLV 89
9.1. The JOREK code 89
9.2. Current Status of the JOREK solver 89
9.3. The PASTIX solver on Marconi 89
9.4. Direct extraction of harmonic matrix blocks 90
9.5. Benchmarking 90
9.6. Performance comparison 91
9.7. Domain decomposition 91
9.8. Memory consumption in the LU decomposition 92
9.9. The complex PASTIX solver 92
9.10. Conclusions 93

1. Executive Summary

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

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

Project acronym	Core team member	Status
BIT2-3	Kab Seok Kang	running
CINCOMP4	Serhiy Mochalskyy	finished
JORSOLV	Prabal Singh Verma	running
LCTURB	Prapanch Nair	running
MAG	Kab Seok Kang	finished
OPT-DG	Tiago Ribeiro	finished
PICOPT2	Nils Moschüring	finished
REFMUL3+	Tiago Ribeiro	finished
SPICE2	Serhiy Mochalskyy	finished

Table 1 Projects and their respective responsible HLST core team members.

Roman Hatzky has been involved in the support of the European users on the CINECA computer, Marconi-Fusion. Furthermore, he was occupied in management and dissemination tasks due to his position as core team leader. In addition, he contributed to the projects of the core team.

Kab Seok Kang worked on the MAG and BIT2-3 projects.

For the MAG project, the principal investigator (PI) planned to implement a multigrid solver with adaptive mesh refinement (AMR) and/or the embedded boundary (EB) method by using publicly available software libraries. We reviewed two algebraic multigrid libraries with AMR, AMReX (from LBNL, USA) and waLBerla (from Universität Erlangen-Nürnberg, Germany), which were suggested by the PI. A decision was taken to use the AMReX library. K. S. Kang installed the AMReX library on a local machine and on Marconi. He tested the installation using the provided examples for linear systems with the adaptive geometric multigrid solver (MLMG).
AMReX provides several multigrid solvers for AMR with EB for Laplacian problems. For the MAG project, we implemented a new linear operator class, which is a derived class of the multigrid solver class of AMReX and handles general second order PDEs. This new class supports a pointwise Jacobi and a red-black Gauss-Seidel smoother and handles a Poisson problem formulated in a polar coordinate system.
We tested two approaches to solve a Poisson problem on a circular domain. One uses the multigrid solver with a polar coordinate system as a reference domain and the other uses an embedded boundary on a uniform mesh. We could achieve correct solutions for both test cases with an acceptable convergence behavior of the iterative solver.
In addition, we could document a reasonable numerical performance for the multigrid solvers on a single core. We also tested parallelization with OpenMP and MPI on a single node. The performance results of the OpenMP parallelization are good. However, the MPI parallelization has to be further improved. To improve the OpenMP/MPI performance, we have to test our implementation with various options, which are available in AMReX in combination with our new multigrid solvers.

For the BIT2-3 project, support was given in 2017 to test and implement the new 3D multigrid solver version. We were waiting for feedback from the PI, which might have triggered some adjustments.

Serhiy Mochalskyy worked on the CINCOMP4 and SPICE2 projects.

For the CINCOMP4 project, we performed different benchmarks and tests in order to determine the performance of the A2 (KNL) and A3 (Skylake) partitions of Marconi. Issues were found that significantly limit their use. Some of them were resolved by the Marconi support team, others, however, are still under investigation.
The so-called “three code benchmark” was executed regularly in order to check the stability of Marconi in terms of the execution time for real production codes. It was found that the wall clock time of codes can fluctuate significantly from one run to the next. Fluctuations in the execution time of more than 20 % were detected. It was found that the fluctuation problem is related to the MPI communication in combination with the Intel Omni-Path interconnect.
Virtual huge pages (HP) were tested on Marconi. We found that they have no influence on code performance in terms of the wall clock time. However, HP support has to be enabled on the system level and the working memory has to be divided into two parts, one for small and one for huge pages. This is a significant limitation of the usage of HP.
The SelFIe software for code profiling was installed and tested on Marconi. The software works correctly without any additional overhead. However, the most important metric for us (the performance in Flops) was absent in the output. The developers of the SelFIe software are currently fixing this problem.
A new ticket system called Request Tracker was installed on Marconi by the system administrators. Tests showed that the software works flawlessly and has a convenient and user-friendly interface.
The JFRS-1 supercomputer at Rokkasho was tested as well using different benchmarks. The machine works stably providing only small fluctuations of the wall clock time during production code computation.

During the SPICE2 project, the electric field calculation subroutine was parallelized. A parallel plane exchange subroutine was developed in order to calculate the electric field for the boundary cells. The subroutines were validated using a variety of tests with synthetic data. The solver was also tested with real data from the SPICE code providing identical results compared to the serial version of the force calculation term. An important gain in terms of memory consumption was achieved together with an overall code speed up of 7.59 %. Poisson solver scalability was shown up to 512 cores. The complete code was checked for correctness using the Forcheck static code analyzer. No errors, which could have had an influence on the results were found.

Nils Moschüring worked on the PICOPT2 project.

The PICOPT2 project focused on the gyro-kinetic PIC (GK PIC) code EUTERPE. This code is widely used in the community, for example for stellarator simulations. The aim of this project was to improve the performance of EUTERPE, mainly via the exploitation of the vectorization capabilities of the Intel Skylake architecture. The project focused on the particle pusher part of the GK PIC algorithm in EUTERPE, making ions pushed per second (ions/s) the main benchmark to gauge improvements.
In order to tackle the task of improving the performance, we first needed to establish the theoretical performance characteristics of the targeted Intel Skylake architecture, especially in regards to the AVX512 instructions, and second we needed to investigate the performance characteristics for the targeted simulation code EUTERPE. After in-depth analysis of the EUTERPE code structure and a segmented performance analysis, involving measuring the FLOPS, memory behavior, cache characteristics and creating a roofline model of different code parts, we concluded that significant performance gains might be possible. Surprisingly, the overwhelming majority of the particle pusher’s FLOPS and data requirements are spend doing field interpolation. Furthermore, the processor cache usage was especially egregious, which could be traced to details of the GK PIC algorithm.
After modelling these two aspects of the problem, a former HLST project with a similar problem description was revisited. In 2010 and 2011, HLST member Nicolaj Hammer

tried to improve the performance of EUTERPE by adapting the particle pusher for increased vector computing usage. The project failed at achieving that goal. We analyzed the adaptions, understood why they did not help in improving the performance and resolved to not re-engineer N. Hammer's changes into the current code. Nonetheless, they supplied some insight into different tactics to write vectorized code.
With the help of the elaborate models and the strong foundation in understanding the EUTERPE particle pusher, a host of modifications was tested. After thorough benchmarking a subset of these adaptions were found to actually improve the performance. These adaptions were code restructuring (4.2 % more ions/s), software prefetching (12.7 % more ions/s), general code optimizations (18.2 % more ions/s), removing redundant computations (32.4 % more ions/s) and code vectorization (82.7 % more ions/s, the previous optimization is included in this number). Enabling all adaptions together, the EUTERPE code is now able to push 89 % more ions per second, while the wall time of the supplied test case decreased by 22.8 %. This is realized without any additional parallelization and will therefore benefit simulations on a very broad scale.

Prapanch Nair worked on the LCTURB project.

Initial results give us confidence to proceed with the ZFP library for compression of data for GENE-3D. Currently, the compression-decompression cycle is performed once during the time loop. In the immediate future, we aim to use only compressed data even within the loops. Thus, each matrix-free operation within the time loop would decompress the data 'block by block' and perform computations. This would achieve a reduced memory foot-print and will enable better use of the nodes' performance. The granularity with which the data is compressed as blocks and then decompressed will also determine the memory footprint.
However, the code uses a matrix solver to solve a Poisson equation (part of Maxwell's equations). The use of a blocked compression and decompression approach would be unfeasible for the linear solver that is implemented in an external library. Hence, overall memory efficiency can only be achieved when the linear system is solved using a matrix-free approach. It is one of the tasks of the Principal Investigator (PI) to implement a matrix-free solver in the near future, which would enable the reduction of the memory footprint of GENE-3D.
In addition to ZFP, other compression libraries with block granularity will be implemented to compare the accuracy for the same compression rate. For example, the SZ library seems to be a good candidate. Other matrix compression libraries which may be more accurate for the system but which compress the entire data structure in one block will not be feasible for this effort. Thus, a comparison across feasible compression libraries will also be performed.

Tiago Ribeiro worked on the OPT-DG and REFMUL3+ projects.

The high order 3D Discontinuous Galerkin code `Fluxo` solves the 3D full MHD equations, including nonlinear and resistive terms. It has an explicit time integration and uses unstructured hexahedral meshes. The code aims to improve the scalability of 3D non-linear MHD simulations of fusion plasmas. `Fluxo` is pure-MPI parallelized and production runs of $\mathcal{O}$(10,000) MPI ranks are possible. The project OPT-DG requested an assessment of the vectorization possibly with AVX-512 instructions of `Fluxo` on the Intel Skylake architecture. A first step comprised developing a suite of very simple tests based on Intel's vectorization documentation webpages to inspect the effect of explicit vectorization on loops whose data layout in memory resembles that of `Fluxo`. The profiling of `Fluxo` followed, exposing the computationally heavy routines. An initial analysis of the compiler auto-vectorization was made with Intel Advisor. This alone allowed the introduction of some explicit OpenMP SIMD directives, which resulted in about 10 % single-core speedup of the test case and suggested

potential for further optimisations. To that end, a reduced version of Fluxo was introduced, including only the hotspot routines for a detailed assessment at the node-level. As expected, it revealed that an efficient utilisation of the wide (256 or 512 bit) vector registers in current NUMA architectures is fundamental as they enable the execution of single instructions on multiple data (SIMD). These findings were communicated to the project coordinator, who decided to submit a new HLST proposal for their implementation in Fluxo.

The REFMUL3 code, developed at the IPFN-IST, is a 3D full-wave code using the Yee scheme with full polarisation that simultaneously copes with o- and x-modes and supports a general external magnetic field and a dynamic plasma. Its parallel implementation was done within an HLST project (2016), yielding very good scalability over a few thousand cores. The development of parallel input/output (I/O) capabilities in REFMUL3 followed in the framework of another HLST project (2018). The project HLST-REFMUL3+ (2019) requested support for the assessment and extension of the latter. The corresponding activities comprised I/O bandwidth scaling measurements on Marconi and a subsequent effort to tune the existing parallel HDF5 implementation for better performance, in particular exploiting the concept of HDF5 chunked data layouts. They further comprised the implementation of a check-point and restart file infrastructure and an assessment regarding the interfacing of the I/O operations in REFMUL3 using the Portable Data Interface (PDI) library.

Prabal Singh Verma worked on the JORSOLV project.

The project aims at optimizing the solver part of the JOREK code by reducing its memory consumption and enhancing its efficiency.
Presently, the preconditioning matrix in JOREK is being constructed from a global matrix. This matrix is distributed among the MPI tasks, which makes it necessary to use expensive MPI all-to-all communication to extract the preconditioning matrix. In order to reduce the computational cost, we are extracting the preconditioning matrix directly from the elementary matrix as blocks of harmonics. Each harmonic block can be addressed by one or more MPI tasks. Nevertheless, in the beginning, we have only considered one MPI task per harmonic block and found that for realistic problems, the direct extraction (without domain decomposition) exhibits much higher computational costs in comparison to the MPI all-to-all communication. Thus, the domain decomposition is necessary to increase the efficiency of the solver, i.e., each harmonic block needs to be addressed by multiple MPI tasks.
In the present form, the JOREK solver does not support this domain decomposition because the currently used PASTIX solver pastix_fortran() cannot solve distributed matrices. Therefore, a distributed PASTIX solver dpastix_fortran() needs to be implemented in JOREK. However, there seems to be a bug in the distributed PASTIX library, which has recently been reported to the PASTIX team. Once the bug is fixed, we will implement dpastix_fortran()in JOREK, and compare the results.
At the same time, we are working on lowering the memory consumption. The memory consumption can be reduced by converting the real-valued preconditioning matrix into a complex one and then employing the complex PASTIX solver for it. However, this conversion is only possible if the real matrix exhibits the desired symmetry, i.e., diagonal elements should be identical and off-diagonal elements should only have opposite signs. We have found that the symmetry, which exists analytically, may be broken by numerical errors. Therefore, the real matrix can no longer be converted into a complex one. Nevertheless, it has been confirmed that if the symmetry is lost due to numerical errors, one can enforce it again, by replacing the diagonal and off-diagonal entries by their respective average values, without meaningfully affecting the results. Thus, the next steps are to convert the preconditioning matrix into a complex one, employ the complex PASTIX solver, and finally compare the results.

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

1.2.1. Training

Nils Moschüring has visited:

- The project coordinator Ralf Kleiber to work for the PICOPT2 project, 24th–28th June 2019, IPP, Greifswald, Germany.
- The project coordinator Ralf Kleiber to work for the PICOPT2 project, 4th–8th November 2019, IPP, Greifswald, Germany.

1.2.2. Internal training

Kab Seok Kang has attended:

- Python for HPC, 29th–30th April 2019, MPCDF, Garching, Germany.
- Advanced C++ with Focus on Software Engineering, 12th–14th June 2019, LRZ, Garching, Germany.
- MaxEnt2019, 30th June–5th July 2019, IPP, Garching, Germany.
- ExaHyPE workshop, 22nd–26th July 2019, LRZ, Garching, Germany.
- PACO 2019 - Workshop on Power-Aware Computing, 5th–6th November 2019, Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany.
- METT VIII – 8th Workshop on Matrix Equations and Tensor Techniques, 6th–8th November 2019, Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany.

Serhiy Mochalskyy has attended:

- IPP ITED laboratory Ringberg meeting, 1st–3rd April 2019, Ringberg, Germany.
- Python for HPC, 29th–30th April 2019, MPCDF, Garching, Germany.
- Deep Learning and GPU programming using OpenACC, 15th–17th July 2019, HLRS, Stuttgart, Germany.
- Introduction to GPU programming using OpenACC, FZJ, 28th–29th October 2019, Jülich, Germany.
- Advanced HPC Workshop, MPCDF 12th–13th November 2019, Garching, Germany.

Nils Moschüring has attended:

- Performance Portability Programming with Kokkos, LRZ, 24th October, Garching, Germany.

Prapanch Nair has attended:

- Advanced HPC Workshop, 12th–13th November 2019, MPCDF, Garching, Germany.
- Node-Level Performance Engineering, LRZ, 3rd–4th December 2019, Garching, Germany.

Tiago Ribeiro has attended:

- EMEA Intel AI DevCon 2019, 23rd January 2019, International Congress Center Munich (ICM), Germany.
- High Performance Parallel IO and post-processing, 11th–13th March 2019, Maison de la Simulation, CEA Saclay, France.
- Introduction to GPU programming using OpenACC, FZJ, October 28th–29th 2019, Jülich, Germany.
- Advanced HPC Workshop, MPCDF, 12th–13th November 2019, Garching, Germany.
- Node-Level Performance Engineering, LRZ, 3rd–4th December 2019, Garching, Germany.

Prabal Singh Verma has attended:

- Python for HPC, 29th–30th April 2019, MPCDF, Garching, Germany.
- HPC code optimisation workshop, 20th–22nd May 2019, LRZ, Garching, Germany.
- Node-Level Performance Engineering, 27th–28th June 2019, HLRS, Stuttgart, Germany.
- Advanced HPC Workshop, 12th–13th November 2019, MPCDF, Garching, Germany.
- Parallelization with MPI and OpenMP, 25th–27th November 2019, University of Göttingen, Göttingen, Germany.

1.2.3. Workshops & conferences

Parabal Singh Verma has attended:
JOREK development meeting, 21st November 2019, CEA, Cadarache, France.

1.2.4. Publications

Da Silva, F., Heuraux, S., Ricardo, E., Silva, A. and Ribeiro, T.: Modelling reflectometry diagnostics: finite-difference time-domain simulation of reflectometry in fusion plasmas, JINST, **14** 2019, C08003.

Da Silva, F., Heuraux, S., Ricardo, E., Silva, A. and Ribeiro, T.: Benchmarking 2D against 3D FDTD codes in the assessment of reflectometry performance in fusion devices, JINST, **14** 2019, C08004.

Da Silva, F., Ferreira, J., De Masi, G., Heuraux, S., Ricardo, E., Ribeiro, T., Tudisco, O., Cavazzana, R., D'Arcangelo, O. and Silva, A.: A first full wave simulation assessment of reflectometry for DTT, JINST, **14**, 2019, C08011.

Hatzky, R., Kleiber, R. Könies, A., Mishchenko, A., Borchardt, M., Bottino, A. and Sonnendrücker, E.: Reduction of the statistical error in electromagnetic gyrokinetic particle-in-cell simulations, J. Plasma Phys., **85** (1), 2019, 1–69.

1.2.5. Meetings

Roman Hatzky attended on a regular basis:

- HPC Operation Committee meeting
- EUROfusion Marconi Ticket Meeting

2. Final report on HLST project MAG

2.1. *Introduction*

Predicting the performance of fusion plasmas in terms of the amplification factor, which is given by the ratio of the fusion power over the injected power, is among the key challenges in fusion plasma physics. In this perspective, turbulence and heat transport are being modelled with the most accurate theoretical framework, using first-principle non-linear simulation tools. Several parallel gyrokinetic codes, that solve the coupled set of the Vlasov and quasi-neutrality equations, provide answers and insights to better comprehend the plasma behavior.

The GYSELA code is currently based on a simplified magnetic configuration with circular concentric magnetic field lines. One of the next objectives is to extend the code to more realistic magnetic configurations: *D*-shape configurations in the core in the short term but also *X*-point configurations in the long term. This will require changing both the semi-Lagrangian scheme for the *5D* Vlasov equation and the quasi-neutrality (based on a modified *2D* Poisson equation) solver. The GEMPIC code, which is a fully kinetic PIC code that is being extended to realistic tokamak geometries, also needs a *2D* Poisson solver for general magnetic geometries in the poloidal plane. Both solvers fit in the category of general elliptic solvers and are solved in the same geometry. Therefore the development of the solver can benefit both codes.

There are at least two possible options for this solver: one uses a stretched polar grid and the other uses a locally refined Cartesian grid. General elliptic solvers in *2D* are needed for solving the gyrokinetic Poisson equation, coming from the quasi-neutrality equation, as well as for the standard Poisson equation that is needed to get the initial electric field from the initial particle positions in a fully kinetic PIC code. In a realistic tokamak geometry, there are two natural options. On the one hand, one can use a flux surface aligned poloidal grid, which has drawbacks at the polar point as well as at the *X*-point where local refinement would be needed when dealing with edge and SOL problems. Moreover, the geometry is more complex. On the other hand one can use a Cartesian grid which does not need to deal with the metric coefficients, but would need to be locally refined in the strong gradient regions at the edge and across the separatrix. Both have pros and cons that should be compared, before implementing one of them in a production code.

In this project, we focus on the development of a *2D* elliptic solver on an adaptive locally refined Cartesian mesh in the poloidal plane. To use a Cartesian mesh, we require an embedded boundary method approach, which allows us to handle a more complex domain like the tokamak wall. The aim of this project is to design, develop and evaluate a new *2D* Poisson solver based on the geometric multigrid method with embedded boundaries to compete with the *2D* finite difference solver on a polar grid. This *2D* geometric multigrid solver for an adaptive Cartesian mesh will be implemented as a standalone program independent from the GYSELA or GEMPIC codes. As the solver will be defined on a Cartesian grid a refined region will be needed at the edge of the tokamak and in the vicinity of the separatrix and the *X*-point as well as close to the boundary, where an embedded boundary method will be needed to handle the wall which will not be aligned with the grid. Specific parts of the standard multigrid method need to be modified to handle adaptive grids, i.e., locally refined meshes that can be changed between iterations, even though they will be fixed for each Poisson solve at a given time. The implementation of the multigrid solver will be used with the open source block structured AMR software framework AMReX that fits the needs of GEMPIC and GYSELA well.

We will provide a comparison of the numerical solution of this new multigrid solver with the finite difference solution described in the reference paper. Also, we will compare the asymptotic complexity and execution time with the original Poisson solver in order to show the benefits of this approach. In the GYSELA code, the Poisson solver does not use the standard *2D* Laplacian operator but a modified Laplacian that depends on

a radial profile of the temperature. The needed modifications to the 2D multigrid solver will be described and implemented in the following.

2.2. *The potential equations and their discretization*

In GYSELA, the normalized quasi-neutrality equation can be written in the following way

$$-\sum_i \mathcal{A}_i \nabla_\perp \cdot (\mathcal{B}_i \nabla_\perp \phi) + \mathcal{C}\left[\phi - \lambda \langle \phi \rangle_{FS}\right] = f, \tag{2.1}$$

where the integral $\langle \phi \rangle_{FS}$ represents the flux surface average of ϕ, i.e.,

$$\langle \phi \rangle_{FS}(r) = \frac{\int \phi \mathcal{J}_x d\theta d\varphi}{\int \mathcal{J}_x d\theta d\varphi}, \qquad \forall \phi(r, \theta, \varphi) \ ,$$

with $\mathcal{J}_x = 1/(B \cdot \nabla \theta)$, the normalized Jacobian space. The parameter λ has been added for testing and can be chosen to be 1 or 0.

The normalized quasi-neutrality equation in the case of adiabatic electrons, for example, is given by

$$\mathcal{A}_i = \frac{A_i}{n_{e_0}}, \quad \mathcal{B}_i = \frac{n_{i_0}}{B_0}, \quad \mathcal{C} = \frac{1}{Z_0^2 T_e}, \quad f = \frac{1}{n_{e_0}} \sum_i Z_i [n_{G_i} - n_{G_i,eq}],$$

where the normalized electron density n_{e_0} is defined as $n_{e_0} = \sum_i Z_0 Z_i n_{i_0}$. For $\mathcal{A}_i = A_s, \mathcal{B}_i = n_{s_0}, \mathcal{C} = 0$, and $f = \sum_s Z_s \int J_{0_s}(f_s - f_{s0}) \mathbf{d}^3 v$, we get the normalized quasi-neutrality equation in the case of fully kinetic electrons.

For the case of trapped kinetic electrons, the normalized quasi-neutrality equation can be written differently for different versions. The ICNSP version is defined by

$$\mathcal{A}_i = A_i, \quad \mathcal{A}_e = A_e, \quad \mathcal{B}_i = n_{i_0}, \quad \mathcal{B}_i = \bar{\alpha}_{t0} n_{e_0}, \quad \mathcal{C} = \frac{(1 - \bar{\alpha}_{t0}) n_{e_0}}{Z_0^2 T_e},$$

$$f = \sum_i Z_i \int J_{0_i}(f_i - f_{i0}) \mathbf{d}^3 v - \int_{\text{trap.}} J_{0_e}(f_e - f_{e0}) \mathbf{d}^3 v,$$

with

$$f_{s0} = \frac{n_{s_0}}{(2\pi T_{s_0})^{3/2}} \exp\left(-\frac{E_s}{T_{s_0}}\right), \quad E_s = \frac{1}{2} v_{G||s}^2 + \mu_s B(r, \theta),$$

where the radial density and temperature profiles for the ions (n_{s_0} and T_{s_0}) are known values while the radial electron temperature profile is defined homothetically to that of the main ions.

For each case, we solve equation (1.1) on a sphere-shaped domain with $\Omega_{r_{\min}} \times [0, L_\varphi]$ where $\Omega_{r_{\min}} = \{(r, \theta) : r_{\min} \le r \le r_{\max}; 0 \le \theta < 2\pi\}$ with proper boundary conditions. We need to compute $\langle \phi \rangle_{FS}$ on $(\theta, \varphi) \in [0, 2\pi] \times [0, L_\varphi]$. Using a decoupling approach, $\langle \phi \rangle(r) = \langle \phi \rangle_{FS}(r)$ can be obtained by solving the *1D* equation

$$\left[-\sum_i \mathcal{A}_i \mathcal{B}_i \left\{\frac{\partial^2}{\partial r^2} + \left(\frac{1}{r} + \frac{1}{\mathcal{B}_i} \frac{d\mathcal{B}_i}{dr}\right) \frac{\partial}{\partial r}\right\} + C\right] \langle \phi \rangle(r) = \langle f \rangle(r). \tag{1.2}$$

Then equation (1.1) becomes

$$-\sum_i \mathcal{A}_i \nabla_\perp \cdot (\mathcal{B}_i \nabla_\perp \Phi) + \mathcal{C}\Phi = f - \langle f \rangle(r). \tag{1.3}$$

Summing up, the algorithm to calculate the solution of equation (1.1) consists of the following steps:

- Solve the *1D* equation (1.2) to get $\langle\phi\rangle$ (r)
- Solve the *2D* equation (1.3) for all φ to get $\Phi(r, \theta, \varphi)$
- Compute $\phi = \Phi + \langle\phi\rangle$

In the following, we will describe the way to solve the *2D* equation (1.3). In realistic tokamak geometry, there are two natural options. One approach uses a flux surface aligned poloidal grid, but this has drawbacks at the polar point as well as at the *X*-point, where a local refinement would be needed when dealing with edge and SOL problems. This approach can be described as using the conformal mapping $\mathcal{F}$ from $\Omega_{r_{\min}}$ to the rectangular reference domain, i.e., using the polar coordinate system and using an adaptive mesh refinement (AMR) near the outer boundary as shown in Fig. 1. This approach has a natural way to compute $\langle f\rangle(r)$ on $[0,2\pi]\times[0,L_\varphi]$ for a given $r\in[r_{\min},r_{\max}]$ in (1.2), which is a 1D problem and can be solved with a direct method. The equation (1.3) becomes

$$\left[-\sum_i \mathcal{A}_i\mathcal{B}_i\left\{\frac{\partial^2}{\partial r^2}+\left(\frac{1}{r}+\frac{1}{\mathcal{B}_i}\frac{d\mathcal{B}_i}{dr}\right)\frac{\partial}{\partial r}\right\}+\frac{1}{r^2}\frac{\partial^2}{\partial\theta^2}+C\right]\Phi(r,\theta) = f(r,\theta)-\langle f\rangle(r) \quad . \tag{1.4}$$

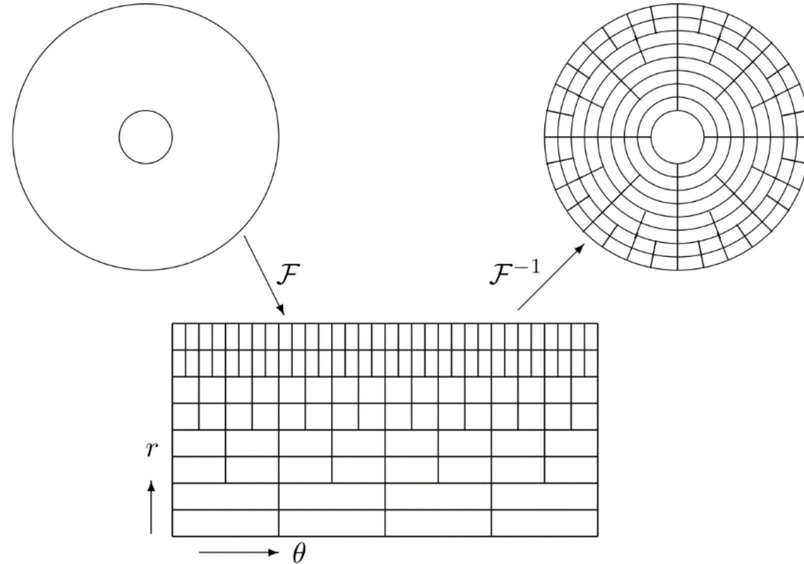

Fig. 1 Current discretization with adaptive mesh refinement.

The current implementation for solving equation (1.4) is realized with a direct method for small problem sizes. Our goal is to apply the multigrid solver MLMG in AMReX to solve larger problems. The MLMG is a robust and parallelized geometric multigrid solver with AMR in AMReX.

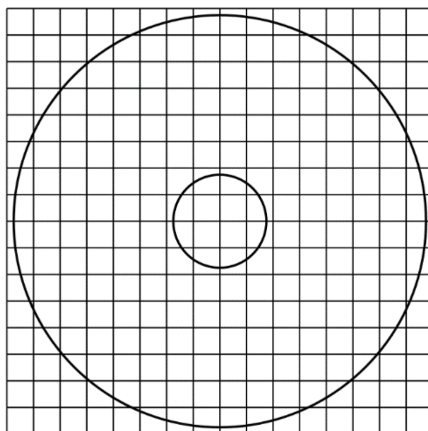

Fig. 2 Uniform Cartesian grid with embedded boundary method.

Another approach we want to pursue is to use the embedded boundary method for the complex domain with a uniform Cartesian mesh as shown in Fig. 2. This approach does not need to deal with the metric coefficients, but would need to be locally refined in the strong gradient regions at the edge and across the separatrix using block-structured AMR. The solver for equation (1.3) is a typical *2D* elliptic solver, but we additionally need a computing routine for $\langle f \rangle (r)$ and $\langle \phi \rangle$ on the grid points of the uniform Cartesian mesh.

2.3. *Basic structure of AMReX*

As a framework for developing the geometric multigrid solver, we use the open source library AMReX. AMReX [1] was developed at LBNL, NREL, and ANL as part of the Block-Structured AMR Co-Design Center in DOE's Exascale Computing Project. It is a publicly available software framework designed for building massively parallel block-structured adaptive mesh refinement (AMR) applications.

Key features of AMReX include:

- C++ and Fortran interfaces
- 1-, 2- and 3-*D* support
- Support for cell-centered, face-entered, edge-centered, and nodal data
- Support for hyperbolic, parabolic, and elliptic solvers on hierarchical adaptive grid structure
- Optional subcycling in time for time-dependent PDEs
- Support for particles
- Support for embedded boundary (cut cell) representations of complex geometries
- Parallelization via flat MPI, OpenMP, hybrid MPI/OpenMP, or MPI/MPI
- Parallel I/O
- Plotfile format supported by AmrVis, VisIt, ParaView, and yt

AMReX consists of three main parts:

- Handling of geometric data and vectors on the geometry:
  Storing information about the grid layout and processor distribution mapping at each level of refinement. Functions to create grids at different levels of refinement, including tagging operations. Flux registers used to store and manipulate fluxes at coarse-fine interfaces.
  Box, IntVect, IndexType, RealBox, Geometry, BoxArray, BaseFab, DistributionMapping, FArrayBox, IArrayBox, FabArray, MultiFab, iMultiFab, MFIter, Tiling, ...
- Adaptive geometric multigrid linear solver (MLMG):

Operations on data at different levels of refinement, such as interpolation and restriction operators. AMReX can use PETSc and HYPRE BoomerAMG as a bottom-solver.
MLMG, MLLinOp, LPInfo, MLCGSolver, MLABecLaplacian, MLPoisson, MLNodeLaplacian, ...

- Support for particles and visualization:
  The particle classes are suited for particles that interact with data defined on a (possibly adaptive) block-structured hierarchy of meshes for Particle-in-Cell (PIC) simulations, Lagrangian tracers, or particles that exert drag forces onto a fluid, such as in multi-phase flow calculations. The goal of AMReX's particle tools is to allow users flexibility in specifying how the particle data is laid out in memory and how to handle the parallel communication of particle data. There are several visualization tools that can be used for AMReX profiles. AmrVis is a standard tool used within the AMReX-community and is designed specifically for highly efficient visualization of block-structured hierarchical AMR data. Files in the plotfile format can also be viewed using the VisIt, ParaView, and yt packages. Particle data can be viewed using ParaView.

2.3.1. AMR process

The AMReX library provides tools and functions for time-stepping simulations with a single-level and multiple levels of refinement. To define the adaptive refined grid, we can use the **AmrCore** class which is derived from the **AmrMesh** class. The **AmrMesh** class can be thought of as a container to store arrays of **Geometry**, **DistributionMapping**, and **BoxArray** (one of each level) instances, as well as information about the current grid structure. The protected data members of the **AmrMesh** class are:

```
protected:
    int                     verbose;
    int                     max_level;
    Vector<InrVect>         ref_ratio;
    int                     finest_level;
    Vector<InrVect>         n_error_buf;
    Vector<InrVect>         blocking_factor;
    Vector<InrVect>         max_grid_size;
    Real                    grid_eff;
    int                     n_proper;
    bool                    use_fixed_coarse_grids;
    int                     use_fixed_upto_level;
    bool                    refine_grid_layout;

    Vector<Geometry>                    geom;
    Vector<DistributionMapping>         dmap;
    Vector<BoxArray>                    grids;
```

The **AmrCore** class is an abstract class and does not actually have any data members, just member functions, some of which override the base class **AmrMesh**. There are no pure virtual functions in **AmrMesh**, but there are five pure virtual functions in the **AmrCore** class. We have to implement these five functions as shown in Fig. **3**. The **TagBox** and **Cluster** classes are used in the grid generation process, but these classes and their member functions are largely hidden from any application codes through simple interfaces such as **regrid()** and **ErrorEst()**. We define a derived class

AmrCoreMAG of the **AmrCore** class, which handles our adaptive refined mesh for the geometric multigrid solver.

```
//! Tag cells for refinement.
virtual void ErrorEst(int lev, TagBoxArray& tags, Real time,
                      int ngrow) override = 0;

//! Make a new level from scaratch using provided BoxArray
//! and DistbutionMapping
virtual void MakeNewLevelFromScratch(int lev, Real time,
             const BoxArray & ba, const DistributionMapping& dm)
             override = 0;

//! Make a new level using provided BoxArray and DistbutionMapping
//! and fill with interpolated coarse level data.
virtual void MakeNewLevelFromCoarse(int lev, Real time,
             const BoxArray & ba, const DistributionMapping& dm)
             override = 0;

//! Remake an existing level using provided BoxArray and
//! DistbutionMapping and fill with interpolated coarse level data.
virtual void RemakeLevel(int lev, Real time, const BoxArray & ba,
             const DistributionMapping& dm) override = 0;

//! Delete level data.
virtual void ClearLevel(int lev) override = 0;
```

Fig. 3 The five virtual functions in the **AmrCore** class.

For another approach of a fixed refinement at the beginning, we found a simple implementation to make AMR with **domain.growLo()** and **domain.refine().** The following source code is for our simple mesh refinement:

```
RealBox rb({rmin,0.}, {rmax,tpi});
Array<int,2> is_periodic{0,1};
Geometry::Setup(&rb, 0, is_periodic.data());
Box domain0(IntVect{0,0},IntVect{n_cellx-1,n_celly-1});
Box domain = domain0;
for (int ilev = 0; ilev < nlevels; ++ilev) {
  geom[ilev].define(domain);
  domain.refine(ref_ratio);
}
domain = domain0;
for (int ilev = 0; ilev < nlevels; ++ilev) {
  grids[ilev].define(domain);
  grids[ilev].maxSize(max_grid_size);
  domain.growLo(0,-n_cellx/2);
  domain.refine(ref_ratio);
}
for (int ilev = 0; ilev < nlevels; ++ilev) {
  dmap[ilev].define(grids[ilev]);
  solution[ilev].define(grids[ilev], dmap[ilev], 1, 1);
}
```

Using this algorithm, we get a refined mesh as shown in Fig. 4.

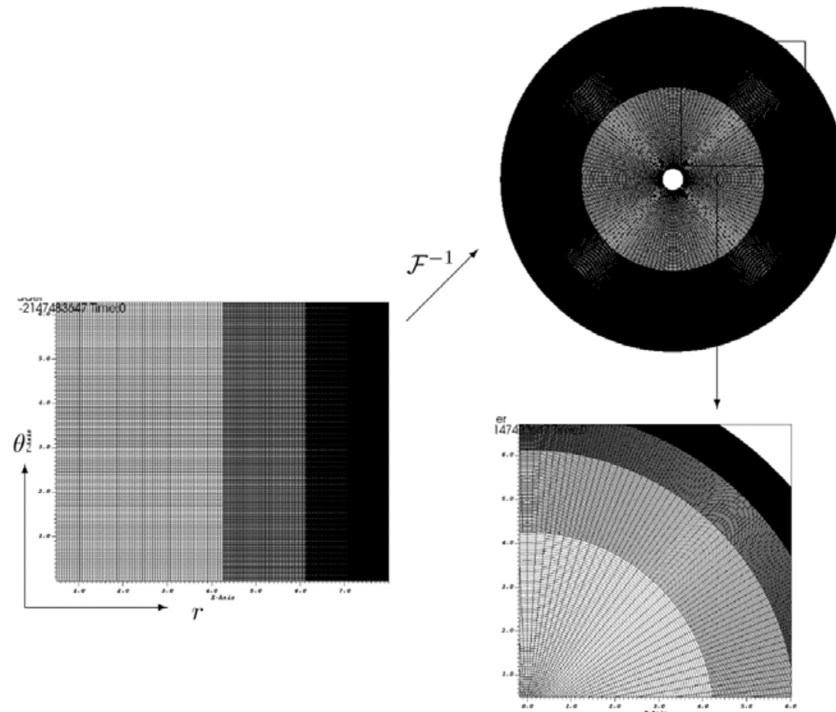

Fig. 4 A simple AMR for a rectangular reference domain and a circular domain with a polar transformation.

2.3.2. The embedded boundary (EB) approach

For computations with complex geometries, AMReX provides data structures and algorithms to employ an embedded boundary (EB) approach in the discretization of PDE. For this approach, the underlying computational mesh is uniform and block-structured, but the boundary of the irregular-shaped computational domain conceptually cuts through this mesh (See Fig. 2). Each cell in the mesh is labeled as regular, cut, or covered, and the computing routine of the finite volume based discretization methods, traditionally used in the AMReX applications, is modified to incorporate these cell shapes. Because this is a relatively simple grid generation technique, the grid for complex geometries is generated quickly and robustly. However, the technique can produce arbitrarily small cut cells in the domain. Practically, these small cells can have significant impact on the robustness and stability of traditional finite volume methods.

A geometry is discretely represented by volumes (*V*) and apertures (*A*) at the AMR level. Without multivalued cells, the volume fractions, area fraction and cell and face centroids (see Fig. 5) are the only geometric information needed to compute second-order fluxes centered at the face centroids, and to infer the connectivity of the cells.

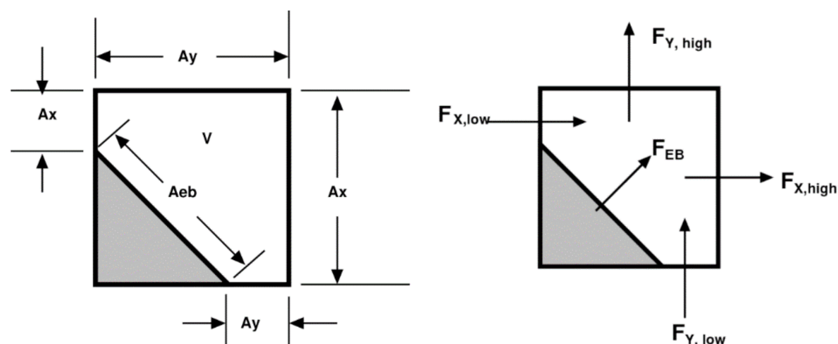

Fig. 5 Representation of a small cell in the EB approach (from Table 12.1 in [1]).

In AMReX the geometric information is stored in a distributed database class that must be initialized at the start of the calculation. The procedure for this is as follows:

- Define an implicit function class of position that describes the surface of the embedded object. Specifically, the function class must have a public member function that takes a position and returns a negative value if that position is inside the fluid, a positive value if it is in the body, and zero if it is at the embedded boundary.
- Make an **EB2::GeometryShop** object using the implicit function.
- Build an **EB2::IndexSpace** with the **EB2::GeometryShop** object and a **Geometry** object that contains the information about the domain and the mesh.

AMReX provides a number of predefined implicit function classes for basic shapes and a number of transformation operations to apply to an object. After initializing the EB database, we build a **EBFArrayBoxFactory**. This object provides access to the EB database in the format of basic AMReX objects such as **BaseFab**, **FArrayBox**, **FabArray**, and **MultiFab**. In Fig. 6 there is an example of initializing the database for a shape and a construction of a **EBFArrayBoxFactory,** which is required for the **MultiFab** (object called **solution**).

```
RealBox rb({-4.2,-4.2}, {4.2,4.2});
Array<int,2> is_periodic{1,1};
Box domain(IntVect{0,0},IntVect{n_cell-1,n_cell-1});
geom.define(domain,rb,CoordSys::cartesian,is_periodic);
grids.define(domain);
grids.maxSize(max_grid_size);
dmap.define(grids);
int required_coarsening_level = 0;
int max_coarsening_level = 100;
RealArray center{0.0,0.0};
Real radius = 4.0;
bool has_fluid_inside = 1;
int ngrow = 4;
EB2::SphereIF sf(radius, center, has_fluid_inside);
EB2::SphereIF sf0(0.5, center, 0);
auto twospheres = EB2::makeUnion(sf,sf0);
auto gshop = makeShop(twospheres);
EB2::Build(gshop, geom, required_coarsening_level,
                  max_coarsening_level, ngrow);
const EB2::IndexSpace& eb_is = EB2::IndexSpace::top();
const EB2::Level& eb_level = eb_is.getLevel(geom);
EBSupport ebs = EBSupport::full;
Vector<int> ng_ebs = {2,2,2};
EBFArrayBoxFactory ebfac(eb_level, geom, grids, dmap,
                         ng_ebs, ebs);
solution.define(grids, dmap, 1, 0, MFInfo(), ebfac);
```

Fig. 6 An example of the EB approach.

2.4. *The MLMG class*

In this project, we implement the adaptive geometric multigrid solver for linear systems (MLMG) of AMReX for the GEMPIC and GYSELA codes.

In AMReX, the **MLMG** class is responsible for the adaptive geometric multigrid linear solver. This class has all required information and basic routines for the multigrid solver

including matrix-vector multiplication (apply), smoother, computing residual, restriction, prolongation, and a coarsest level solver (bottom-solver). The main difficulty in implementing a multigrid solver is that the basic routines in the multigrid solver have to fit to the problem and the resulting discretizations. The constructor of the **MLMG** class has only one argument which must be an instance of the **MLLinOP** class. The constructor and the basic functions in the **MLMG** class call functions of the **MLLinOP** class, which is referenced as **linop** in the **MLMG** class. We present two important functions, **mgVcycle** and **computeResidual**, of the **MLMG** class in Fig. 7 and Fig. 8. Currently, the bottomsolver can use one of the following simple iterative solvers (smoothers): BICGSTAB, CG, and solvers from other libraries such as HYPRE and PETSc.

```
// in   : Residual (res)
// out  : Correction (cor) from bottom to this function's local top
void MLMG::mgVcycle (int amrlev, int mglev_top) {
  const int mglev_bottom = linop.NMGLevels(amrlev) - 1;
  for (int mglev = mglev_top; mglev < mglev_bottom; ++mglev) {
    cor[amrlev][mglev]->setVal(0.0);
    bool skip_fillboundary = true;
    for (int i = 0; i < nu1; ++i) {
      linop.smooth(amrlev, mglev, *cor[amrlev][mglev],
                   res[amrlev][mglev],skip_fillboundary);
      skip_fillboundary = false;
    }
    computeResOfCorrection(amrlev, mglev);
    linop.restriction(amrlev, mglev+1, res[amrlev][mglev+1],
                     rescor[amrlev][mglev]);
  }
  if (amrlev == 0){ bottomSolve(); }
  else {
    cor[amrlev][mglev_bottom]->setVal(0.0);
    bool skip_fillboundary = true;
    for (int i = 0; i < nu1; ++i) {
      linop.smooth(amrlev, mglev_bottom, *cor[amrlev][mglev_bottom],
                  res[amrlev][mglev_bottom],skip_fillboundary);
      skip_fillboundary = false;
    }
  }
  for (int mglev = mglev_bottom-1; mglev >= mglev_top; --mglev) {
    addInterpCorrection(amrlev, mglev);
    for (int i = 0; i < nu2; ++i) {
      linop.smooth(amrlev, mglev, *cor[amrlev][mglev],
                  res[amrlev][mglev]);
    }
  }
}
```

Fig. 7 The basic cycle of the multigrid solver: **MLMG::mgVcycle()**.

```
// Compute single AMR level residual without masking.
void MLMG::computeResidual (int alev) {
    MultiFab& x = *sol[alev];
    const MultiFab& b = rhs[alev];
    MultiFab& r = res[alev][0];
    const MultiFab* crse_bcdata = nullptr;
    if (alev > 0) {
        crse_bcdata = sol[alev-1];
    }
    linop.solutionResidual(alev, r, x, b, crse_bcdata);
}
```

Fig. 8 A typical example of a function in the **MLMG** class, which performs basic operations for the multigrid solver.

As explained above, the **MLLinOP** class plays a critical role in the **MLMG** class which is to construct the solver according to the considered problem. However, the **MLLinOP** class is an abstract class which only has pure virtual functions that are called by functions in the **MLMG** class. The implementation of these virtual functions in the **MLLinOP** class are done in a derived class of the **MLLinOP** class. AMReX has some derived classes for the discretization of second order partial differential equations. We summarize the derived classes (except MLCellABecSec) of the **MLLinOP** class in the current version of AMReX in Fig. 9. In these derived classes, the MLEBABecLap class supports the EB approach.

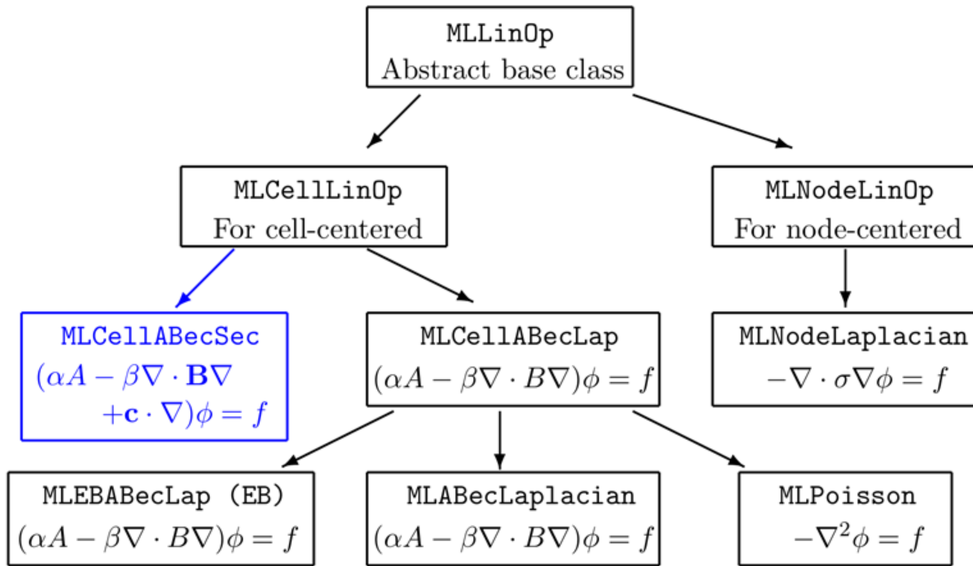

Fig. 9 The abstract base class **MLLinOp** and some of its derived classes.

The arguments of the constructor of these linear operator classes are shown in Fig. 10. The first three arguments of it are related to the domain of the problem, including the geometric shape, mesh sizes, and distribution of the domain to the computational nodes. The next argument has the information about the multigrid solver and the last argument gives the type of the solution. These linear operator classes also support the EB approach. We noticed that the AMReX developer team is still working on EB and learned the usage of EB through the given examples.

```
MLNodeLaplacian (
  const Vector<Geometry>& a_geom,
  const Vector<BoxArray>& a_grids,
  const Vector<DistributionMapping>& a_dmap,
  const LPInfo& a_info = LPInfo(),
  const Vector<FabFactory<FArrayBox> const *>& a_factory = {}
);
```

Fig. 10 The arguments of the constructors of the linear operator classes, exemplified by the **MLNodeLaplacian** constructor definition.

2.4.1. New derived class MLCellABecSec of MLLinOp

In the linear operator classes derived from **MLLinOp**, we need to implement the required functions according to a given PDE and its discretization. In Fig. 11, we present the declarations of some functions that have crucial roles in the multigrid solver. AMReX uses C++ and Fortran for its implementation. These functions are independent of the dimension and require computationally intensive functions, which are implemented separately for each dimension and which are implemented in Fortran.

```
virtual void restriction (int amrlev, int cmglev, MultiFab& crse,
                          MultiFab& fine);
virtual void interpolation (int amrlev, int fmglev, MultiFab& fine,
                          const MultiFab& crse);
virtual void averageDownSolutionRHS (int camrlev, MultiFab& crse_sol,
                        MultiFab& crse_rhs,const MultiFab& fine_sol,
                        const MultiFab& fine_rhs);
virtual void apply (int amrlev, int mglev, MultiFab& out, MultiFab& in,
                    BCMode bc_mode,StateMode s_mode,
                    const MLMGBndry* bndry=nullptr);
virtual void smooth (int amrlev, int mglev, MultiFab& sol,
                     const MultiFab& rhs,bool skip_fillboundary=false);
```

Fig. 11 The virtual functions of the linear operator classes.

Among the functions in Fig. 11, some functions such as **restriction()**, **interpolation()**, and **averageDownSolutionRHS()** depend only on the discretization, i.e., cell-centered or nodal-centered. Other functions such as **apply()** and **smooth()** depend on the discretization and the targeted problems.

In **MLCellABecLap** and **MLABecLaplacian**, the targeted problem is

$$(\alpha A - \beta \nabla \cdot B \nabla)\phi = f$$

where α and β are constants and A and B are scalar functions on the domain. Our polar coordinated Laplacian problem does not fit to this. Therefore, we need to consider more general problem cases. We implemented the new derived class **MLCellABecSec** from **MLCellLinOp** to solve the following general second order PDEs

$$(\alpha A - \beta \nabla \cdot \mathbf{B} \nabla + \mathbf{c} \cdot \nabla)\phi = f$$

where α and β are constants, A is a scalar function, $\mathbf{B}$ is a matrix function and $\mathbf{c}$ is a vector function on the domain. The main difference between the new implementation and **MLCellABecLap** is the handling of these coefficients. We found that AMReX has most of the required functions to handle coefficients except the matrix coefficients, which can be easily implemented by using functions which handle vector coefficients.

Our implementation is based on the implementation of **MLCellABecLap,** which is a cell-centered method, i.e.,

$$(\nabla \cdot \mathbf{F})_{i,j} = \frac{1}{\Delta x}[(\mathbf{F})^{\bullet}_{i+1,j} - (\mathbf{F})^{\bullet}_{i,j}] + \frac{1}{\Delta y}[(\mathbf{F})^{\circ}_{i,j+1} - (\mathbf{F})^{\circ}_{i,j}]$$

$$\left(\frac{\partial \phi}{\partial x}\right)^{\bullet}_{i,j} = \frac{1}{\Delta x}(\phi_{i,j} - \phi_{i-1,j}) \qquad \phi^{\bullet}_{i,j} = \frac{1}{2}(\phi_{i,j} + \phi_{i-1,j})$$

$$\left(\frac{\partial \phi}{\partial x}\right)^{\circ}_{i,j} = \frac{1}{\Delta y}(\phi_{i,j} - \phi_{i,j-1}) \qquad \phi^{\circ}_{i,j} = \frac{1}{2}(\phi_{i,j} + \phi_{i,j-1})$$

$$\left(\frac{\partial \phi}{\partial x}\right)_{i,j} = \frac{1}{\Delta x}(\phi^{\bullet}_{i+1,j} - \phi^{\bullet}_{i,j})$$

$$\left(\frac{\partial \phi}{\partial x}\right)_{i,j} = \frac{1}{\Delta y}(\phi^{\circ}_{i,j+1} - \phi^{\circ}_{i,j})$$

for $\mathbf{F} = B\nabla\phi$ and $\mathbf{F} = \mathbf{B}\nabla\phi$ with index as shown in Fig. 12. For **MLABecLaplacian** we then have

$$\alpha A_{i,j}\phi_{i,j} - \frac{\beta}{(\Delta x)^2}\left[B^{\bullet}_{i+1,j}(\phi_{i+1,j} - \phi_{i,j}) - B^{\bullet}_{i,j}(\phi_{i,j} - \phi_{i-1,j})\right]$$
$$- \frac{\beta}{(\Delta y)^2}\left[B^{\circ}_{i,j+1}(\phi_{i,j+1} - \phi_{i,j}) - B^{\circ}_{i,j}(\phi_{i,j} - \phi_{i,j-1})\right] = f_{i,j}$$

and for **MLCellABecSec**

$$\begin{aligned}
\alpha A_{i,j}\phi_{i,j} &- \frac{\beta}{(\Delta x)^2}\left[B^{xx\bullet}_{i+1,j}(\phi_{i+1,j} - \phi_{i,j}) - B^{xx\bullet}_{i,j}(\phi_{i,j} - \phi_{i-1,j})\right] \\
&- \frac{\beta}{4\Delta x\Delta y}\Big[B^{yx\bullet}_{i+1,j}(\phi_{i+1,j+1} + \phi_{i,j+1} - \phi_{i+1,j-1} - \phi_{i,j-1}) \\
&\qquad - B^{yx\bullet}_{i,j}(\phi_{i,j+1} + \phi_{i-1,j+1} - \phi_{i,j-1} - \phi_{i-1,j-1})\Big] \\
&- \frac{\beta}{4\Delta x\Delta y}\Big[B^{xy\circ}_{i,j+1}(\phi_{i+1,j+1} + \phi_{i+1,j} - \phi_{i-1,j+1} - \phi_{i-1,j}) \\
&\qquad - B^{xy\circ}_{i,j}(\phi_{i+1,j} + \phi_{i+1,j-1} - \phi_{i-1,j} - \phi_{i-1,j-1})\Big] \\
&- \frac{\beta}{(\Delta y)^2}\left[B^{yy\circ}_{i,j+1}(\phi_{i,j+1} - \phi_{i,j}) - B^{yy\circ}_{i,j}(\phi_{i,j} - \phi_{i,j-1})\right] \\
&+ \frac{c^x_{i,j}}{2\Delta x}(\phi_{i+1,j} - \phi_{i-1,j}) + \frac{c^y_{i,j}}{2\Delta y}(\phi_{i,j+1} - \phi_{i,j-1}) = f_{i,j}.
\end{aligned}$$

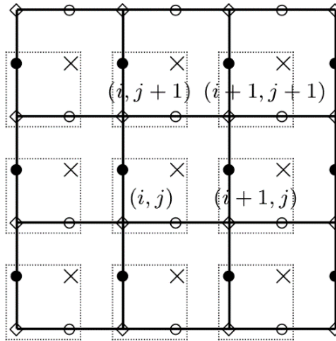

Fig. 12 Indexing for cell-centered discretization in **MLCellABecLap** and **MLCellABecSec**.

To implement **MLCellABecSec**, we modify the implementation of **MLCellABecLap** and **MLABecLaplacian** to handle **B** and **c**. For the smoother, the current implementation of AMReX uses a line red-black Gauss-Seidel algorithm that solves a tri-diagonal system in one direction (one-dimensional). We implemented a pointwise Jacobi and Gauss-

Seidel algorithm including a red-black Gauss-Seidel algorithm for the smoother in **MLCellABecSec**.

2.4.2. Current AMReX limitations

The current implementation of MLMG only supports the l^{∞}-norm for the computation of the residual norm, which makes it hard to estimate the global error for smoothing effects. Therefore, we need to implement the l^2-norm for residuals, which gives a reasonable measure of the convergence behavior for the error reduction of the iterative solvers.

AMReX does not support a polar coordination system visualization, but some visualization programs support the required coordinate transformations. For this purpose, we use the visualization program VisIt as a post-process step.

2.5. *Numerical validation*

To validate our implementation, we use a known solution and the resulting right-hand side on the computational domain $r = [0.5, 4.0]$, $\theta = [0, 2\pi)$, which is shown in Fig. 13, for a real domain and its reference domain. For the EB approach, we need an extended area for the domain, which will contain the EB.

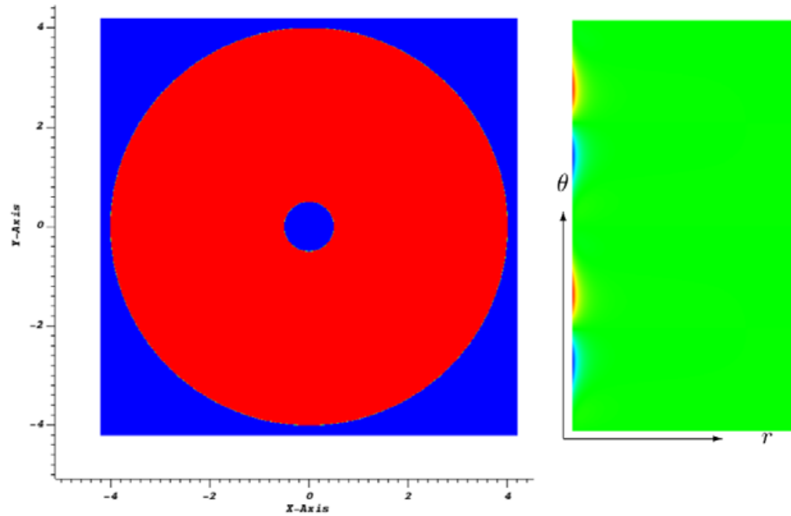

Fig. 13 The computational domain $r = [0.5, 4.0]$, $\theta = [0, 2\pi)$ and its reference domain.

First, we test our implementation with a simple second order PDE, namely the Poisson problem where we use the exact solution as shown in Fig. 14 and its resulting RHS in Fig. 15. For handling the EB approach in AMReX, the values of all vectors (solution and RHS) outside of the domain are set to zero. Therefore, we might not be able to recognize the boundaries of the domain.

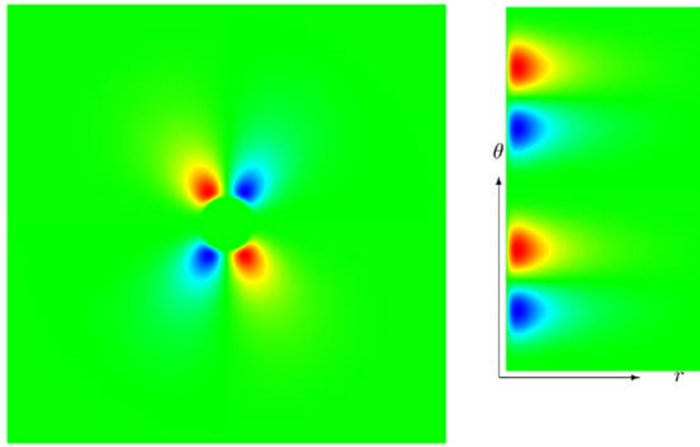

Fig. 14 The exact solution on the computational domain r = [0.5, 4.0], θ = [0, 2π) and its reference domain.

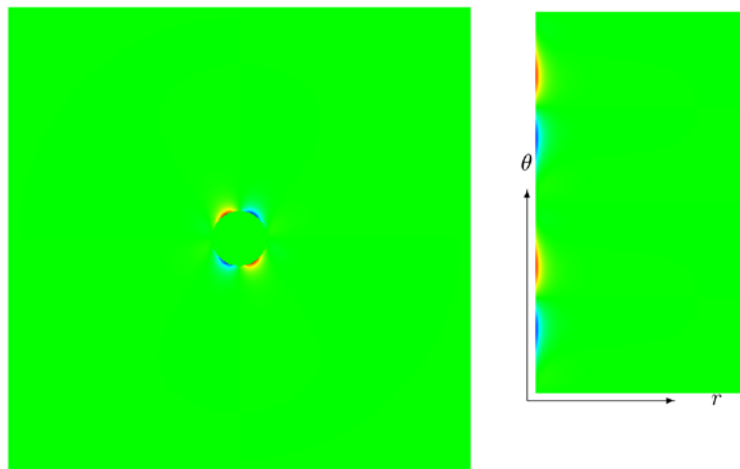

Fig. 15 The RHS on the computational domain r = [0.5, 4.0], θ = [0, 2π) and its reference domain.

We can provide numerical solutions by using two methods, one is using the EB approach (using the **MLEBABecLap** class) and the other is using a polar coordinate transformation (using **MLCellABecSec** class) as shown in Fig. 16. From these numerical results, we conclude that our implementation is valid.

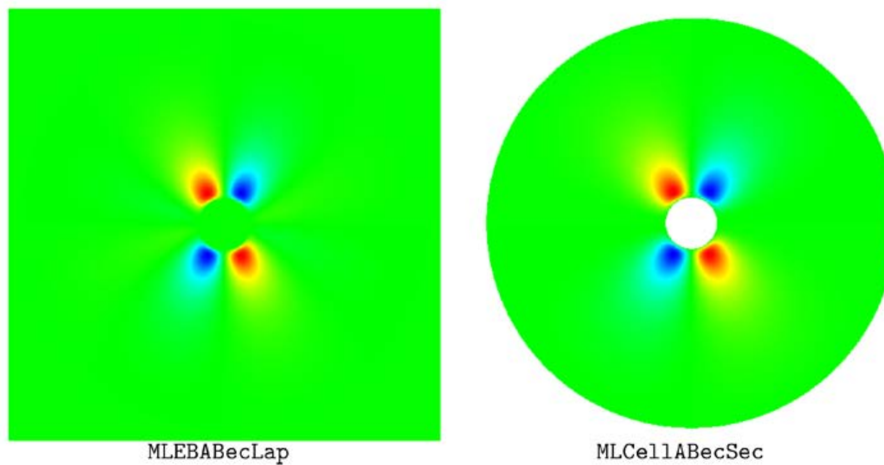

Fig. 16 The solution on the computational domain r = [0.5, 4.0], θ = [0, 2π) using two different methods.

In the following, we investigate the numerical performance of the solvers. The most important numerical property of the multigrid solvers is that the required number of iterations is almost constant. It only slightly depends on the problem size and on the various options of the multigrid method, e.g., the number of coarse levels, the type of the smoothing operator, the number of smoothing steps, etc. However, these small dependencies vanish when a large number of smoothing steps is used. This is especially useful in the debugging process of the multigrid solvers.

We consider the two solvers (**MLEBABecLap** and **MLCellABecSec**), introduced above, both solving the same problem on the computational domain, but having different reference domains. For the **MLEBABecLap** solver we extend the computational domain. Therefore, the number of real cells, which are located inside the reference domain, is only about 70 % of the total number of cells. When we use 1024 cells in each direction, the total number of cells in the computational domain is 1,048,576 and the number of real cells is about 0.735 M. For the **MLCellABecSec** solver with a θ-r rectangular reference domain, we need to use twice as many cells along the θ-direction, compared to the number of cells in the r-direction, to get converged results of the multigrid solver. When we use an adaptive mesh on half of the domain, i.e., 512 cells along the r-direction (1024 cells in the θ-direction), the number of cells is about 0.786 M. In the following figures, we will denote 0.735 M and 0.786 M as 0.8 M. For the next finer level, the total number of cells becomes four times as many, because the number of cells in each direction doubles. We will denote these finer levels as 3 M, 12.5 M, 50 M, 201 M, 805 M, and 3.2 B.

In addition to the options for the multigrid solvers, there are additional options for AMReX. We use the default options of AMReX for both solvers with a fixed number of coarse levels (eight). However, we change the number of pre- and post-smoothing steps. First, we use two pre- and post-smoothing steps for the **MLEBABecLap** solver (denoted by • in Fig. 17). As expected for a multigrid solver the result has only a weak dependency on the problem size. However, when using two pre- and post-smoothing steps for the **MLCellABecSec** solver we get a result, which strongly depends on the problem size (denoted by + in Fig. 17). Therefore, we need to increase the number of pre- and post-smoothing steps for the **MLCellABecSec** solver. We already get reasonable results with three pre- and post-smoothing steps (denoted by ◊ in Fig. 17). Thus, we conclude that the two multigrid solvers behave as expected.

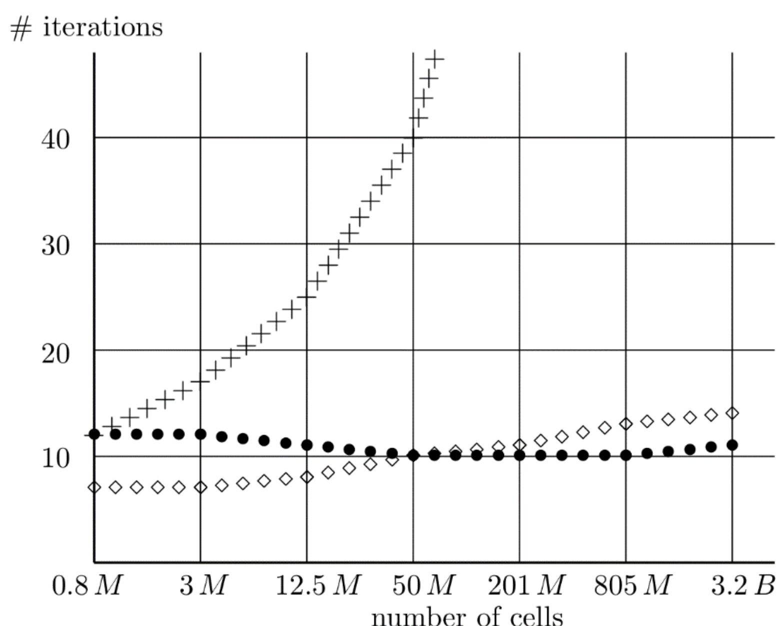

Fig. 17 The required number of iterations to reach convergence of the **MLEBABecLap** solver with two pre- and post-smoothing steps (•) and **MLCellABecSec** solver with two pre- and post-smoothing steps (+) and three pre- and post-smoothing steps (◊).

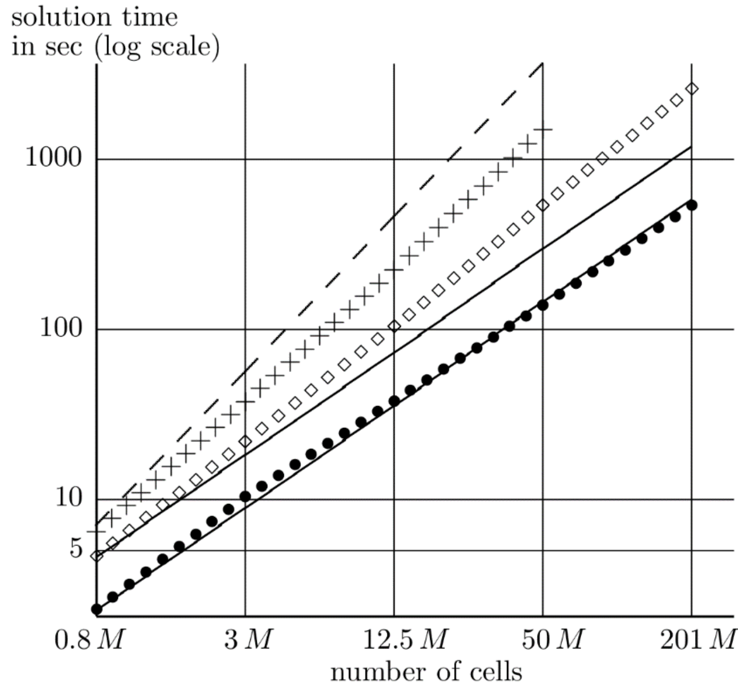

Fig. 18 The solution time in seconds of the **MLEBABecLap** solver with two pre- and post-smoothing steps (•) and **MLCellABecSec** solver with two pre- and post-smoothing steps (+) and three pre- and post-smoothing steps (◊). Solid line denotes the ideal scaling case and --- denotes a Krylov subspace solver.

Next, we consider the solution times on a single core in seconds. We can solve a problem with up to 201 M cells on a single node with a single MPI task (one core). In Fig. 18, we plot three cases for the two solvers, **MLEBABecLap** with two pre- and post-smoothing steps (•) and **MLCellABecSec** with two (+) and three pre- and post-smoothing steps (◊). As a comparison for the scaling behavior, we plot the ideal case as a solid line and a so-called worst case, which is the results of a Krylov subspace solver, denoted by - - -. The results show that the **MLEBABecLap** solver is almost ideal and the case with three pre- and post-smoothing steps still has a good scaling behavior.

Next, we investigate the effect of parallelization on our implementation for solvers with a problem size of 201 M cells. We test an OpenMP implementation on a single MPI task using multiple threads and compare it to a pure MPI parallelization using an equal amount of cores (number of threads for OpenMP is equal the number of cores for MPI tasks). We only use our two best scaling solvers from the previous tests, **MLEBABecLap** with OpenMP (•) and pure MPI (○) and **MLCellABecSec** with OpenMP (◊) and pure MPI (+). For comparison, we plot the ideal case (- - -). For both solvers, the OpenMP parallelization has a good performance behavior. Especially, we have almost perfect numerical parallel performance for the **MLCellABecSec** solver. However, the pure MPI parallelization does not gain anything from the parallelization. These MPI parallelization result show that the current options in AMReX do not scale for the MPI parallelization. Therefore, we need to test the current multigrid solver with various options to get the optimal numerical performance on high performance computers.

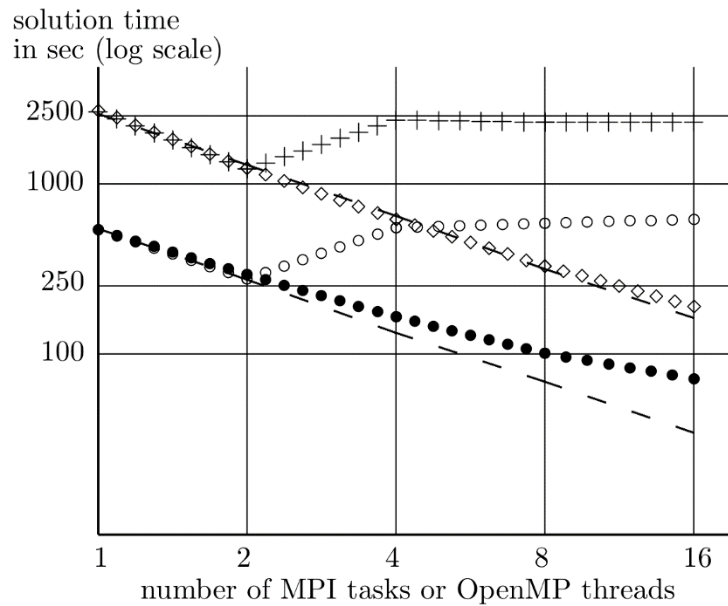

Fig. 19 The solution time in seconds for an MPI and OpenMP parallelization. For the **MLEBABecLap** solver, • denotes with OpenMP and ○ denotes with pure MPI. For the **MLCellABecSec** solver, ◊ denotes with OpenMP and + denotes with pure MPI. --- denotes an ideal scaling case.

2.1. *Conclusions and outlook*

For the MAG project, the principal investigator (PI) planned to implement a multigrid solver with adaptive mesh refinement (AMR) and/or the embedded boundary (EB) method by using publicly available software libraries. We reviewed two algebraic multigrid libraries with AMR, AMReX (from LBNL, USA) and waLBerla (from Universität Erlangen-Nürnberg, Germany), which were suggested by the PI. A decision was taken to use the AMReX library. K. S. Kang installed the AMReX library on a local machine and on Marconi. He tested the installation using the provided examples for linear systems with the adaptive geometric multigrid solver (MLMG).

AMReX provides several multigrid solvers for AMR with EB for Laplacian problems. For the MAG project, we implemented a new linear operator class, which is a derived class of the multigrid solver class of AMReX and handles general second order PDEs. This new class supports a pointwise Jacobi and a red-black Gauss-Seidel smoother and handles a Poisson problem formulated in a polar coordinate system.

We tested two approaches to solve a Poisson problem on a circular domain. One uses the multigrid solver with a polar coordinate system as a reference domain and the other uses an embedded boundary on a uniform mesh. We could achieve correct solutions for both test cases with an acceptable convergence behavior of the iterative solver.

In addition, we could document a reasonable numerical performance for the multigrid solvers on a single core. We also tested parallelization with OpenMP and MPI on a single node. The performance results of the OpenMP parallelization are good. However, the MPI parallelization has to be further improved. To improve the OpenMP/MPI performance, we have to test our implementation with various options, which are available in AMReX in combination with our new multigrid solvers. Finally, we need to implement our solvers into the GYSELA code.

2.2. *Reference*

[1] AMReX Team, amrex Documentation Release 19.02-dev

[2] Nicolas Bouzat, Camilla Bressan, Virginie Grandgirard, Guillaume Latu, Michel Mehrenberger. Targeting realistic geometry in Tokamak code Gysela, ESAIM: Proceedings and Surveys, EDP Sci

3. Final Report on HLST project CINCOMP4

The CINCOMP4 project is a continuation of the CINCOMPX projects [1, 2, 3, 4] and it is dedicated to provide support for European scientists who use the Marconi supercomputer located at CINECA. The Marconi supercomputer was launched in July 2016 and its official production phase started in mid-October 2016. The fusion community has access to two partitions of Marconi named A2 and A3. A2 is based on the latest and final generation [5] of the Intel Xeon Phi product family (*Knights Landing*). The A3 partition is equipped with Intel Xeon 8160 processors (*Skylake*). In the framework of this project, both the hardware and the software of the A2 and A3 partitions were tested using a variety of benchmarks. Multiple issues were found on both partitions and subsequently reported to the Marconi support team via the ticket system.

3.1. *The Marconi supercomputer architecture*

The Marconi supercomputer is located in Bologna at the largest Italian computing centre named CINECA. It currently consists of two partitions named A2 and A3. The A2 partition, which is equipped with processors of the Intel Xeon Phi product family (*Knights Landing*), provides a computational power of about 11 Pflops. The A3 partition is equipped with the latest Intel Xeon 8160 processors (*Skylake*) and provides 8.0 Pflops. One can find a detailed description of the CPU architecture and the partition structure in our previous reports [1, 2, 3, 4] or on the official Marconi user's guide web page [6]. The European fusion community only has access to the so-called Marconi-Fusion part. This part includes 2410 nodes (8.0 Pflops) of the A3 and 449 nodes (1 Pflops) of the A2 partition. The tests in this report will only concern the Marconi-Fusion part of Marconi.

3.2. *Marconi performance stability test*

We continue to execute the Marconi stability test that was developed in the previous project [4]. For this benchmark, three codes (STARWALL, GENE and EUTERPE) are run each month and the run times between the months are compared. The STARWALL code is a pure MPI code which uses the external ScaLAPACK library [7, 8]. The GENE code is a hybrid MPI+OpenMP code that uses the LAPACK, FFTW, SLEPc and HDF5 libraries [9]. The EUTERPE code is an MPI code that uses the PETSc [10] library. The codes were launched on both, the SKL and KNL partitions. In order to increase the statistical significance each run was repeated at least three times and the average, minimum and maximum values were calculated from the results.

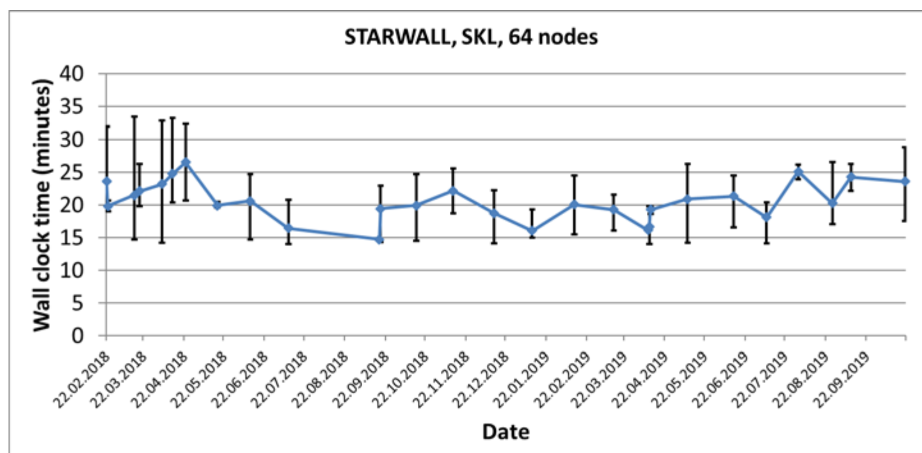

Fig. 20 The execution time in minutes of the STARWALL code, which was executed on the SKL partition using 64 nodes. Minimum and maximum values of at least three runs for each date are indicated by the error bars.

Fig. 20 shows the average wall clock time, together with its minimum and maximum indicated as an error bar, of the STARWALL code, which was run on the SKL partition using 64 nodes. For an ideally stable supercomputer operation the execution time should be identical from run to run, i.e. we would expect a straight line without any error bars. In our case, the wall clock time significantly fluctuates during tests within one day (one campaign) and during the whole measurement (all campaigns). For example, on the 5th of April 2018 we launched three instances of the code. All of them provided different execution times: test1=14.19 min; test2=22.62 min and test3=32.82 min. During the whole examination period, three tests finished after about 14 minutes and five after more than 30 minutes. The reason for these differences is still unclear. One of the explanations could be an ailing node as discussed in detail in [2].

The fluctuations of the execution time are less pronounced when a code is launched on the KNL partition. Fig. 21 shows the wall clock time of the STARWALL code, which was executed on the KNL partition using 64 nodes. After 22.04.2018 we found hardly any fluctuations. A similar behavior was also found for two other codes (GENE and EUTERPE). The fluctuations on the KNL partition could be hidden by the much larger execution time of each code, which is higher by factor of five in comparison to the SKL partition. The empty region in the figure indicates unavailability of the partition on this day (15.02.2018).

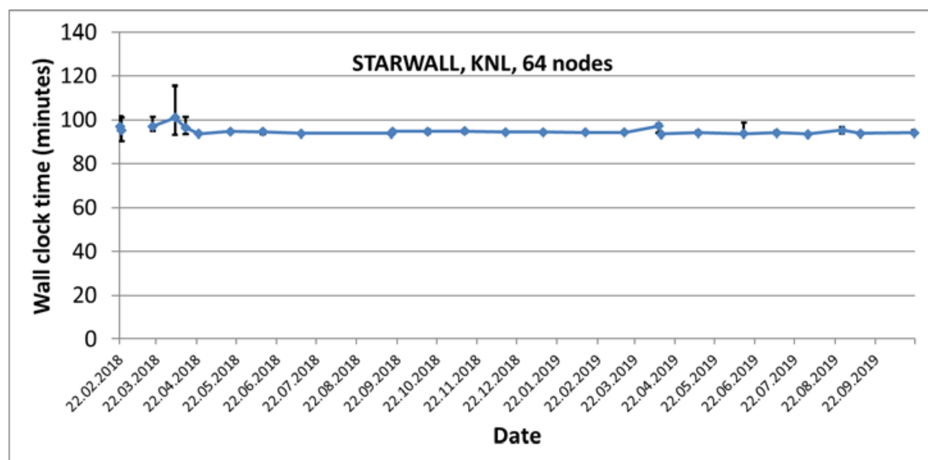

Fig. 21 The execution time in minutes of the STARWALL code, which was executed on the KNL partition using 64 nodes. Minimum and maximum values of at least three runs for each date are indicated by the error bars.

During this benchmark, a significant amount of jobs failed due to various reasons on Marconi. Fig. 22 shows the percentage of failed jobs on both partitions for each campaign (36 jobs were launched in total each time – 18 on KNL and 18 on SKL). As one can see, there were only a few campaigns without any failed jobs. The jobs crashed with a similar probability on both partitions reaching a maximum of 15 (~85 %) on KNL (10.04.2018) and 11 (~60 %) on SKL (09.04.2018).

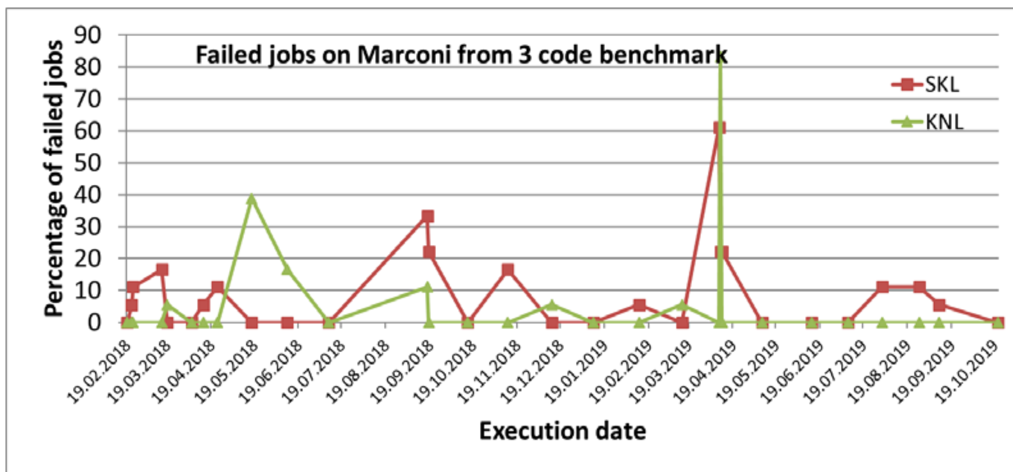

Fig. 22 Percentage of failed jobs on Marconi during the execution of the three code benchmark.

3.3. *Test of the huge virtual pages (HP) queue*

In the previous HLST projects VIRIATO2 [11] and MPI3-DG [12] a bug was found in the Intel MPI library that significantly increased the inter node communication time when shared memory windows were used. This bug disappears with Intel MPI 2019 in conjunction with launching the code on nodes with huge virtual pages (HP) with a size of 2 MB [13].

The problem is that huge pages need to be enabled on the system level, which is not the case for the standard configuration on Marconi, where the default page size is set to 4 kB. Therefore, a special queue with HP was created for a limited time to test if they would have any impact on real production codes.

The three codes (STARWALL, GENE and EUTERPE), which we described above, were tested using both the small pages in the standard queue and the huge pages in the HP queue. In the HP queue we can activate huge pages via an environment variable; without this activation the standard small pages were used in this queue as well.

Three runs were executed for each queue and the results for the EUTERPE code are presented in Fig. 23. No significant difference was found between the runs. The execution time fluctuates in the range of 20 %, which is as expected from the three code benchmark described above. Similar results were obtained for the other two codes. Therefore, no drawbacks were found in terms of wall clock time for using the HP queue.

However, in order to enable HP at the system level one has to choose the amount of memory (RAM) that is dedicated to it. The rest of the memory is devoted to small pages. Thus, we were able to use both the small and the huge pages on the same queue (Fig. 23, red and green lines). In doing so we split the total memory of the node into two parts, reducing the total available memory for the job. This is an important disadvantage of using HP.

Enabling HP on the Marconi system is still under discussion. There are a few options: 1) create a queue which dedicates all memory to HP; 2) create a queue which dedicates part of the memory to HP and the rest to small pages as it was in our test queue; 3) wait until Intel resolves the problem with the shared memory windows.

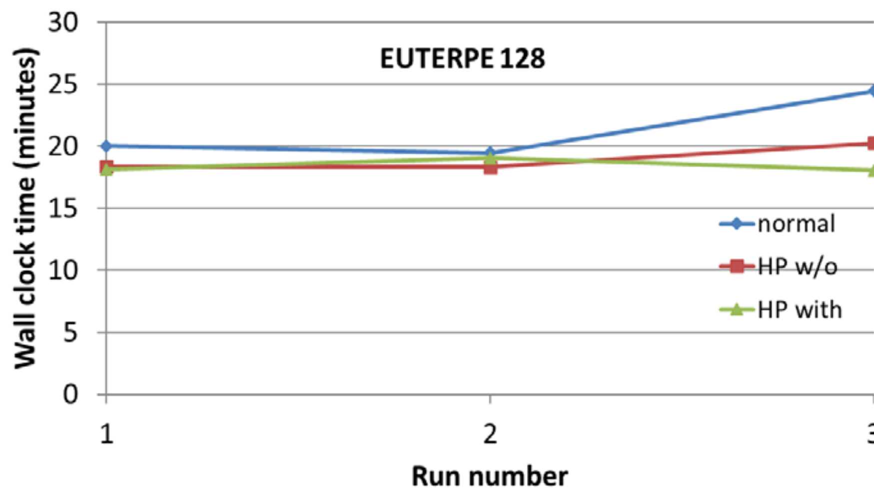

Fig. 23 Wall clock time of the EUTERPE code executed on: (i) the standard queue with small pages (blue line), (ii) the huge pages test queue without HP activation (red line) and (iii) the huge pages test queue with HP activation (green line).

3.4. *Test of the selFIe software package*

SelFIe (SElf and Light proFIling Engine) [14] is a tool to profile Linux software without recompiling. It is a dynamic library, which can be linked by setting the LD_PRELOAD environment variable before running the software. The operation committee of the Marconi supercomputer decided to install this software in order to check the performance of codes in Flops. The software was set up by the Marconi support team and checked by the HLST to see if there is any influence on the code execution time. Two identical runs were performed, one with SelFIe and one without it. The execution time was the same for both runs. Therefore, we can conclude that the software has no impact on the code run time. This is in agreement with the description of the software. However, while the SelFIe output provides plenty of diagnostic metrics it does not provide the performance in Flops. We contacted the developers of the software via the Marconi support team and asked if it would be possible to fix this issue. At the moment, this is still under investigation. Meanwhile, another library, which can also measure the performance of codes in Flops, will be tested on Marconi. This software is developed at the Max Planck Computing and Data Facility in Garching and is called *hpcmd* [22].

3.5. *Test of the new ticket system*

In the beginning of 2019, a new version of the ticket system was introduced on Marconi. The system, called Request Tracker (RT) [15], is used to coordinate tasks and manage requests among a community of users. The system has a simple and user friendly interface that should help users of Marconi to manage their tickets. The RT was tested by the HLST and no drawbacks or crashes were found. Therefore, the software was integrated into the system and opened for production.

3.6. *The JFRS-1 supercomputer*

In 2018 the Japanese National Institutes for Quantum and Radiological Science and Technology (NIRS) deployed the JFRS-1 supercomputer (Japan Fusion Reactor Simulator-1) to support the ITER fusion project. The Cray corporation manufactured the machine. In collaboration between EUROfusion and NIRS we tested this supercomputer and compared our results with Marconi.

3.6.1. **JFRS-1 system specification**

The JFRS-1 supercomputer has 4.2 PFlops peak performance and is equipped with the Intel Xeon Gold 6148 processor. Cray developed its own interconnect called Aries [16], which can provide a bandwidth from 12.5 GBps to 14.5 GBps depending on the

position of a node. A detailed comparison of several supercomputers used for fusion research is given in Table 2.

Table 2. Specifications of several supercomputers use for fusion research.

Name	**Marconi**	**JFRS-1**	**COBRA**	**JUWEL**
Total peak performance (PFlops)	7.4	4.2	9.84	10.4
Processor name	Intel Xeon Platinum 8160	Intel Xeon Gold 6148	Intel Xeon Gold 6148	Intel Xeon Platinum 8168
CPU frequency (GHz)	2.1	2.4	2.4	2.7
Number of cores per node	48	40	40	48
Peak performance per node (GFlops)	3211	3072	3072	4147
Interconnect	OPA	Aries	OPA	Mellanox EDR InfiniBand
Inter node bandwidth GB/s	12.5	12.5 – 14.5	12.5	12.5
Intra node bandwidth (GB/s)	255.94	255.94	255.94	255.94

3.6.2. STREAM benchmark

The STREAM benchmark [17] is one of the most popular benchmarks in the high performance computing community for measuring the memory bandwidth between CPU and RAM. Fig. 24 presents the results obtained with the STREAM benchmark on the Marconi and on the JFRS-1 supercomputer. Both machines show a typical distribution for a NUMA shared memory system [18] with the compact thread pinning method.

A maximum bandwidth of ~191 GB/s can be reached on Marconi. This is ~74.5 % of the theoretical peak memory bandwidth specified by the vendor (255.94 GB/s) [19]. The JFRS-1 supercomputer provides ~194 GB/s maximum bandwidth, which is ~76 % of the theoretical peak memory bandwidth (255.94 GB/s) [20]. These results are a typical fraction of the peak performance for such HPC systems.

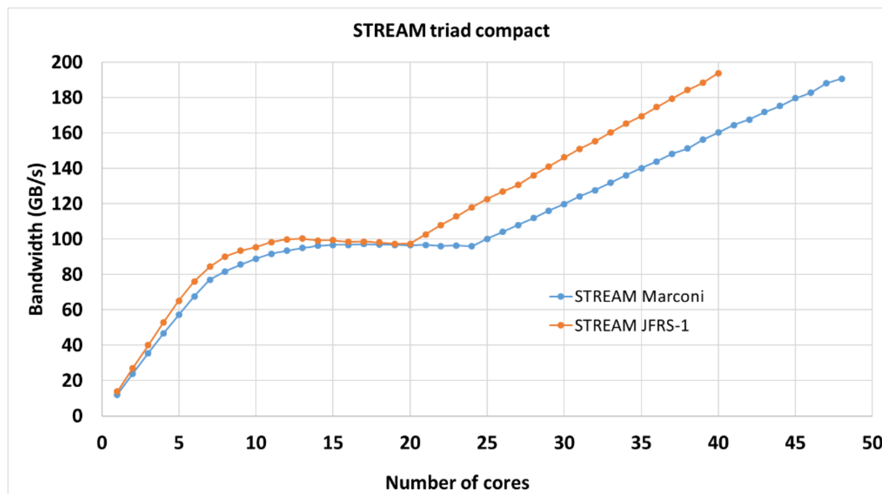

Fig. 24 Results of the STREAM benchmark on a Marconi and JFRS-1 compute node using the Intel 2018 compiler and the *-xCORE-AVX512* flag. Full compilation command line: *icc -xCORE-AVX512 -O3 –qopenmp -mcmodel=medium -DSTREAM_ARRAY_SIZE=400000000 -DVERBOSE -DNTIMES=50 stream.c -o stream_skl.x*

3.6.3. Intel MPI Benchmark (IMB)

The PingPong test from the IMB suite [21] was used to test the Marconi and JFRS-1 network performance. The obtained results are shown in Fig. 25. Both supercomputers have a similar behavior for small message sizes <0.01 MiB. For large message sizes (> 1MiB), Marconi has a higher bandwidth for the inter-node test. However, for the intra-node test JFRS-1 is superior. We have to point out that this benchmark was run using the standard configuration of the supercomputers. The results can vary when changing the limit of the EAGER and/or RENDEZVOUS protocols or by using different providers for the communication. Also we can increase the number of receive buffers at runtime, which can improve the bandwidth for particular message sizes.

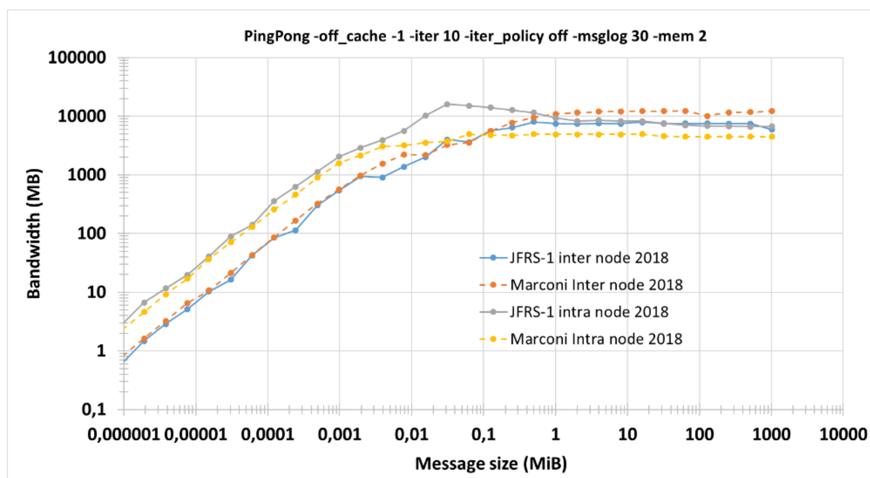

Fig. 25 Comparison of the memory bandwidth for the inter- (solid line) and intra-node (dashed line) PingPong test on the Marconi and on the JFRS-1 supercomputer.

3.6.4. Linpack SMP benchmark

The Intel SMP Linpack benchmark was run in order to measure the peak performance of a compute node of each supercomputer. On Marconi 1941.91 GFlops were achieved. This is ~60 % of the theoretical peak performance (3211 GFlops). On JFRS-1, we obtained 1859.46 GFlops. This corresponds to ~60 % of the theoretical peak performance (3072 GFlops). Therefore, an identical performance in terms of the percentage of the theoretical peak was measured on both supercomputers.

3.6.5. **Performance stability test**

For the last test, we checked the performance stability of JFRS-1 using the STARWALL code from the three code benchmark described above and compared the obtained results with other supercomputers. 10–15 identical runs of the STARWALL code using 16 nodes were executed on Marconi, COBRA, JUWELS and JFRS-1 (see Table 2). The results are shown in Fig. 26. The wall clock time fluctuates on COBRA, JUWELS and Marconi. However, the fluctuation amplitude on JUWELS (~20 %) is lower in comparison to Marconi (35 %) and COBRA (38 %). The best results were obtained on JFRS-1. The wall clock time fluctuations are in the range of 2 %. The code performance on JUWELS is the fastest with a base line of about 13 minutes. We expected these results as the JUWELS system has the highest peak performance per node (Table 2). Nevertheless, having identical compute nodes on COBRA and JFRS-1, the wall clock time on JFRS-1 is much smaller in comparison to COBRA. Therefore, in contrast to the other tested supercomputers JFRS-1 shows a stable and fast performance for a real code.

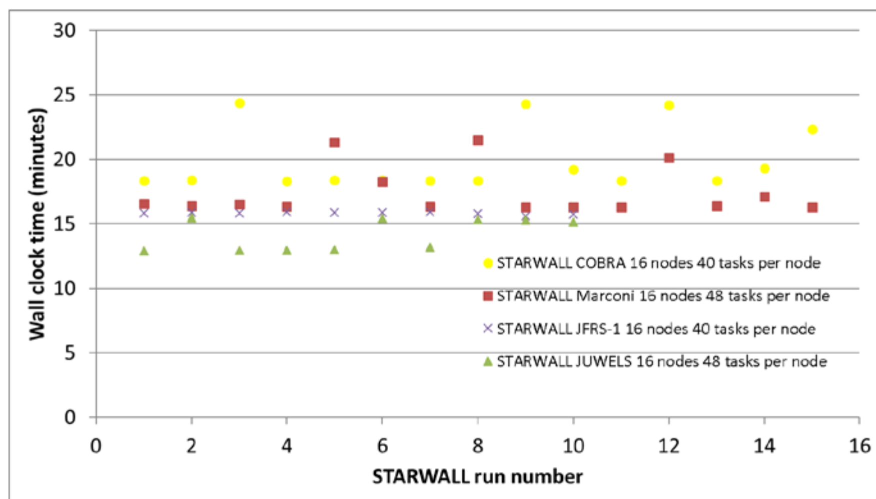

Fig. 26 The execution time in minutes of the STARWALL code which was run on the SKL partition of Marconi (red squares), COBRA (yellow circles), JUWELS (green triangles) and JFRS-1 (violet cross) using 16 nodes.

3.7. *Marconi job waiting time in the queueing system*

In order to measure an average job waiting time in the Marconi queueing system we developed the following benchmark. A cron job queues an empty job every day at a particular time and the waiting time before the job is started is recorded. We want to produce the maximum waiting time scenario. Thus, a 64 nodes job with a requested time of 24 hours is chosen. The job is launched at 11.00 a.m., which is typically a busy time for a supercomputer. The results are presented in Fig. 27. During five months of testing only four jobs waited longer than ten hours with a maximum waiting time of ~35 hours. The average waiting time was 1.47 hours.

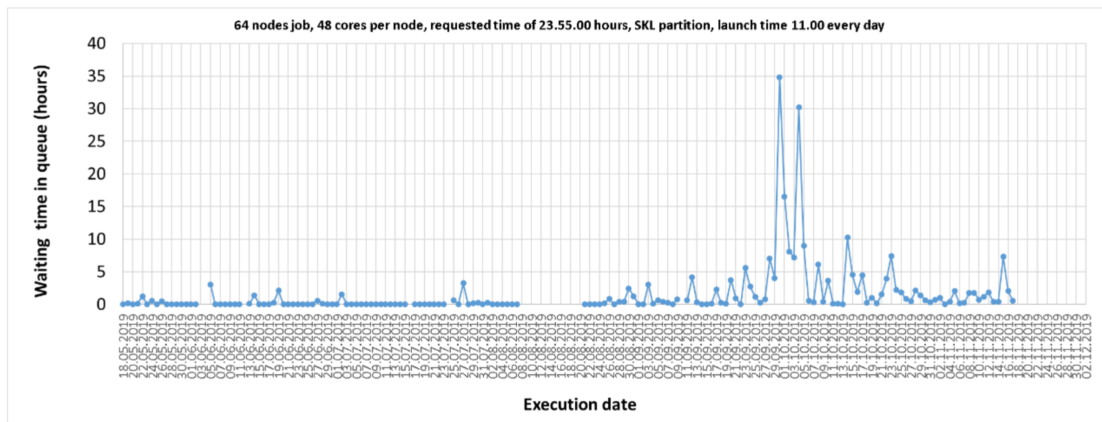

Fig. 27 The waiting time in the Marconi queueing system for a 64 nodes job with a requested time of 24 hours.

3.8. *Beta testing of the Intel oneAPI model*

Intel is developing the oneAPI model to simplify programming for any hardware developed by Intel. The model includes modern C++ features and provides parallelism using a newly developed programing language called Data Parallel C++ (DPC++). This language allows the execution of the same binary on different hardware such as CPU, GPU and FPGA.

The HLST participated in the closed beta testing of the Intel oneAPI model and the DPC++ programming language. All tests were done on the Intel DevCloud cluster [23] where the DPC++ compiler is installed. The tests were dedicated to linear algebra operations such as the sum of two vectors. All runs were successful without any crashes. However, the DPC++ programming language is quite complex and requires the knowledge of C++11. Moreover, taking into account that most of the fusion codes are written in FORTRAN, conversion to oneAPI will not be straightforward and will require a lot of programming effort.

3.9. *Conclusions*

Different benchmarks and tests were performed in order to determine the performance of the A2 (KNL) and A3 (Skylake) partitions of Marconi. Issues were found that significantly limit their use. Some of them were resolved by the Marconi support team, others, however, are still under investigation.

The so-called “three code benchmark” was run regularly in order to check the stability of Marconi in terms of the execution time for real production codes. It was found that the wall clock time of codes can fluctuate significantly from one run to the next. Fluctuations in the execution time of more than 20 % were detected. It was found that the fluctuation problem is related to the MPI communication rather than to the Intel Omni-Path interconnect.

During the “three code benchmark” a significant amount of jobs failed due to different reasons on Marconi. We investigated all these issues.

Huge virtual pages (HP) were tested on Marconi. We found that they have no influence on code performance in terms of the wall clock time. However, HP support has to be enabled on the system level and the working memory has to be divided into two parts, one for small and one for huge pages. This is a significant limitation of the usage of HP.

The SelFIe software for code profiling was installed and tested on Marconi. The software works correctly without any additional overhead. However, the most important metric for us (the performance in Flops) was absent in the output. The developers of the SelFIe software are currently fixing this problem.

A new ticket system called Request Tracker was installed on Marconi. The software works flawlessly and has a convenient and user-friendly interface.

The JFRS-1 supercomputer was tested using different benchmarks. The machine works stably, incurring only small fluctuations of the wall clock time during production code computation.

The average waiting time in the Marconi queueing system was measured to be 1.47 hours for a 64 nodes job with a requested execution time of 24 hours.

3.10. ***References***

[1] S. Mochalskyy, HLST annual report 2016
[2] S. Mochalskyy, HLST annual report 2017
[3] N. Moschuering, HLST annual report 2017
[4] S. Mochalskyy, HLST annual report 2018
[5] https://itpeernetwork.intel.com/unleashing-high-performance-computing/
[6] https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide
[7] Merkel P. and Sempf M. 2006 Proc. 21st IAEA Fusion Energy Conf. (Chengdu, China) TH/P3-8
[8] Merkel P., Strumberger E., Linear MHD stability studies with the STARWALL code arXiv:150804911 (2015)
[9] http://genecode.org/
[10] http://fusionwiki.ciemat.es/wiki/EUTERPE
[11] T. Ribeiro, HLST annual report 2017
[12] T. Ribeiro, HLST annual report 2018
[13]https://software.intel.com/sites/default/files/managed/eb/54/An_Introduction_to_MPI-3.pdf
[14] https://github.com/cea-hpc/selFIe
[15] https://bestpractical.com/?rt=4.4.3
[16] https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf
[17] https://www.cs.virginia.edu/stream/
[18] https://en.wikipedia.org/wiki/Non-uniform_memory_access
[19] https://ark.intel.com/content/www/us/en/ark/products/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz.html
[20] Super Computer System Cray XC50, Users Guide, Ver-1.06, Cray Japan Inc. 7/18/2019
[21] https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[22] https://www.mpcdf.mpg.de/services/computing/performance-monitoring
[23] https://software.intel.com/en-us/devcloud/oneapi

4. Final Report on HLST project SPICE2

4.1. *The SPICE codes*

The SPICE (Sheath Particle In Cell) package includes two codes: SPICE2 (2D3V) and SPICE3 (3D3V) [1, 2]. These codes can perform simulations of magnetized plasmas in contact with solid objects and have been successfully used for the study of plasma deposition in the vicinity of castellated plasma-facing components (PFCs) [3, 4]. In order to resolve the virtual cathode (formed at the surface due to the thermionic emission), the size of the PIC cells has to be relatively small (one tenth of the Debye length). Preliminary 3D simulations with a cell size of one quarter of the Debye length have shown important modifications of the escaping thermionic current arising from a localized hotspot. In order to perform complete 3D simulations without any simplifications, substantial improvements in the scaling and parallelization of the code have to be done.

4.2. *Status of the code*

SPICE3 is written in Fortran 90 and provides its output in the Matlab MAT binary format. The code is parallelized using domain decomposition. In the previous HLST project SPICE [5] a 3D PETSc [6] parallel Poisson solver was developed. The solver was validated using synthetic and real data. The solver has been integrated into the SPICE code and production runs were performed. However, it was found that the original force calculation subroutine requires a full (not distributed) potential matrix. This produces a sequential overhead and requires a significant amount of memory. The current project is dedicated to parallelizing this subroutine and to modifying the structure of the distributed potential sub-matrices in order to use them in a parallel calculation of the force term.

4.3. *Distributed calculation of the E-field*

The calculate of the electric field is calculated for each PIC cell by taking the gradient of the potential ($\boldsymbol{E} = -\nabla\phi$). In the SPICE code, the first order central-difference scheme is used to discretize this equation:

$$E_x = -\frac{\phi_{i+1,j,k} - \phi_{i-1,j,k}}{\Delta x}, E_y = -\frac{\phi_{i,j+1,k} - \phi_{i-1,j,k}}{\Delta y}, E_z = -\frac{\phi_{i,j,k+1} - \phi_{i-1,j,k}}{\Delta z}.$$

A problem in the parallelization procedure arises at the boundary points of each sub-domain. The electric field at the boundaries needs the potential value of the first and the last plane of the neighboring processes. For example, process number two has the following boundaries in the *x*-direction: x_{start}=10, x_{end}=20. In order to calculate the electric field at *x*=10 ($E_{i=10,j,k} = -\frac{\phi_{11,j,k} - \phi_{9,j,k}}{\Delta x}$) the potential at *x*=9 is required. This potential is located in the memory of the neighboring process number one. The same situation exists at the right boundary, where in order to calculate the electric field at *x*=20 the potential at *x*=21 is required, which is located in the memory of process number three.

This problem has been resolved by developing a special subroutine that performs the exchange of the boundary planes between neighboring processes in all three dimensions (**Fig. 28**). The *MPI_SENDRECV* subroutine was used to perform the communication between the processes. In order to decrease the communication time all values at the boundaries were grouped in a vector (*MPI_TYPE_VECTOR*) before communication started.

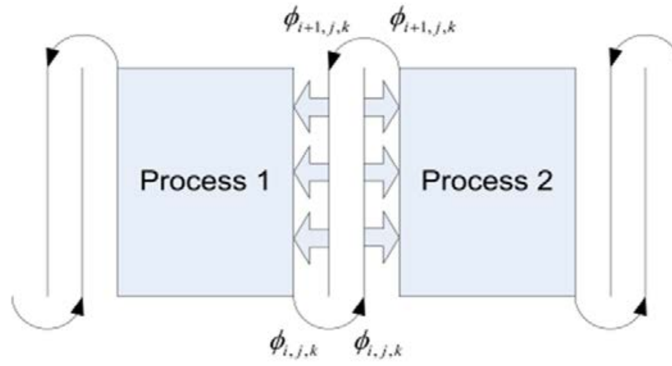

Fig. 28 Schematic view of the boundary potential data exchange between parallel tasks.

Finally, all processes obtain the necessary boundary potential of their neighbors for the parallel calculation. Having distributed the required data, the parallelization of the force calculation subroutine is prepared.

For the next step, we test the complete parallel code and compare the results with the old version of the code. Fig. 29 shows the obtained potential distribution as a function of the *x*-direction for both the SPICE3 sequential calculation of the electric field (orange line) and for the new parallel version (blue line). The obtained results from both solvers are on top of each other. Therefore, the results confirm the correctness of our parallel code version.

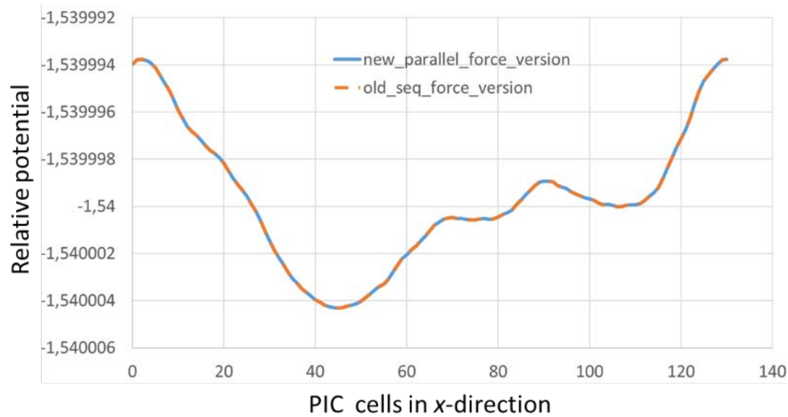

Fig. 29 Potential distribution in the *x*-direction in the *y*=15, *z*=32 plane obtained from the full parallel SPICE code version (blue line) and from a version of the code with sequential calculation of the electric field (orange line). The simulation domain was distributed among 64 MPI tasks in the *x*- and *y*-direction.

The new parallel force term does not need the global potential matrix any more. The matrix covers the dimensions along the grid cells in three spatial directions (*Poten_global(Nx, Ny, Nz)*). For a large production run, the mesh can reach a size of up to 512 entries in all dimensions. Therefore, one GB per MPI task is saved by our improvement. This is quite a significant amount, if we consider that one node of the Marconi supercomputer, consisting of 48 cores, has a maximum available memory of 180 GB. Thus, one-fourth of the total node memory is freed up. Moreover, the execution time of the complete SPICE code decreases by 7.59 % when using the parallel subroutine.

4.4. *Performance of the complete SPICE code*

The project coordinator measured the execution time of a complete PIC cycle of the SPICE code with both the old sequential version of the solver and the electric field and the new parallel version of the code. The results are presented in **Fig. 30**. Using 64 MPI tasks with the old serial multigrid solver the total execution time per iteration was about

0.9 s. With the new parallel version, the execution time decreased by a factor of two to 0.48 s.

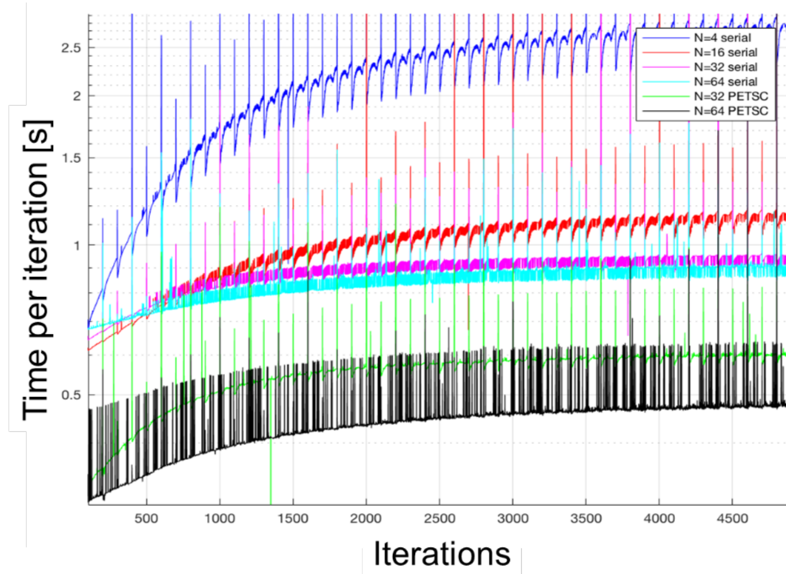

Fig. 30 Wall clock time per complete PIC iteration.

The scalability of each independent subroutine was also tested using a moderate grid size (256x256x256). Fig. 31 shows the execution time of different subroutines as a function of the number of cores. The new parallel Poisson solver scales up to the maximum tested number of 512 cores (pink line). When using a higher number of cores, the particle transfer subroutine (green line) becomes a bottleneck. In order to improve the global code scalability further this subroutine has to be optimized.

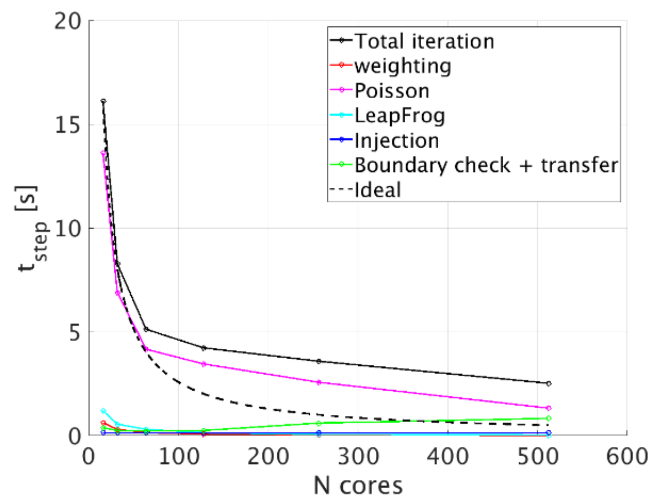

Fig. 31 Execution time of different subroutines versus number of cores.

4.5. *Bug check*

Finally, we checked the complete parallel version of the code for correctness. Run time debugging was performed with the Intel compiler. Afterwards the source code was also checked with the Forcheck static analyzer [7]. One uninitialized variable was found that could produce unexpected behavior, but it was only used in the debugging version of the code. We also identified a wrong sequence of arithmetical operators. Two "+" operators were used one after another.

More than 200 unused variables and more than 30 unused dummy arguments were detected. A few mismatches with the Fortran standard were also detected that had no

influence on the correctness of the code. A detailed report containing all these issues was compiled and sent to the project coordinator.

4.6. *Conclusions*

During the SPICE2 project, the electric field calculation subroutine was parallelized. A parallel plane exchange subroutine was developed in order to calculate the electric field for the boundary cells. The subroutines were validated using a variety of tests with synthetic data. The solver was also tested with real data from the SPICE code providing identical results compared to the serial version of the force calculation term. An important gain in terms of memory consumption was achieved together with an overall code speed up of 7.59 %. Poisson solver scalability was shown up to 512 cores. The complete code was checked for correctness using the Forcheck static code analyzer. No errors, which could have had an influence on the results were found.

4.7. *References*

[1] M. Komm, PhD thesis, “Studium okrajového plazmatu Tokamaku a jeho interakce s první stěnou”, Praha 2011

[2] Z. Pekarek, PhD thesis, “Advanced techniques of computer modelling in low- and high-temperature plasma physics”, Prague 2012

[3] M. Komm et al., Nucl. Fusion 57 (2017)126047

[4] J. Coenen et al., Nucl. Fusion 55 (2015) 23010

[5] S. Mochalskyy, HLST annual report 2018

[6] https://www.mcs.anl.gov/petsc/

[7] http://www.codework-solutions.com/development-tools/forcheck-fortran-analysis/

5. Final Report on HLST project PICOPT2

5.1. *Introduction*

The HLST project PICOPT2 aims at helping developers of gyro-kinetic particle-in-cell (GK PIC) codes to leverage the performance benefits of modern supercomputer architectures. To this end, it envisions to provide optimized implementation strategies for common GK PIC algorithms. GK PIC codes have very special capabilities and demands. Recent computer architectures (KNL, SKL) bring special constraints to optimization efforts as well, with vectorization being the major one.

PICOPT was proposed with the hope that the narrow algorithmic definition of GK PIC codes can be exploited in order to provide optimizations for many current simulation codes. Since refactoring and optimization are large-scale code changes, which demand very careful examination of the code followed by implementation and intensive testing stages, GK PIC codes are especially suitable and worthwhile for this task as they are a code class, which consumes proportionally high amounts of computing resources.

The second installment of this project, called PICOPT2, focuses on the GK PIC code EUTERPE. This code is widely used in the community, for example for stellarator simulations. The aim of this project is to improve the performance of EUTERPE via the exploitation of the vectorization capabilities of the Intel Skylake architecture.

For details about the algorithm of the EUTERPE code, please consult the documentation of the PICOPT project [1].

5.2. *Theoretical peak AVX512 performance*

In order to understand the potential performance gains of the new Intel Skylake SP (SKL) processors Platinum 8160 (used for Marconi A3) and Gold 6148F (used for Cobra at the MPCDF), we will present a short comparison with the Intel Broadwell E5-2697 v4 (BDW) processor. We chose this processor for this comparison, because it was previously available on Marconi and is the predecessor, which fits the closest in terms of general behavior. Intel Knights Landing (KNL) processors have a very different architecture and do therefore not lend themselves as easily to this comparison.

Xeon	L1 total	L1 / core	L2 total	L2 / core	L3 total	L3 / core
	private 50 % I – 50 % D		private		shared, write-back	
E5-2697 v4	1.125MiB	2 * 32KiB	4.5MiB	256KiB	45MiB	2.5MiB
			8-way set associative		inclusive (contains L2) 20-way set associative	
Platinum 8160	1.5MiB	2 * 32KiB	24MiB	1MiB	33MiB	1.375MiB
			primary 16-way set associative quadruple TLB		non-inclusive (victim) 11-way set associative	
Gold 6148F	1.25MiB	2 * 32KiB	20MiB	1MiB	27.5MiB	1.375MiB
			equal to 8160		equal to 8160	

Table 3 Cache configuration comparison between a BDW and two SKL processors. The given values are for one processor, not for one node, which commonly consists of two processors.

You can find a comparison between the different cache configurations of these three processors in Table 3. As you can see, the cache configuration is vastly different. It is very significant that the L2 cache has grown a lot and is four times larger per core in the newer architecture. This should be kept in mind when designing performance critical loops. Data reuse inside a core should prove to be simpler to achieve and more effective for SKL. The L2 cache is also much more efficient, sporting a 16-way associativity (up from 8-way) and a quadrupled translation lookaside buffer (TLB). All these changes make the L2 cache much more powerful, even though its latency increased from 11 to 13 cycles due to the increased size. In order to achieve these improvements, the L3 cache was made much weaker. It is smaller and less efficient. It loses associativity, and has been degraded to a victim cache. The reason for this degradation is not the common reason for victim caches, which is to support a direct mapped cache with a fully associative one. In this case, both caches are set associative. The reason in this case is that with the L2 cache size of 1MiB and the L3 cache size of 1.375 MiB, it would not make sense to have the L3 cache be inclusive. Being non-inclusive means that L2 cache lines may only found in the L3 cache if they have been previously evicted and subsequently reloaded from/to L2. An exclusive cache would guarantee that L3 never holds any cache lines that are resident in L2. Cache lines are not transferred from memory to the L3 cache directly; they are always loaded to L2. This makes memory access (snooping) in L3 less efficient. More importantly, this makes using the L3 cache efficiently relatively hard to do. Most workflows will have local variable data not exceeding 1 MiB and will therefore be solely residing in the L2 cache. Only if the working data set size of an algorithm specifically spills out of L2, but still fits into the L3 cache, will we be able to gain an advantage. Therefore, the L3 cache will only help us if our working data set size is between 1 MiB and about 2.375 MiB per core. For data set sizes that are bigger than 2.375 MiB per core, the L3 efficiency will go down quite fast. The BDW cache configuration made use of the L3 cache for working data set sizes of 256 KiB to 2.5 MiB. Since many workflows reuse more than 256 KiB, the L3 cache is much more important there.

In summary, since the private (to each core) L2 cache is more powerful and the shared (over cores on a socket) L3 cache is less powerful, it is even more important to aim at good data reuse and data locality in production codes.

Xeon	Type	Controllers	Channels	Max BW / CPU	Max BW / node
E5-2697 v4	DDR4-2400 (MHz)	1	4	71.53GiB/s	143.06GiB/s
Platinum 8160 Gold 6148F	DDR4-2666 (MHz)	2	6	119.21GiB/s	238.36GiB/s

Table 4 Memory controller configuration comparison between BDW and SKL. The maximum bandwidth (BW) is given by $b_{\max} = f \cdot c \cdot 64/8$, where f is the frequency and c is the number of channels.

Another very performance critical part is the memory subsystem. The respective parameters are given in Table 4. The bandwidth has increased by roughly 70 %, thanks to a slightly increased clock speed and additional memory controllers. This number will be very important when evaluating the performance of EUTERPE in later parts of this report.

Table 5 is, in parts, taken from [4], but we adapted the numbers to show the correct amount of maximum achievable floating-point operations per second (FLOPS). These numbers are critical to understanding the performance of code running on an SKL machine. One thing to note, regarding the performance of the SKL partition, is the emergence of the Intel Turbo Boost (ITB) technology. ITB allows increased CPU frequencies, depending on the used instructions (and the amount of active cores). An overview of this behavior is shown in Table 5.

Instructions	Frequency		TFLOPS / node		
	Base	ITB	Base	Base + FMA	ITB + FMA
8160 Normal	**2.1GHz**	2.8GHz	**0.20**	0.40	0.54
8160 AVX2	**1.8GHz**	2.5GHz	**0.69**	1.38	1.92
8160 AVX512	**1.4GHz**	2.0GHz	**1.08**	2.15	3.07
6148F Normal	**2.4GHz**	3.1GHz	**0.19**	0.38	0.50
6148F AVX2	**1.9GHz**	2.6GHz	**0.61**	1.22	1.66
6148F AVX512	**1.6GHz**	2.2GHz	**1.02**	2.05	2.82

Table 5 Frequencies and FLOPS of SKL processors when all cores are active. ITB stands for the Intel Turbo Boost technology and FMA refers to fused-multiply-add instructions. The numbers in bold face are the most relevant numbers for our purposes.

The amount of FLOPS is calculated according to $\mathrm{FLOPS} = f \cdot V \cdot 48$ where f is the frequency, the factor 48 signifies the number of cores per node and the factor V stems from

$$V = b \Big/ \big(\underbrace{8}_{\text{per byte}} * \underbrace{8}_{\text{per double}} \big) * \underbrace{2}_{\text{VPUs}} * \underbrace{2}_{\text{FMAs}} ,$$

where b is two for normal operation, 256 for AVX2 operation and 512 for AVX512 operation, respectively. The final additional factor of four comes from the amount of vector processing units (VPUs) and the amount of floating operations per fused-multiply-add (FMA) instruction (this means that this factor of two is not considered for the TFLOPS/node – Base column in Table 5).

Unfortunately, ITB is not enabled on both the Marconi A3 and the MPCDF Cobra system. ITB would increase the theoretical peak performance to 3.07 TFLOPS and 2.82 TFLOPS, respectively, but would require more energy and increased cooling. Since there are not many codes, which make use of the additional performance, since it can only be realized through heavy use of vector instructions, it is sensible to keep ITB switched off.

Because of this, from all the numbers given in Table 5, mainly the ones in column TFLOPS/node – Base, printed in bold face, are of interest. Codes will rarely be able to exploit FMA instructions consistently, which makes the additional factor of two rather fictional.

5.3. *Performance analysis of EUTERPE*

In order to perform a targeted and efficient optimization of EUTERPE we will first need to understand and analyze the characteristics of it. This project focuses on the particle-pushing algorithm in order to improve performance. The reasons for this are that a) particle pushing takes up to 30 % of the runtime for most scientific applications and b) particle pushing has a high likelihood of being compute bound even on modern processors. We will revisit these two points when investigating the performance data.

Since EUTERPE offers a host of different configurations and regimes, we obtained two test cases for this project from Axel Koenies and Ralf Kleiber.

The first case uses a field grid size of 64 x 64 x 64 and three different kinetic species consisting of 0.8 million ions, 6.4 million electrons and 0.8 million fast ions. The original test case planned for 40 times as many particles but we reduced this number in order to increase the turn-around on our benchmarks. This reduction should not meaningfully change the performance results. The benchmark uses the complex version of the code. This benchmark represents a regime in which the particle pusher dominates the simulation. We call this test case the linear test case.

The second test case uses a field grid size of 64 x 512 x 512, no electromagnetic simulations and only a single ion particle species with 100 million particles. We call this test case the turbulence test case. We will focus on the first test case in the upcoming discussion. The reason for this is that the second test case does not perform too different from the first one. Particle pushing still dominates the simulation and the memory pattern does not change meaningfully, as we will show later.

EUTERPE might enter a different regime of operation when using even larger fields and more computing cores. In that case, the field solver might overtake the particles in resource consumption, as its scaling properties are inherently worse. These regimes are very hard to replicate in a good way. Furthermore, the field solver is using a third party library, Petsc, the optimization of which is not part of this project.

We analyzed the performance of the particle-push loop with respect to multiple important metrics, using the likwid toolset [5]. We perform measurements using the following metrics: (a) the time the loop takes to complete, measured with the RDTSC (read timestamp counter) instruction, (b) the amount of floating-point operations (FLOPS), separately for different vector register sizes, (c) the amount of data read and written and (d) the number of cache hits and misses. In order to focus solely on the particle-push loop EUTERPE was interfaced using the likwid marker API. This API enables tight control on the specific parts of the code, which are considered for the metrics. We used different markers for the different particle species, but merged the ion and fast ion results as these species behave the same way in the particle-pushing algorithm. All results in this chapter are per particle values, where applicable.

We were able to split the particle loop into three parts. The subpart "fields" collects all operations inside the loop, which are averaging the electromagnetic field values, the subpart "equil" collects operations pertaining to calculating the equilibrium field values for particle pushing and "push" incorporates all other operations inside the loop. This splitting was not done in the straightforward way of using more marker regions. This approach was tried first and proved to be unreasonably expensive. Since these markers are so expensive it could also easily lead to major variations in the measured performance, which would make the obtained results highly questionable. The splitting was therefore achieved by making the two segments "fields" and "equil" optional during compilation. The code was then run three times: with "fields" switched off → V_{nofields}, with "equil" switched off → V_{noequil} and with both "fields" and "equil" switched off → V_{noboth}. Using the performance counter results for these three different runs it is possible to compute discrete values for the three regions:

$$V_{\text{push}} = V_{\text{noboth}}$$

$$V_{\text{fields}} = V_{\text{noequil}} - V_{\text{noboth}}$$

$$V_{\text{equil}} = V_{\text{nofields}} - V_{\text{noboth}}$$

V can be any one of t, $FLOPS$, $data_{\text{read}}$, $data_{\text{written}}$, $n_{\text{cache miss}}$, $n_{\text{cache hit}}$, and they are collected for each rank, summed up over ranks and divided by the total amount of performed particle pushes. From these component specific values, we can then calculate component specific bandwidths or arithmetic intensities. FLOPS are calculated using $FLOPS = \sum_{\text{ranks}}(FLOPS_{\text{rank}}/t_{\text{rank}})$.

There are some drawbacks to this approach: The data comes from different benchmarks and the code may perform differently if some code segments are switched off (especially the caching behavior will be different). The large size of the different regions should limit the negative influence on our results.

All benchmarks were performed on the MPCDF Cobra supercomputer, using 64 ranks on two compute nodes. The ranks were distributed evenly over all sockets in order to benefit from the maximum available bandwidth. We enabled data alignment to 64-byte boundaries, full vectorization and interprocess optimization. Setting `--zmm-usage=high` increased the performance and is therefore always on.

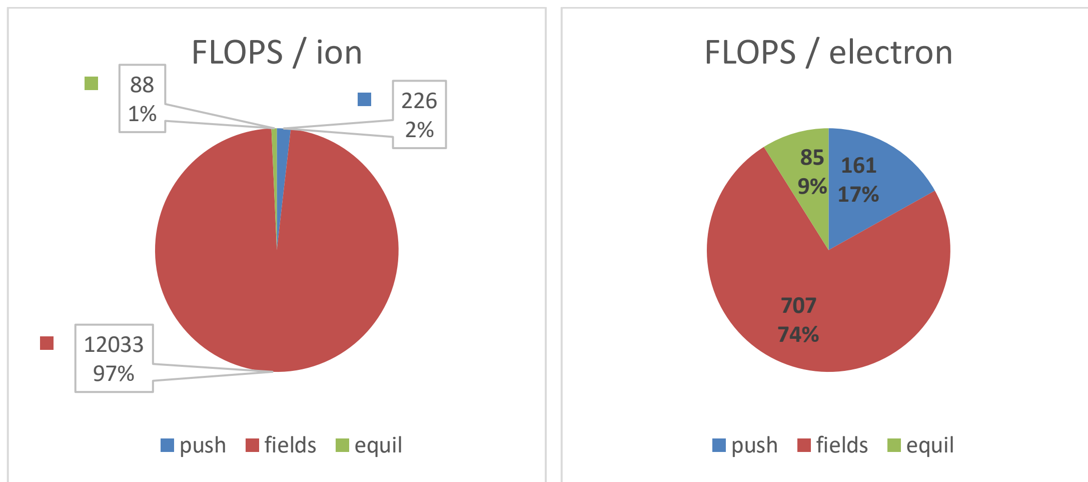

Fig. 32 Amount of FLOPS per particle performed for different subparts of the particle loop, for ions on the left side and electrons on the right side. The given values are the amount of double precision FLOPS per particle. The FLOPS for this measurement have been processed in any kind of vector register. The upper number is the amount of FLOPS; the lower number is the percentage in terms of the total amount of FLOPS.

Fig. 32 shows a detailed breakdown of the amount of FLOPS needed to push one ion or electron as well as specifically which part of the loop consumes these FLOPS. From this plot, we can immediately derive one important conclusion. The particle pusher is not mainly pushing particles. The predominant amount of work is done when averaging the electromagnetic fields. The function could therefore more fittingly be named "field averager" instead of "particle pusher". Specifically, the EUTERPE code uses 97 % and 74 % of the FLOPS in the particle-push loop per ion or electron, respectively, to perform the field averaging process. The difference between the ions and electrons can be explained by the fact that electrons are not configured to use gyro-points in these benchmarks, which is also the standard for simulations performed with EUTERPE.

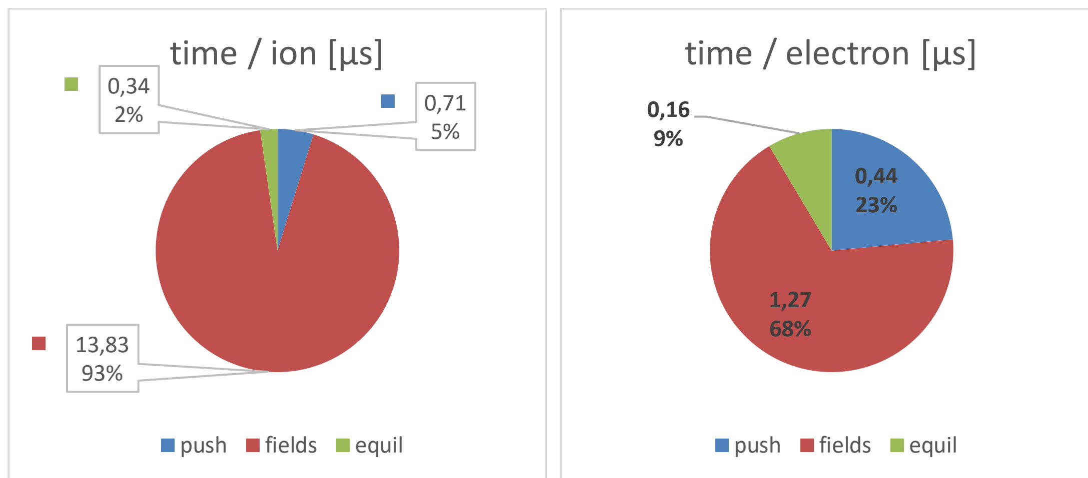

Fig. 33 The time the push loop takes to complete per particle for different subparts of the particle loop, for ions on the left side and electrons on the right side. The upper number is the time in µs, the lower number is the percentage in terms of the total time the loop takes.

The next evaluation, shown in Fig. 33, compares the time the particle-push loop needs to finish one particle, split into different subparts of the particle loop. The large majority of the run time is spend averaging the fields. It has almost the same structure as Fig. 32, but the subtle differences do tell a story. Comparing Fig. 32 and Fig. 33, the field averaging takes a smaller percentage of the runtime in comparison with the percentage of FLOPS it takes. This means that the "push" and the "equil" part are using more time, doing less FLOPS. This additional time can only be spend waiting for data. Therefore, the field averager is able to use the memory system slightly more efficiently. From these two first evaluations, it becomes apparent that the major hotspot of the particle

pushing algorithm is found in the “fields” part. We will therefore concentrate our efforts on optimizing this part.

Fig. 34 The amount of bytes read from main memory for different subparts of the particle loop, for the ions on the left and the electrons on the right. The values are per particle. Additionally, each subpart is split into a “read” and “write” value.

Fig. 34 analyzes the data usage of the particle-push loop. We can make some interesting observations. First, the gyro-point averaging is again a very important factor. Ion pushing uses most of the consumed data in order to produce the averaged field value. About two thirds of the data is used that way, while electrons use only one third for this purpose. Second, the total amount of data is very different for the different species. Ions use about 9 Kbytes per particle, while electrons use only about 1 Kbyte per particle. This difference is very important for later optimizations as the amount of data per FLOP, the so-called arithmetic intensity, is one of the most important metrics when gauging the potential speed-up. Third, the “fields” part of the ion benchmark actually writes data. This is surprising, as this part of the algorithm does not need to produce any lasting results. The averaged field value is used to push the particle and is consequently discarded. The same holds true, albeit to a lesser degree, for the equilibrium segment of the algorithm for ions. We hope that this problem may be susceptible to later optimization efforts.

Combining the previous data, we can compute the algorithmic intensity of the different push loop segments for the different species. Combining this with the theoretical peak performance of the Cobra machine (see section 6.2), we can plot roofline diagrams and locate the investigated algorithms in these plots (Fig. 35). This plot gives an overview over the potential maximum speed-up any optimization can hope to achieve. The percentages given in this plot set a maximum performance gain, under the caveat that the optimization is not able to change the algorithmic intensity of the algorithm.

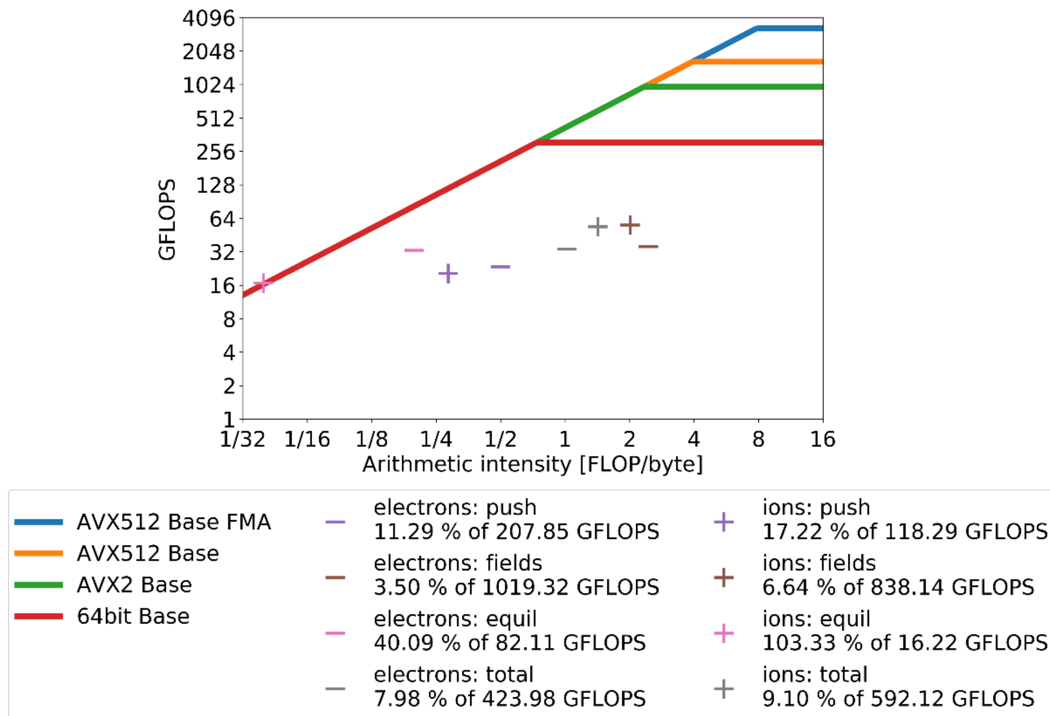

Fig. 35 Roofline plot for EUTERPE on Cobra, for different species and different segments of the particle-push loop. “Base” refers to the base frequency of the core. ITB is not used for this plot. We also give the percentage of the maximum performance as well as this maximum performance value for each species-segment combination, which is related to its specific arithmetic intensity value.

We can make a couple of observations. First, it is unlikely that FMA will help a lot in the particle-push loop. The algorithmic intensity is much too low. Second, the part of the algorithm with the highest potential of optimization seems to be the field-averaging segment. For ions as well as electrons, this part shows remarkably low performance in comparison with its arithmetic intensity. This part is also the one with the highest likelihood of cache misses, as the access pattern to the field values may be almost random in nature. Since this is likely a large part of the explanation for the observed performance, we will investigate the cache behavior in the following. Third, the push part may only gain about a factor of four by improved vectorization without changing its arithmetic intensity. This will be of interest for section 5.4 of this report. Fourth, the equilibrium component of the particle-push loop seems to operate at its maximum performance already. It can only be improved by changing the arithmetic intensity. Fifth, the particle-push loop as well as each single individual component are likely to be memory bound and not compute bound on SKL. Algorithmic modifications may change this, but it is unlikely. Sixth and last, we can conclude that the “push” as well as the “fields” part can still improve in performance and are currently not pushing a machine performance boundary, as seen from a roofline model perspective.

In the following, we will try to pinpoint the current performance ceiling of the algorithm. For this, we will inspect the cache behavior of the current implementation. Fig. 36 shows the L2 cache miss rates. The L2 cache is performing acceptably as a working cache. Since the L2 cache gets many hits in comparison to the L3 cache, caches misses, which lead to memory loads, are not pronounced and less visible. The performance of the L3 cache is shown in Fig. 37. The L3 cache is not performing as a cache at all, since the majority of requests are misses. The “push” part suffers from cache misses in the particle array. The “fields” part has the best behavior but is still rather bad. The small size of the `Phi_bspl`, `Apar_bspl`, `Apar_sympl_bspl`, `Apar_ham_bspl_sum` arrays (Table 6), used in the “fields” part, makes this possible. The caching behavior of these fields is good and overshadow the effects of the `b_equ_getfield` array, which cannot be cached properly. Please refer to Table 6 for the exact values and section 5.6 for further discussion.

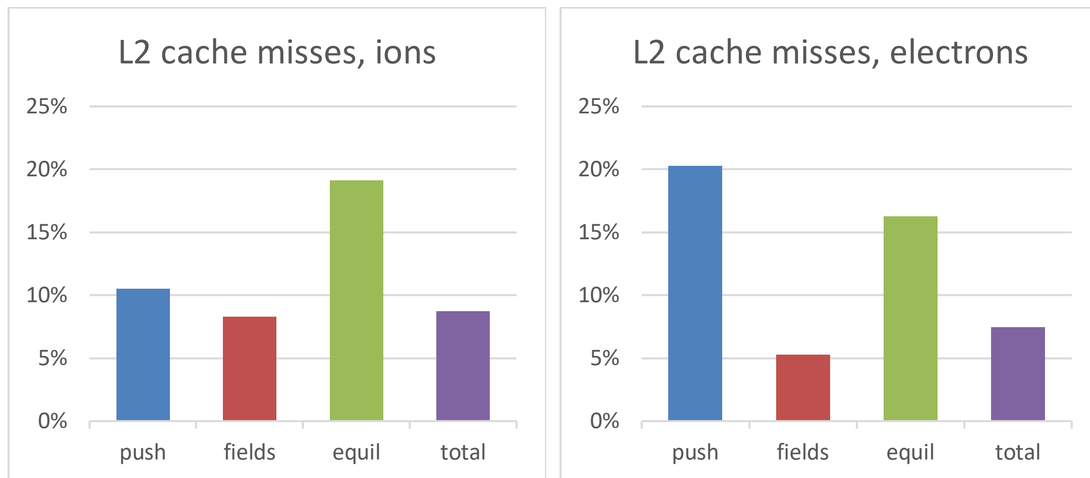

Fig. 36 L2 cache misses for ions and electrons.

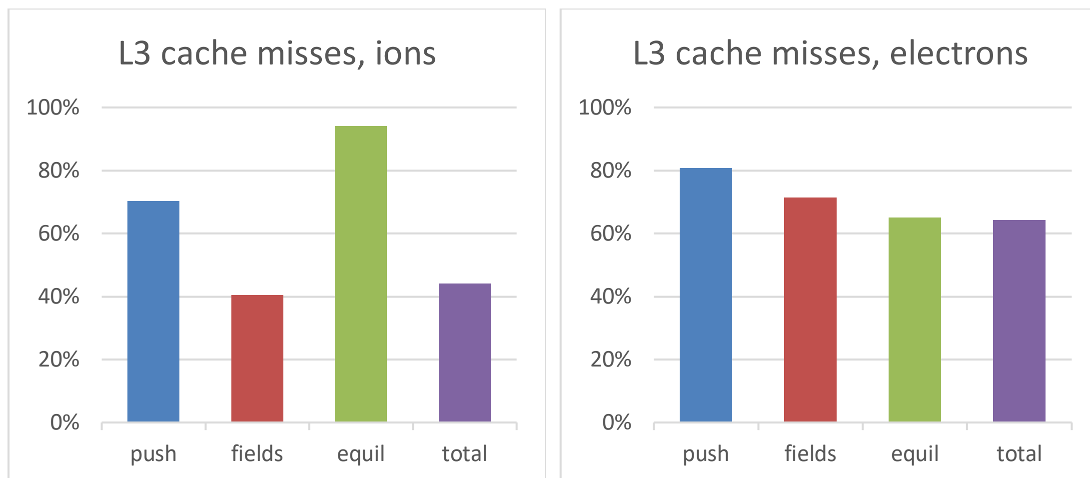

Fig. 37 L3 cache misses for ions and electrons.

There is an obvious difference between the two particle kinds. The difference in the "fields" part can be explained by noting that the electrons, employing no gyro-points, use far less data of the small arrays. These smaller arrays (`Phi_bspl`, `Apar_bspl`, `Apar_sympl_bspl`, `Apar_ham_bspl_sum`) generate a lot of cache hits for the ions.

5.4. *Revisiting the EUTERPE HLST project (EHP)*

During the EHP [2] [3], former HLST core team member Nicolaj Hammer modified the EUTERPE code in order to improve its performance. A major part of these adaptions was concerned with the exploitation of vectorization capabilities of modern processors. The target processor for this project had a vector register length of only 128 bits and could therefore perform two double precision calculations using one instruction. The current generation of processors, Skylake SP (SKL), contains 512-bit wide register, which can hold up to eight double precision values, which are coupled with two vector processing units resulting in potentially 16 floating-point operations per instruction. If these instructions happen to be multiply-add instructions, we can gain another factor of two. We therefore deem it worthwhile to revisit the changes made by N. Hammer and see if they are able to shine now that there is an increased vector length and instructions support available.

Before analyzing the new performance data of the modified code base, we will briefly describe the modifications. Similar to this project, the EHP focused on the particle-pushing algorithm.

The most efficient way to exploit the vector capabilities in regards to the particle-pushing algorithm would be to vectorize over the particle loop. Vectorization is most efficient if the amount of loading and storing to and from the vector registers from and to non-vectorized storage is kept to a minimum, as each of these operations will need to do a gather or scatter operation. Vectorized loads and stores are much better but require the data to be in the correct order in memory. Hence, vectorization is best done in a way, which makes sure that all calculations can consistently be done in the vector registers without any intermediate serialization. The 32 vector registers of SKL SP make this more likely but still non-trivial. Unfortunately, the particle-push loop body contains a multitude of complex and varied function calls. If these are to be vectorized in an efficient manner, vector versions of all functions are required and these function calls need to be efficient as well. Please refer to chapter 5.8 for a short discussion of this problem. The complex loop body was very likely the reason for the decision to try a different approach for the EHP.

Instead of attempting to fully vectorize the particle loop, the loop was split up into sub-loops, which were then vectorized individually. An illustration of these changes can be found in Fig. 38. This change transforms the single particle loop into a series of loops nested in an outer loop. Each inner loop performs the operations of a certain subregion of the previous loop but uses only a select subgroup of the particles. We then iterate over these subgroups in the outer loop. Through this rearrangement, it becomes possible to isolate specific operations and vectorize them without regards for the other operations in the loop. This allows for explicitly serialized regions as well.

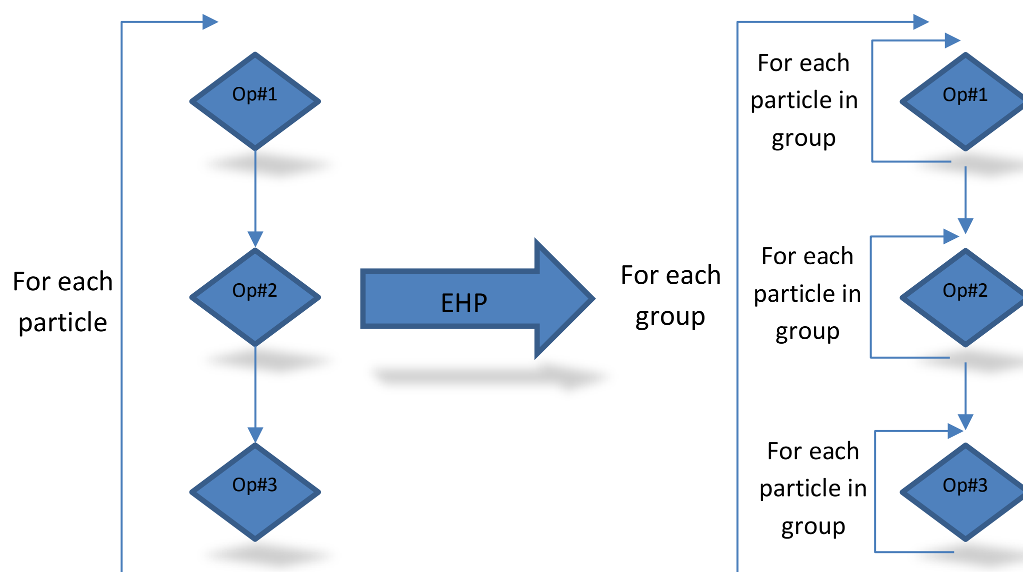

Fig. 38 Illustration of the modifications to the particle loop made during the EHP. “Op#n” refers to three different groups of operations performed inside the particle loop. The smaller inner loops on the right side were subsequently vectorized.

The drawback to this approach is the forced serialization of the data in between each pair of inner loops. As each inner vectorized loop may utilize all vector register in order to perform the vectorized calculations, it becomes highly unlikely that the particle data remains in the vector registers in between loops. This very likely leads to additional instructions to gather and scatter the particle data.

One could argue that the compiler should indeed be able to perform this kind of optimization itself. Since the advent of the OpenMP 4.0 standard and its new powerful vectorization control pragmas it is indeed possible to force the compiler to vectorize the particle loop. Since it will then naturally need to work around linear regions and functions, it may very well result in similar code. The compiler does not do this optimization automatically as its heuristics tell it that it is not worthwhile. We will investigate the performance implications of the EHP modifications in the following and will find out if the compiler is right.

We need to define a suitable performance measure in order to properly determine advantages and disadvantages of the optimization effort. The metric we choose is the number of pushed particles per second. This metric is inversely proportional to the average time a single particle needs to finish its iteration in the particle loop. This metric is computed by measuring the time it takes each rank to finish all iterations (the marker API allows to specifically measure the particle-push loop), computing the average of this number over all ranks and dividing by the total number of particle pushes performed during the whole benchmark: $P = N_{\text{pushed}} N_{\text{ranks}} / \sum_{\text{ranks}} t_{\text{rank}}$. The correct formula would be $P = \sum_{\text{ranks}} (N_{\text{pushed,rank}} / t_{\text{rank}})$, but the difference between these formulas should be negligible. The first formula uses the average time each rank takes, while the second formula uses the individual time of each rank.

We will need to differentiate between particle kinds for this metric as well, as electrons do not use gyro-particles and ions do. This makes the push algorithm very different for the two particle species. This is achieved by using two sets of marker regions.

The test cases, which were used for the EHP version of EUTERPE, are very likely the same test cases Nicolaj Hammer used. Nicolaj's changes are based on EUTERPE 2.62, which is from now on referred to as the original version, and use the complex version of the code. They use a relatively small grid size of 16x16x16. We changed the original test case slightly by enabling kinetic electrons.

The benchmark configuration pushes 128 million particles, equally divided in electrons and ions, twice for each step of 80 steps using 16 processes. For all runs, we used a memory alignment of 64 bytes, enabled interprocess optimization, and pinned the 16 processes consecutively to one node of the Cobra cluster (Intel Xeon Gold 6148F CPUs) in a way that each socket received eight processes. This was done in order to maximize the available bandwidth. The theoretical peak performance characteristics of this system are shown in section 6.2. We carefully checked the optimization reports of the compiler and adapted the block size to fit to the larger vector registers. We made sure that the optimized code version is using vectorized instructions on aligned data wherever it was optimized by Nicolaj to do so.

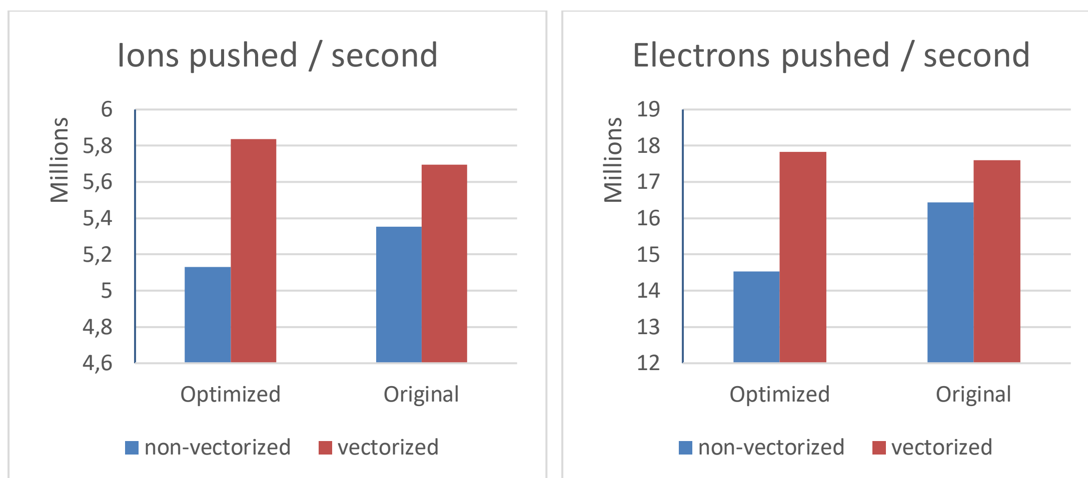

Fig. 39 The principal performance metric, the number of pushed particles per second for the ions on the left and the electrons on the right. Please carefully note that neither *y*-axis starts at zero.

A comparison, of the principal performance between the code, which was optimized during the EHP, and the original code, can be found in Fig. 39. Interestingly, the performance metric does not change meaningfully. The original version is slightly faster in the non-vectorized version, which shows the overhead the code changes introduce, while the optimized version is slightly faster in the vectorized version. Unfortunately, the speedup is only about 2.5 % for ions and 1.3 % for the electrons.

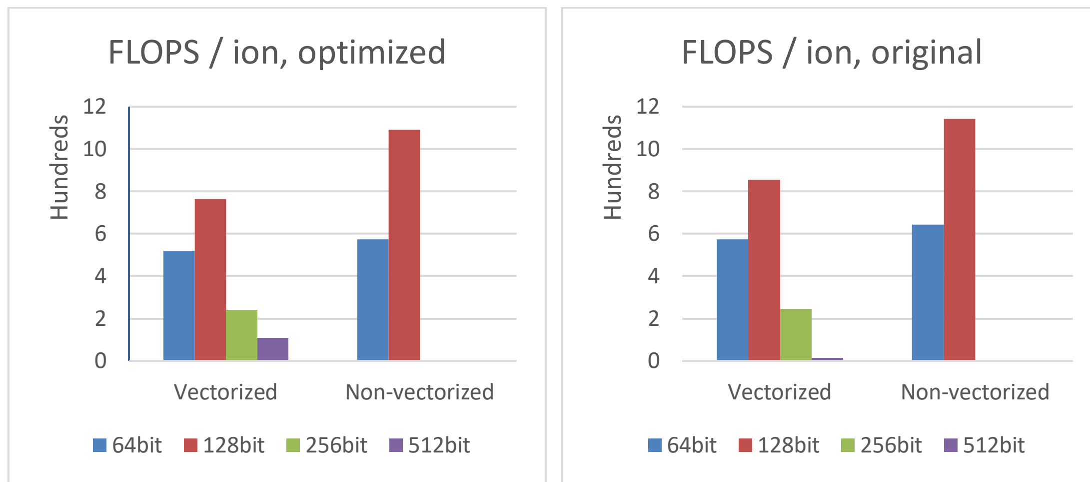

Fig. 40 Comparison between the different types of instructions used to perform floating-point operations per ion. The given number is the amount of double precision floating point operations performed using a specific vector instruction length. This is not the amount of retired instructions. Larger vector instructions perform more floating-point operations per instruction, which has been factored into the presented numbers.

One interesting other number, which may help to understand this behavior, is the number of AVX FLOPS, which is the number of floating point operations, which either use AVX2 or AVX512 operations, per particle push. You can find a comparison of the amount of different instructions used in order to perform floating-point operations in Fig. 40 and Fig. 41. These figures are the key to understanding why the optimization effort did not yield any meaningful results. For ions, the optimized code version calculates 109 FLOPS using AVX512 instructions, while the original version calculates only 14 FLOPS this way. This difference is, unfortunately, not at all meaningful since the ion particle loop needs about 1650 FLOPS to complete. Why is the code not using more AVX512 instructions, even though many loops have been vectorized? The reason for this is the fact that a major part of the FLOPS in the particle-pushing algorithm is used for field interpolation (Fig. 32). The field interpolation does not lend itself well to being vectorized and has not meaningfully changed in the optimized code version. So while Nicolaj's modifications did speed-up the specific code part which is responsible for the actual particle pushing, it did nothing for the far more expensive field interpolation part of the particle loop. This is also visible in the electron data (Fig. 41). Electron computation needs less field averaging, as shown in Fig. 32, so it is more susceptible to Nicolaj's optimizations. It therefore uses many more AVX instructions in the optimized version. They still do not produce an appreciable speed-up, though, which is probably because the additional overhead of the vector processing eats up the improvements.

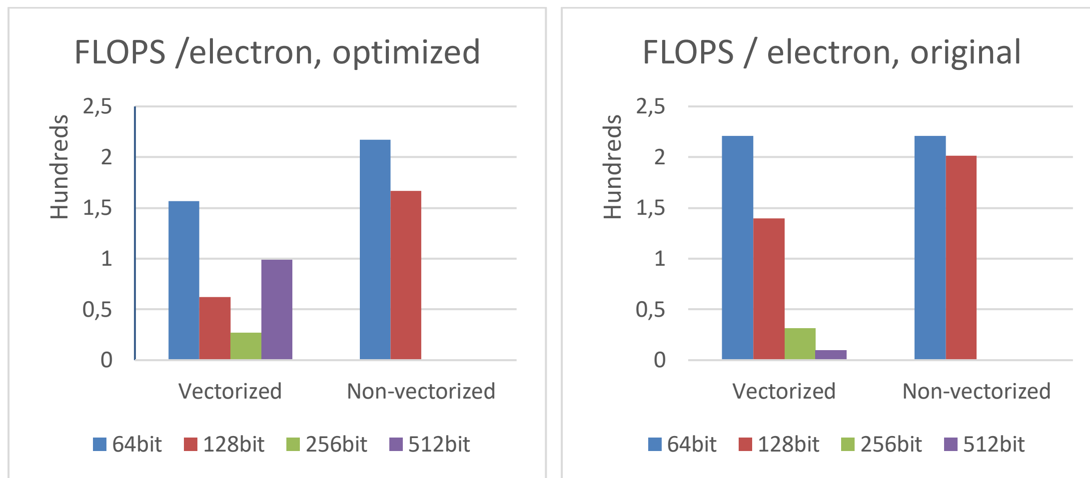

Fig. 41 Comparison between the different types of instructions used to perform floating-point operations per electron. The given number is the amount of double precision floating point operations performed using a specific vector instruction length. This is not the amount of retired instructions. Larger vector instructions perform more floating-point operations per instruction, which has been factored into the presented numbers.

These numbers show that the optimizations do not offer any performance benefit even when used on machines with larger vector registers. These findings fit well to the observations made in section 5.3. In that chapter, we have seen that the "push" part of the algorithm plays only a minor role in the overall performance. Since this was the main code segment, which was optimized by Nicolaj, these optimization are not able to produce any benefit. We will therefore not be concerned with these optimizations anymore and will search for different, more efficient optimization strategies in the upcoming sections.

5.5. *Structural Changes*

The most efficient way to increase performance is to decrease the amount of FLOPS in the executed code. This can be done by using arithmetic reordering or by getting rid of duplicate computations. In order to enable some significant reductions in duplicate computations we made some structural changes to the code. These changes will also enable improved software prefetching (chapter 5.6).

We introduce a new derived type called `GYRO_POINTS` (Listing 1).

```
type GYRO_POINTS
  SEQUENCE
  REAL, DIMENSION(NAVG_MAX) :: rt_x, zt_x, phit_x, R_hat, Z_hat, phi_hat
  INTEGER :: navg
end type GYRO_POINTS
```

Listing 1 Initial version of the new derived type `GYRO_POINTS`. The `SEQUENCE` keyword will allow for more efficient vectorization as it guarantees the order of the fields in the derived type and therefore their specific memory alignment in relation to the memory alignment of a variable of its type.

It is a structure of arrays holding the positions of the gyro-points in two different coordinate systems. Previously EUTERPE employed the arrays themselves without containing them in a structure. The new derived type makes the code more readable, since it reduces the amount of variables in the main loop considerably. We will also exploit this new structure to hold even more data when implementing subsequent optimizations, found in chapter 5.7.

```
PARTICLE_LOOP: DO ip = 1, npart_loc
...
CALL gyroring(rt, zt, phit, gyrofac, &
              B_abs, b_norm, gyro_tilt, &
              species, navg, .TRUE., gps)
...
IF (calc_pot == 1) THEN
  IF (neocl == 0) THEN
    IF (.NOT. cka_euterpe) THEN
      CALL getfield(rt, phit, st, chit, &
          partiallnB, gps, &
          Phi_bspl, mgPhi, species)
    ELSE
      mgPhi=RCZERO
      DO mode=1, n_modes
        CALL getfield(rt, phit, st, chit, &
            partiallnB, gps, &
            eivec_phi_bspl(:,:,:,mode), &
            mgPhi_mode(:,mode), species)
        ...
      END DO
    END IF
  ELSE
#if defined(rcversion_real)
    IF (boltzmann == 0) THEN
      CALL getfield(rt, phit, st, chit, &
          partiallnB, gps, &
          Phi_bspl, mgPhi, species)
    ELSE
      CALL getfield(rt, phit, st, chit, &
          partiallnB, gps, &
          Phi_bspl, mgPhi, species, Psi1)
    END IF
#endif
  END IF
  mgPsi = mgPhi
  IF (elmag == 1) THEN
    CALL getfield(rt, phit, st, chit, &
        partiallnB, gps, &
        Apar_bspl, mgApar, species, Apar)
    ...
  END IF
  IF (elmag_pullback == 1) THEN
    CALL getfield(rt, phit, st, chit, &
        partiallnB, gps, &
        Apar_sympl_bspl, mgApar_sympl, &
        species, Apar_sympl)
    CALL getfield(rt, phit, st, chit, &
        partiallnB, gps, &
        Apar_ham_bspl_sum, mgApar_ham, &
        species, Apar_ham)
    ...
    IF (use_rmh_scheme) THEN
      CALL getfield(rt, phit, st, chit, &
          partiallnB, gps, nparPhi_bspl, &
          bufc, species, nparPhi)
    END IF
  END IF
ELSE
...
END IF
END DO
```

Listing 2 Modified code in `part.f90`. All occurrences of `gps` used to be varying combinations of at least three of `rt_x`, `zt_x`, `phit_x`, `R_hat`, `Z_hat` and `phi_hat`. This makes the code more readable as it is immediately clear which function needs to have information about the gyro-points. The amount of those functions will also increase through additional optimizations, which are detailed in later chapters of this report.

Having a derived type makes these changes possible without additional changes in the affected code parts. You can find the affected code parts in Listing 2. The number

of gyro-points `navg` was included in the structure as well, since it is equal to the size of its arrays.These changes bring a slight performance improvement with them as well. Too many function arguments in hot paths of the code may adversely affect performance in a meaningful way, since more elaborate calling conventions need to be invoked by the compiler. This change enables EUTERPE to push about 4.2 % more ions, 2 % more electrons and decreases the wall time of the main loop of the linear test case by 2.1 %.

5.6. *Software prefetching*

Modern CPUs suffer from the so-called bandwidth gap. This term describes the widening gap between processor and memory speeds. Loading a cache line from memory takes approximately 200 cycles on SKL and may break pipelining as well. One widely employed tool to fight this shortcoming is the usage of prefetching. Two different kinds of prefetching are possible in most modern CPUs.

Hardware prefetching uses dedicated transistors in the CPU to track and pre-emptively load specific addresses from main memory to the cache hierarchy. This happens before the currently running software even exhibits a need for them. The most common and simplest form is the prefetching of subsequent memory to a previously requested address. The hardware prefetcher will look for contiguous memory access and start prefetching addresses, which follow previously requested ones. Current CPUs employ much more sophisticated strategies to foresee which parts of the memory might be useful in the future. Unfortunately, CPU manufacturers do not make the specifics of these algorithms public. Developers are taught to make use of hardware prefetching and it has indeed become one of the most important strategies in writing efficient code.

Evidently, hardware prefetching cannot be foolproof. The simplest counterexample is an algorithm, which needs data from randomized locations in memory. In this case, the hardware prefetching mechanism may actually hurt performance as unnecessary data starts to clog the cache hierarchy. GK PIC's particle pusher needs the field values at the locations of the gyro-points. Ordinary PIC can make this field access contiguous by sorting the particles by grid cell. Unfortunately, this is not possible for GK PIC as the gyro-points request rings of varying sizes around their center coordinate. If the hardware prefetcher is not able to prefetch the data, performance can suffer a lot. Only if the randomly accessed data is small enough to fit into the CPU caches, the performance will not degrade.

This is a good explanation for the behavior shown in Fig. 37. The randomly chosen memory locations, coupled with the large field size, make good caching behavior of EUTERPE impossible.

In this case, the case of memory access with the properties,

- Random memory access (no hardware prefetching) and
- Accessed memory region >> Last level cache (LLC),

it may prove useful to employ software prefetching. Most modern CPUs offer instructions, which generate memory loads into cache without actually using the respective memory addresses to perform calculations. These instructions are called prefetch instructions. SKL offers five such instructions [7]: `PREFETCHNTA`, `PREFETCH0`, `PREFETCH1`, `PREFETCH2`, `PREFETCHW` and `PREFETCHWT1`. The first instruction prefetches for non-temporal access, which marks the prefetched line for early replacement and fetches only into L2 cache. The numbered prefetches perform prefetching into specific cache levels: 0 -> L1 cache (+L2+L3), 1 -> L2 cache (+L3), 2 -> L2 cache for early replacement. `PREFETCHW` and `PREFETCHWT1` prefetch data for write access. Most compilers will offer dedicated functions or subroutines to insert these instructions into the binary. The Intel Fortran compiler offers the `mm_prefetch` subroutine for this purpose. Using this function, it is possible to start loading a certain address at a point in the program where it is not yet used, setting up for later calculations, which can then make use of it without incurring a cache miss penalty. It

can be seen as a form of hiding communication time. It is important to note that this is only a hint to the processor and may be ignored.

In order to understand the performance behavior of EUTERPE we need to understand the involved field sizes. Table 6 shows the sizes of the fields used in the “fields” part of the two test cases.

Field name	Linear	Turbulence
`b_equ_sc`	3.125 MiB	7.03125 MiB
`rho_sc`	0.390625 MiB	0.88 MiB
`b_equ_getfield`	30.52 MiB	1.37 MiB
`Phi_bspl`	102.09375 KiB	2.59 MiB
`Apar_bspl`	102.09375 KiB	0 MiB
`Apar_sympl_bspl`	102.09375 KiB	0 MiB
`Apar_ham_bspl_sum`	102.09375 KiB	0 MiB

Table 6 The sizes of the fields used in the “fields” part of the particle-pushing algorithm. Values are given per rank and may be rounded.

Most fields will fit into the cache of one core, which has a size of 2.375 MiB (L2+L3). The hardware prefetcher will be able to spot the frequent usage of the smaller fields and will likely keep them in memory even when the larger fields need to be accessed. One exception to this is `b_equ_getfield`, a field that holds equilibrium field data, needed for determining the electromagnetic fields at the particle positions. Its size of over 30 MiB guarantees that it will not be stored in cache. Since its access is random, it is a good target to test software prefetching. The other exception is `b_equ_sc`, which is only used in the very center part of the simulation box. According to hotspot analysis, its access is very rare and unimportant for performance considerations. According to [6], the prefetch distance is not important as long as it is above a certain threshold, which correlates with the memory latency. In our case, a prefetch distance of about 200 cycles should therefore be sufficient.

The turbulence test case is not meaningfully different in most cases, except that it does not suffer from the huge `b_equ_getfield` array. The increased size of the `Phi_bspl` does not suffice to change the behavior in a meaningful as it still fits into the cache.

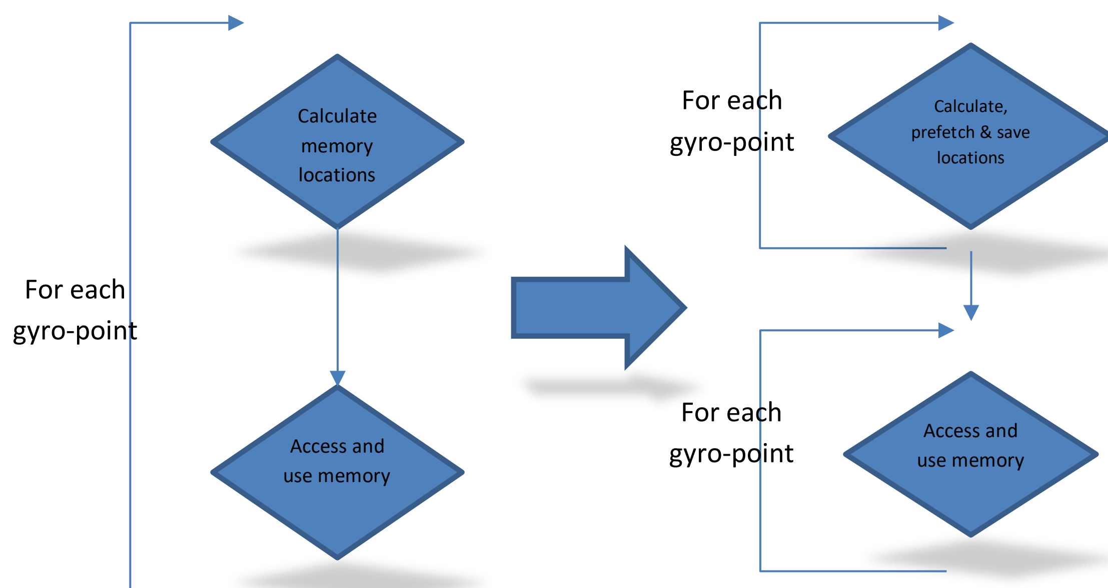

Fig. 42 General restructuring of gyro-point loops to enable software prefetching.

Fig. 42 shows the general structure of gyro-point loops in the EUTERPE code and the structural change to incorporate software prefetching. These loops are found inside the particle loop and iterate over the gyro-points of each particle. They are also found in other parts of the code, for example the charge assignment function. In general, the maximum number of gyro-points for each particle is limited to 32. This sets the maximum number of iterations of this loop. Splitting the loop into two loops, whereby the first loop solely generates the subsequently needed memory locations, enables software prefetching of these locations without incurring meaningful overhead as the amount of buffered data is quite small. The prefetch distance is sufficient as well.

Implementing software prefetching for the equilibrium field `b_equ_getfield` in the gyro-point loop (in `equil.F90:equil_for_getfield_phi()`) increases the number of ions pushed per second by 12.7 % and the number of electrons pushed per second by 3.5 %. We implemented it for other occurrences of similar loops as well but achieved no speed up, which is consistent with the previous explanations and Table 6. The total wall time of the main loop of the linear test case decreases by 5.1 %.

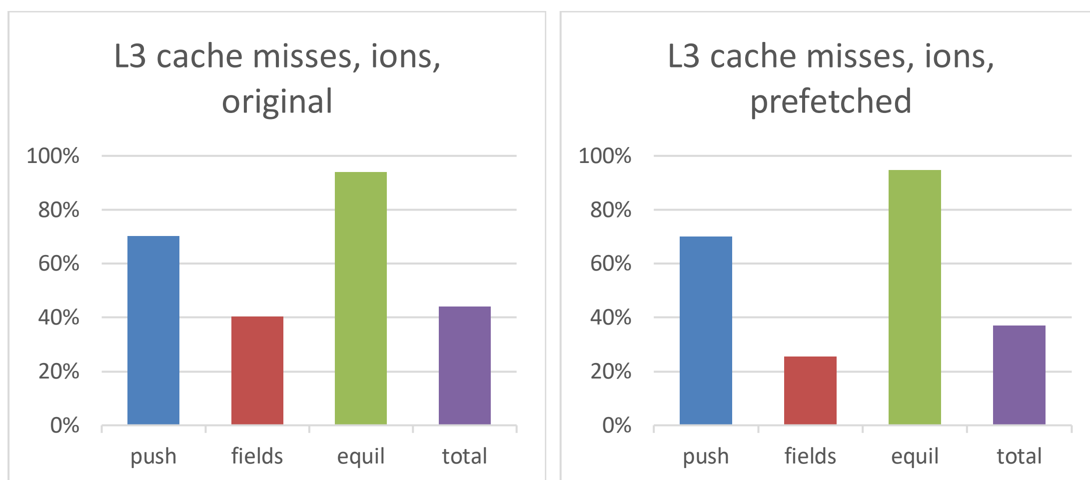

Fig. 43 L3 cache misses for ions for the original code and the optimized code using software prefetching for the `b_equ_getfield` array.

In order to test our model, we measured the L3 cache misses with `b_equ_getfield` prefetching. The results are shown in Fig. 43. The cache misses in the "fields" part have been reduced quite meaningfully.

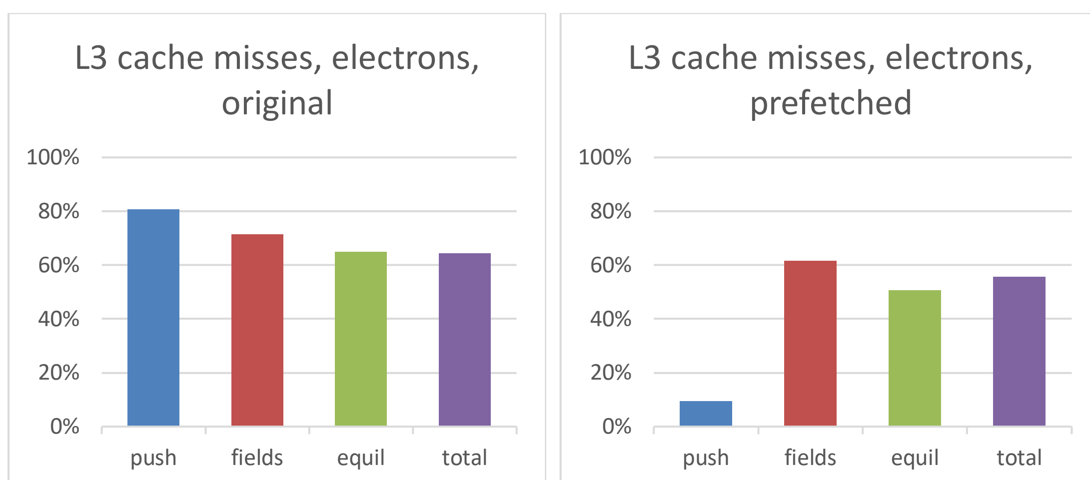

Fig. 44 L3 cache misses for electrons for the original code and the optimized code using software prefetching for the `pic1_loc`, `pic2_loc` and `work1_loc` array.

We also tried software prefetching on the "push" part of the particle loop. Even though we do not expect great returns in performance, as this part plays only a minor role in the overall computation, we were interested to see the effects. This prefetching does not follow the outline shown in Fig. 42. Instead, we introduced prefetching of the next

elements in the particle arrays inside the particle loop, directly after the “fields” part. Since the whole loop takes about 30000 cycles to complete, this should be a sufficiently large prefetching distance. Tests with different positions confirmed this. Fig. 44 shows the L3 cache misses in the case of software prefetching of the `pic1_loc`, `pic2_loc` and `work1_loc` arrays. We can see that in the case of electrons, the L3 cache misses almost entirely disappear. We are able to push about 1.5 % more electrons per second with that improvement. Surprisingly, this change brought no improvements to the ion performance with it (it even decreases ions pushed by 0.4 %, which may very well be noise). The total wall time of the linear test case increased by 1.8 %, which is why we do not include this change in our final assessment.

5.7. *Removing redundant computations*

An illustration of the original main particle-push loop is shown in Fig. 45. The call sites of the `getfield` function are shown in Listing 2. It can be called multiple times for each iteration of the particle-push loop. The parameters of our benchmark result in the `getfield` function being called four times, interpolating the `Phi_bspl`, `Apar_bspl`, `Apar_sympl_bspl` and `Apar_ham_bspl_sum` fields (these are some of the potential representations of dummy argument `pot_bspl`).

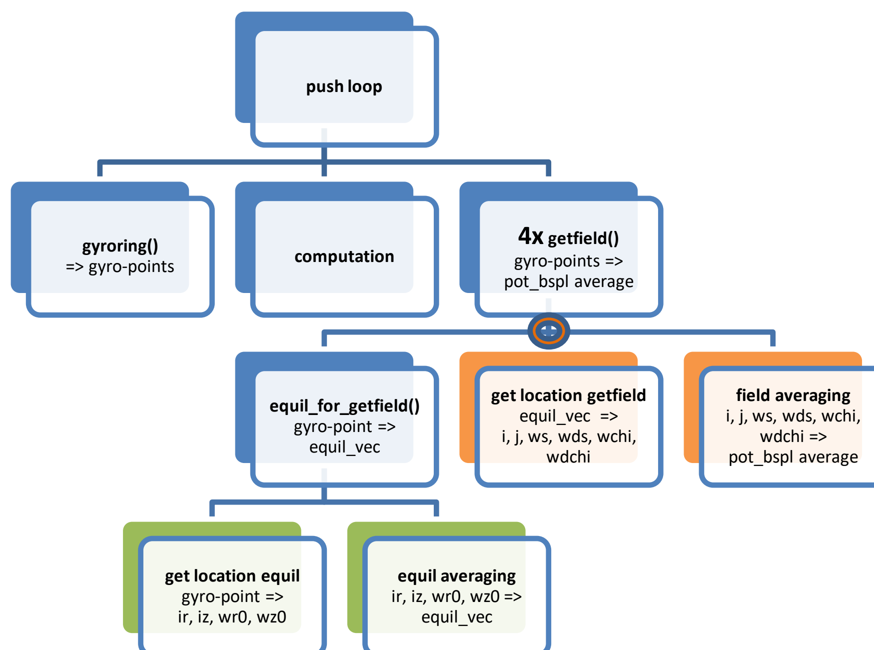

Fig. 45 Initial structure of the particle-push loop. Program execution goes from left to right, entering lower levels where available. The orange circle signifies the loop over the gyro-points. Green boxes mark functions, which generate the equilibrium field positions and average their values, while orange boxes mark functions, which generate the dynamic field positions and average their values. Names with () are functions in the EUTERPE code base. Expressions containing a “=>” give the input and output values of the respective step in the program. `pot_bspl` is a dummy argument in the `getfield` function and stands for any field, which may be interpolated by it. It is therefore a different array for different calls of the `getfield` function.

The included dependencies between the different code steps (given by the expressions containing a “=>”) in Fig. 45 make it apparent that there is a meaningful amount of duplicate computations in the `getfield` function. The only requirements to compute a specific `pot_bspl` average value are the values of `i`, `j`, `ws`, `wds`, `wchi` and `wdchi` for each gyro-point. We can reuse these values for each call of the `getfield` function (for a different `pot_bspl`) and do not need to recompute them every time. By computing these four values for each gyro-point once and buffering them, we can reduce the amount of FLOPS necessary to push a particle. Since the amount of gyro-points per particle is very small (< 32) this will not incur a memory reload penalty. The full computation of a single particle should always fit into the cache

of our system even when these additional buffers are accounted for. This was shown in chapter 5.3, where we measured the amount of memory read per ion to be about 9 KiB.

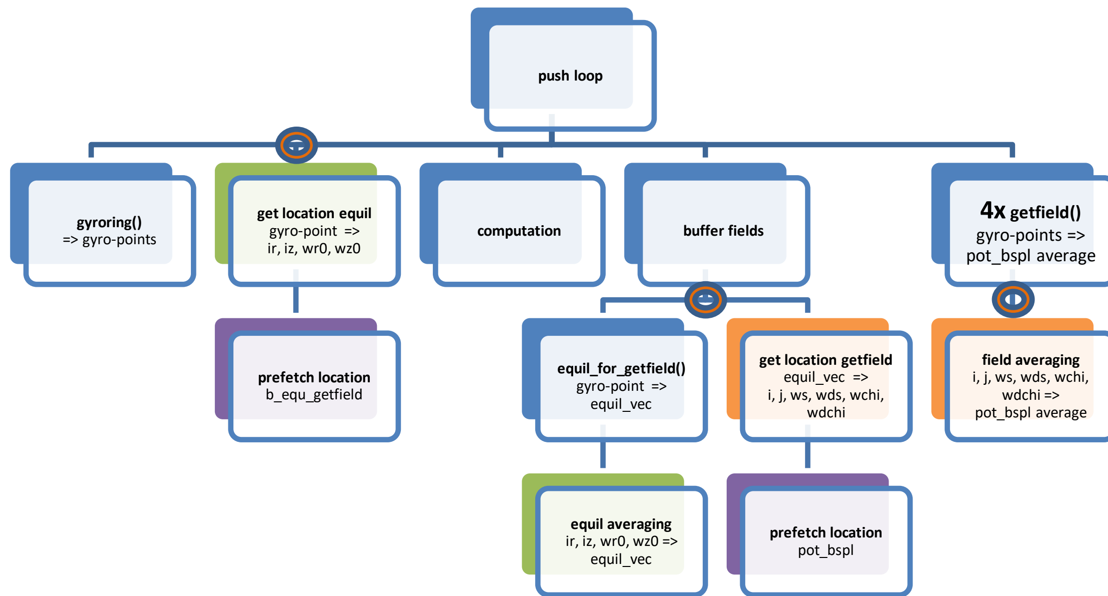

Fig. 46 New structure of the particle-push loop. Program execution goes from left to right, entering lower levels where available. Orange circles signify a loop over the gyro-points. As in Fig. 45, green boxes mark functions, which generate the equilibrium field positions and average their values, and orange boxes mark functions, which generate the dynamic field positions and average their values. The two boxes of each color are now in two different gyro-point loops, making use of the transformation shown in chapter 5.6. Names with () are functions in the EUTERPE code base. Expressions containing a "=>" give the input and output values of the respective step in the program. The new purple boxes contain added prefetch functionality (explained in chapter 5.6 as well). `pot_bspl` is a dummy argument in the `getfield` function and stands for any field, which may be interpolated by it. It is therefore a different array for different calls of the `getfield` function.

The new code structure is shown in Fig. 46. Its main advantage is the fact that it gets rid of any redundant computation. However, it also offers improved prefetching distances, which may also be modified more easily. Chapter 5.6 goes into more detail how prefetching improves performance. This improved prefetching distance does not translate into improved performance, as the initial prefetch distance seems to already be large enough. The new structure did allow to test this thoroughly.

A prerequisite for these changes were the structural code changes explained in chapter 5.5. The new derived type for the gyro-points is an appropriate structure to hold all the buffered data. The new data structure is shown in Listing 3. The structure also contains some values, which are not given for each gyro-point separately. These relate to the position of all gyro-points on the phi axis. The utility of `navg_unfiltered` will become apparent in chapter 5.8. The order of the fields in this derived type was deliberately chosen to enable 64 byte aligned memory access in the case of vectorized code. The additional buffered data will increase the memory footprint per particle by less than 8 KB. This will still comfortably fit into sufficiently high cache levels.

These changes allowed for an increase of 32.4 % in pushed ions, 20.1 % in pushed electrons and a decrease of the wall time in the main loop of the linear test case by 10 %.

```
type GYRO_POINTS
  SEQUENCE
  REAL, DIMENSION(NAVG_MAX) :: rt_x, zt_x, phit_x, R_hat, Z_hat, phi_hat
  REAL, DIMENSION(NAVG_MAX) ::  incell_r, incell_z
  REAL, DIMENSION(8,NAVG_MAX) :: equil_for_getfield_vec
  INTEGER, DIMENSION(NAVG_MAX) :: cell_idx_s, cell_idx_chi, cell_idx_r,&
                                          cell_idx_z
  REAL, DIMENSION(-1:2,NAVG_MAX) :: ws_buffer, wchi_buffer
  REAL, DIMENSION(-1:2,NAVG_MAX) :: wds_buffer, wdchi_buffer
  REAL, DIMENSION(-1:2) :: wphi_buffer
  REAL, DIMENSION(-1:2) :: wdphi_buffer
  INTEGER :: cell_idx_phi
  INTEGER :: navg
  INTEGER :: navg_unfiltered
end type GYRO_POINTS
```

Listing 3 Final version of the new derived type `GYRO_POINTS`. The variable names are not fully compliant to the names in the other parts of this documentation. They have been chosen to make the code more easily readable, since the derived structure contains more fields that need to be distinguished properly. The correspondence is as follows: `ir ≡ cell_idx_r`, `iz ≡ cell_idx_z`, `wr0 ≡ incell_r`, `wz0 ≡ incell_z`, `i ≡ cell_idx_s`, `j ≡ cell_idx_chi`.

5.8. *OpenMP 4.0 SIMD clauses*

Generally, SIMD vectorization may improve the performance of loops if the programmer can supply some additional information to the compiler:

- The `private` clause tells the compiler that it needs to provide copies of a specific variable for each vector lane.
- Specifying a `reduction` allows the compiler to vectorize loops with reduction dependencies.
- Using the new `declare simd` directive the compiler may be able to vectorize loops containing functions calls. When using this directive it is always advised to use the `–vecabi cmdtarget` compiler flag. It tells the compiler to provide versions of functions fully exploiting the vector capabilities of the machine, without regard for linking problems on different processors. It is also important to note that the Intel Fortran compiler will always use call by reference, at least in the numerous tests we performed. It is not possible to keep the values in vector registers necessitating extra loads and stores. This must be kept in mind when assessing possible performance benefits.
- The `aligned` clause may improve performance if the compiler is not able to prove memory alignment. This is especially relevant for subroutine arguments. Users are always advised to use the `-align array64byte` compiler flag. This will align all arrays on 64-byte boundaries, which is the correct alignment for AVX512 vectorization.

Basis for exploiting the AVX512 capabilities of a processor is to explicitly allow the compiler to use the specific instruction set (via the command line flag `-xCORE-AVX512`) and the respective CPU registers (via the `-qopt-zmm-usage=high` command line flag). After some deliberation, we conclude that advanced users are recommended to not use the `mtune` command line flag. The trouble with this flag is its ability to reset configurations made through accompanying command line flags, which makes proper benchmarking and testing harder. Unfortunately, the Intel Fortran compiler is not yet perfectly stable with the new OpenMP pragmas. We encountered a host of compiler bugs and reported them to CINECA and Intel for fixing.

The main hotspot of the code can be found in the body of the two innermost loops of function `getfield_phi`. These loops iterate over the, in our test cases, 27 field values, needed to compute the field acting on a specific gyro-point. We tried to force explicit vectorization of these loops by introducing the pragma, shown in Listing 4, in front of it. It is important to add the `-qopenmp-simd` flag to your compilation command line, even though Intel documents say openmp simd is enabled by default, since, in our case, it was not enabled by default.

```
!$omp simd reduction(+:field_s,field_chi,field_phi,pot) &

!$omp      private(temp1,temp2) collapse(2) &

!$omp      aligned(ws,wphi,wchi,wds,wdphi,wdchi,pot_bspl)
```

Listing 4 OpenMP 4.0 SIMD pragma introduced in the `getfield_phi` function in the `field.f90` source file.

Unfortunately, this did not give any performance improvements. Going through the compiler reports and investigating the assembly of the generated binary, it is apparent that the contents of the loops, shown in Listing 5, do not lend themselves well to vectorization. There are multiple reasons for this.

First, there are multitudes of irregular memory accesses in the code. The problem, in this case, is not the caching of data. In this case, the `pot_bspl` , `wphi`, `wdphi`, `wchi` and `wdchi` arrays are accessed in a non-linear fashion throughout the course of the loops. Due to these accesses, the compiler needs to include vector gather instructions into the assembly. According to [8] these instructions have a reciprocal throughput of about 5 cycles (up to 9 cycles if collecting floats) and need four ports out of the available seven, which makes it very bad for pipelining. Additionally, many instructions are spend on preparing the necessary index vectors. This rather high cost makes these new instructions dubious in their utility for increasing performance. They may be beneficial in loops with a better ratio between non-linear memory accesses and computation. From this discussion, we can conclude that their main intended purpose is not performance, but the generation of more vectorization possibilities, enabling the compiler to vectorize previously non-vectorizable loops. Including larger arrays, with copies of the values for each iteration of the loops, did not help performance even though it got rid of some fraction of the irregular accesses.

Second, the loops have only three iterations each, which hinders proper pipelining. Third, sporadic usage of AVX512 or AVX2 registers may incorporate additional overhead of a certain up- and down spinning of the respective hardware ([9], reported for Skylake, but highly likely relevant for SKX as well).

```
DO kc = -1, 1
      DO jc = -1, 1
            temp1 = ws(-1) * pot_bspl(i - 1, j+jc+1, k_loc+kc+1) + &
                   ws(0) * pot_bspl(i, j+jc+1, k_loc+kc+1) + &
                   ws(1) * pot_bspl(i + 1, j+jc+1, k_loc+kc+1)
            temp2 = wphi(kc) * wchi(jc)
            pot = pot + temp2 * temp1
            field_s = field_s + temp2 * &
                  (wds(-1) * pot_bspl(i - 1, j+jc+1, k_loc+kc+1) + &
                   wds(0) * pot_bspl(i, j+jc+1, k_loc+kc+1) + &
                   wds(1) * pot_bspl(i + 1, j+jc+1, k_loc+kc+1))
            field_chi = field_chi + wphi(kc) * wdchi(jc) * temp1
            field_phi = field_phi + wdphi(kc) * wchi(jc) * temp1
      END DO
END DO
```

Listing 5 The body of the innermost loop in the `getfield_phi` function in the `field.f90` source file. The values of `i`, `j`, and `k_loc` are constant during the execution of the loop. This is the updated version of the loop, which was reordered to improve performance, as explained in chapter 5.9.

Fortunately, the most expensive loop in the code, the one that encapsulates the code shown in Listing 5, has some more instructions. We can employ the tactic used by Nicolaj Hammer in the EHP project, elaborated on in chapter 5.4. We can split up larger, unvectorizable loops into smaller ones. Since the first part of the gyro-point loop (Listing 5) is not vectorizable, it is impossible to vectorize later parts of it. Splitting off the later parts makes them vectorizable. This process is comparable to the process that is illustrated in Fig. 38. Unfortunately, the loop contains an early out for gyro-points, which lie outside of the domain. This would make vectorization much harder. To get rid of this obstacle we implemented a gyro-point filter, which shrinks the gyro-point array, making an early out unnecessary. For this reason we need an extra `INTEGER` value in the `GYRO_POINTS` derived type named `navg_unfiltered`. This will contain the original amount of gyro-points, which is used for averaging the field values. The code therefore assumes that the fields outside of the simulated box are zero.

It was also important to remove another case of branching in this loop. The code differentiates between species inside the loop. This can be moved outside with the additional benefit that the electron branch does not even contain a loop over `navg` anymore. Electrons always have `navg = 1` as they do not employ gyro-points. A subsumption of all these modification can be found in Listing 6.

Due to implementation details, these changes rely on the changes introduced in chapter 5.7. Therefore, we can only give the combined speed-up value of these two modifications. Removing redundant computations while employing the loop break-up with a SIMD vectorized second loop allows for 82.7 % more ions and 25.9 % more electrons being pushed and a decrease in the wall time of the main loop of the linear test case by 19.3 %.

```
DO l = 1, navg
  IF … THEN
    CYCLE
  END IF
  load_location
  Listing 5 content
  heavy_computation(l)
     SELECT CASE (species)
       => field_av, pot_av
END DO
```

```
filter_particles
DO l = 1, navg
  load_location
  Listing 5 content
  buffer field_s, field_chi,&
           field_phi, pot
END DO
SELECT CASE (species)
CASE (IONS, FAST)
!$omp simd reduction(+:field_av,pot_av)&
!$omp private(…) &
!$omp aligned(…)
DO l = 1, navg
  heavy_computation(l)
       => field_av, pot_av
END DO
CASE (electrons)
  heavy_computation(1)
       => field_av, pot_av
```

Listing 6 The left box shows the original version of the code, the right box shows the optimized version enabling vectorization.

5.9. *General code changes*

We improved the code structure in several places by reducing the amount of `float` to `integer` conversions. This has been done only in major performance hotspots of the code since the performance differences per call are not very big. The specific changes are shown in Listing 7 and Listing 8. We also reordered the most expensive loop, leading to the form shown in Listing 5. We can push about 18.2 % more ions and 8.3 % more electrons with these optimizations. The total wall time of the linear test case decreases by 7.6 %.

```
cell_idx_r = INT((loc_r-sgrid(0))
             * dsgrid_inv)
incell_r = (loc_r-sgrid(0)) *
          dsgrid_inv - cell_idx_r
```

```
cell_r = (loc_r-sgrid(0)) *
          dsgrid_inv
cell_idx_r = INT(cell_r)
incell_r  = cell_r - AINT(cell_r)
```

Listing 7 Code change in `fields.f90` to reduce the number of type conversions. The left box shows the original version; the right box shows the improved version. One `integer` to `float` conversion has been replaced by the `AINT` function.

```
cell_idx_r = MAX(1, MIN(nequ_rm1,
              INT(cell_r)+1))
IF (loc_r < ger_min)
           cell_r = 0.0
IF (loc_r > ger_max)
           cell_r = cell_idx_r
incell_r = cell_idx_r - cell_r
```

```
cell_idx_r = INT(cell_r) + 1
incell_r = aint(cell_r + 1.0)
           - cell_r
IF (loc_r < ger_min) THEN
  cell_idx_r = 1
  incell_r = 1.0
ELSE IF (loc_r > ger_max) THEN
  cell_idx_r = nequ_rm1
  incell_r = 0.0
END IF
```

Listing 8 Code change in `equil.f90` to reduce the number of type conversions and branches. The left box shows the original version; the right box shows the improved version. One `integer` to `float` conversion has been replaced by the `AINT` function. The `MIN` and `MAX` functions have been removed by taking advantage of an already existing `IF` condition. The variable `cell_r` is not used after this code part.

5.10. *Conclusion*

The speedups for the various code improvements are shown in Table 7. The different improvements do not add up to each other. In order to understand this we would need to look more closely at the bottlenecks during the code execution. Table 7 shows that we achieved an 89 % speed-up in the targeted code paths. This was realized without any additional parallelization and will therefore benefit simulations on a very broad scale.

You can find an updated version of the roofline plot in Fig. 47. Almost all markers improve in both metrics, the arithmetic intensity as well as the number of FLOPS.

Performance metric	Ions per s	Electrons per s	Wall time of linear test case main loop
New gyro-points struct	4.2 %	2 %	2.1 %
b_equ_getfield prefetch	12.7 %	3.5 %	5.1 %
General code changes	18.2 %	8.3 %	7.6 %
No redundant computation	32.4 %	20.1 %	10 %
No redundant computation + hotspot part simd	82.7 %	25.9 %	19.3 %
All improvements	89 %	26.8 %	22.8 %

Table 7 Overview of the implemented improvements of the code. All values are comparisons to the baseline case. For the first two performance metrics, this refers to an increase, for the last metric it refers to a decrease, of the measured value.

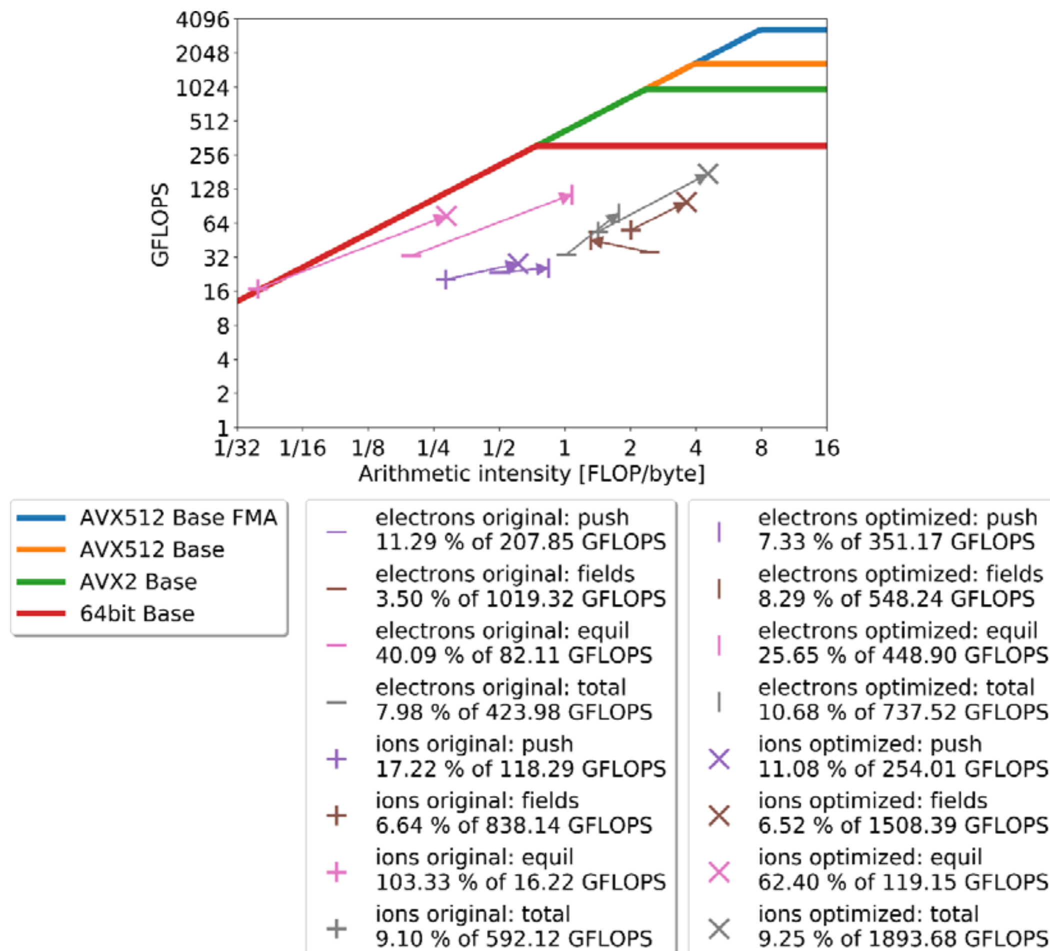

Fig. 47 Updated roofline plot for EUTERPE on Cobra, for different species and different segments of the particle-push loop. "Base" refers to the base frequency of the core. ITB is not used for this plot. We also give the percentage of the maximum performance as well as this maximum performance value for each species-segment combination, which is related to its specific arithmetic intensity value. The original values, as found in Fig. 35, are repeated in this plot and complemented with the results of the fully optimized code version. The arrows in the plot go from the value of the original code to the value of the optimized code.

5.11. *Summary*

The PICOPT2 project focused on the gyro-kinetic PIC (GK PIC) code EUTERPE. This code is widely used in the community, for example for stellarator simulations. The aim of this project was to improve the performance of EUTERPE, mainly via the exploitation of the vectorization capabilities of the Intel Skylake architecture. The project focused on the particle pusher part of the GK PIC algorithm in EUTERPE, making ions pushed per second (ions/s) the main benchmark to gauge improvements.

In order to tackle the task of improving the performance, we first needed to establish the theoretical performance characteristics of the targeted Intel Skylake architecture, especially in regards to the AVX512 instructions, and second we needed to investigate the performance characteristics for the targeted simulation code EUTERPE. After in-depth analysis of the EUTERPE code structure and a segmented performance analysis, involving measuring the FLOPS, memory behavior, cache characteristics and creating a roofline model of different code parts, we concluded that significant performance gains might be possible. Surprisingly, the overwhelming majority of the particle pusher's FLOPS and data requirements are spend doing field interpolation. Furthermore, the processor cache usage was especially egregious, which could be traced to details of the GK PIC algorithm.

After modelling these two aspects of the problem, a former HLST project with a similar problem description was revisited. In 2010 and 2011, HLST member Nicolaj Hammer tried to improve the performance of EUTERPE by adapting the particle pusher for increased vector computing usage. The project failed at achieving that goal. We analyzed the adaptions, understood why they did not help in improving the performance and resolved to not re-engineer N. Hammer's changes into the current code. Nonetheless, they supplied some insight into different tactics to write vectorized code.

With the help of the elaborate models and the strong foundation in understanding the EUTERPE particle pusher, a host of modifications was tested. After thorough benchmarking a subset of these adaptions were found to actually improve the performance. These adaptions were code restructuring (4.2 % more ions/s), software prefetching (12.7 % more ions/s), general code optimizations (18.2 % more ions/s), removing redundant computations (32.4 % more ions/s) and code vectorization (82.7 % more ions/s, the previous optimization is included in this number). Enabling all adaptions together, the EUTERPE code is now able to push 89 % more ions per second, while the wall time of the supplied test case decreased by 22.8 %. This is realized without any additional parallelization and will therefore benefit simulations on a very broad scale.

5.12. *Bibliography*

[1] Hatzky, R. (2018). *HLST Core Team Report 2018.* EUROfusion.

[2] Hatzky, R. (2010). *HLST Core Team Report 2010.* EUROfusion.

[3] Hatzky, R. (2011). *HLST Core Team Report 2011.* EUROfusion.

[4] Hatzky, R. (2017). *HLST Core Team Report 2017.* EUROfusion.

[5] https://github.com/RRZE-HPC/likwid/wiki

[6] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc, "When Prefetching Works, When It Doesn't, and Why", ACM Transactions on Architecture and Code Optimization, Vol. 9, No. 1, Article 2 (2012)

[7] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z

[8] Agner Fog. Technical University of Denmark, Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs

[9] Agner Fog. https://www.agner.org/optimize/blog/read.php?i=415

6. Report on HLST project LCTURB

6.1. *Introduction*

During the past decade, HPC systems have improved by two orders of magnitude in computational efficiency (FLOP/s) while the memory size (MS) and storage bandwidth (SB) of these systems have only improved by one order of magnitude. This is evident from Table 1, which lists the maximum FLOP/s, MS and SB of some exemplary HPC systems of the past decade. Such a trend suggests the need for reduced memory foot print and reduced data storage options for legacy codes to run on HPC systems of the future.

The plasma turbulence code GENE-3D is limited by memory and often the parallelization configuration is chosen based on memory bounds. The goal of the project LCTURB is to validate and implement lossy compression of data during the runtime of the code to keep its memory footprint low while ensuring reasonable accuracy.

Preliminary results already indicate the feasibility of at least one lossy compression library. Statistically similar results between turbulence simulations performed in single and double precision motivated a preliminary implementation of lossy compression on the distribution function array. This hints at the potential for reduced memory footprint and therefore a better utilization of performance of the cluster nodes.

Table 8 Attributes of supercomputers showing their performance (PF), memory size (MS), and storage bandwidth (SB) over the last decade [1].

Supercomputers	Year	Class	PF	MS	SB
Cray Jaguar	2008	1 Pflops	1.75 Pflops	360 TB	240 GB/s
Cray Blue Waters	2012	10 Pflops	13.3 Pflops	1.5 PB	1.1 TB/s
Cray CORI	2017	10 Pflops	20 Pflops	1.4 PB	1.7 TB/s
IBM Summit	2018	100 Pflops	200 Pflops	>10 PB	2.5 TB/s

6.2. *Performance of GENE-3D*

GENE-3D is a fork of the legacy code, Gene. It uses finite difference discretization along all the data dimensions and is suited for asymmetric geometries. The largest data array is a distribution function and is six-dimensional. From a production scale sample run (128 nodes, 40 cores each) on the COBRA supercomputing system at MPCDF, some of the relevant performance parameters are shown in Fig. 48. The memory bandwidth and performance per socket has a steady behavior through its runtime. The code has a memory footprint of up to 100 GB per node.

The algorithmic intensity of the code is estimated from the average memory bandwidth and performance to be 0.33 Flop/Byte per socket. The roofline plot of a socket of the COBRA cluster is shown in Fig. 49. The current performance of the GENE-3D code is annotated in the same plot. It seems that there is considerable room for improvement, given the fact that the code is a stencil code that uses finite difference discretization. The cache-miss ratio (last level cache) in Fig. 48 being very high suggests room for improvement using optimal blocks of data. The cache access can be improved and this could result in shifting the (orange) point in the roofline plot (Fig. 49) towards the machine's maximum performance.

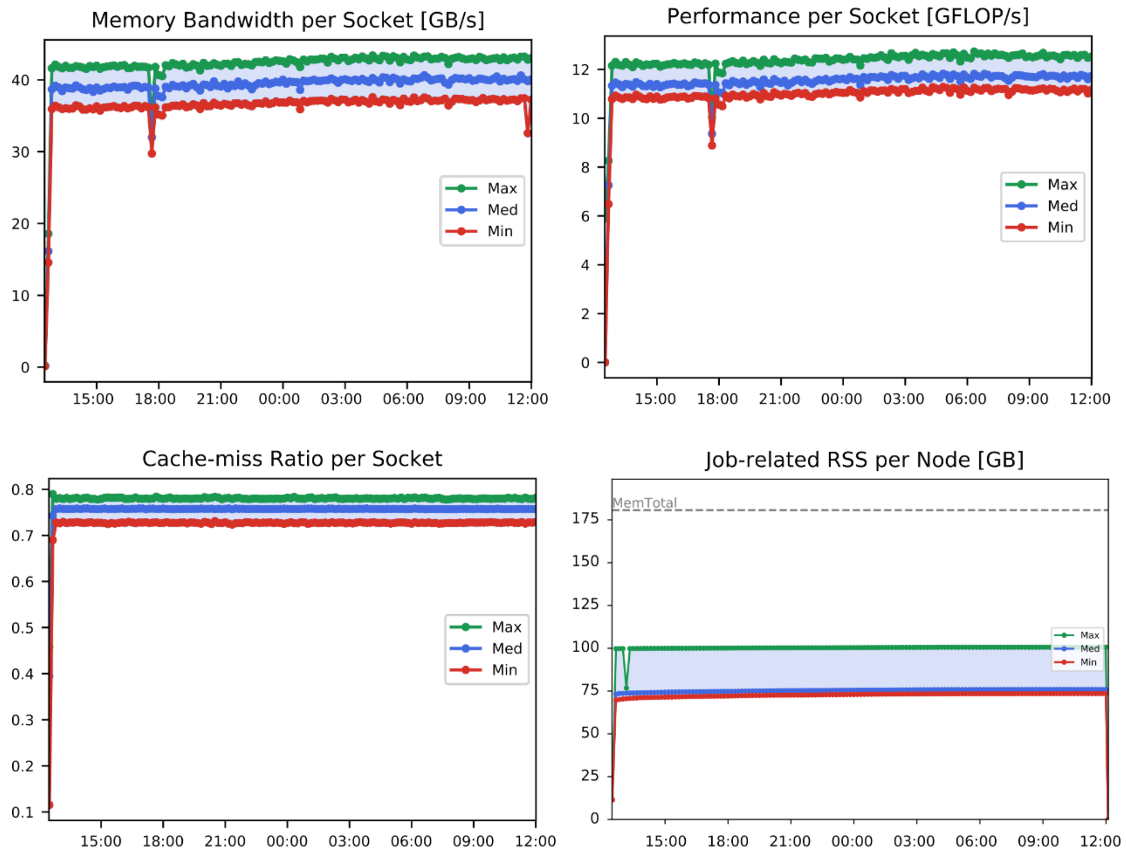

Fig. 48 Initial performance analysis of GENE-3D on Cobra cluster. The timelines are max, median and min respectively of all the samples (nodes/socket) for each timestamp per socket (or node). A timestamp is probed every ten minutes. RSS refers to the Resident-Set-Size occupancy of the user processes.

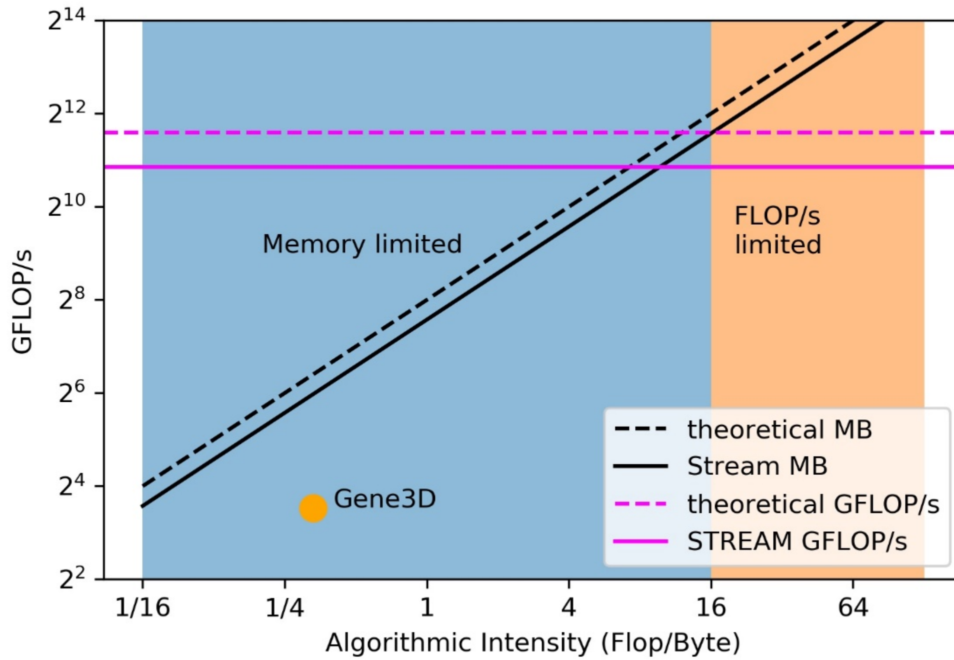

Fig. 49 Roofline plot of performance of Cobra node showing current performance of GENE-3D. MB refers to the maximum bandwidth.

6.3. *Data Compression*

As an initial candidate for compression of data, the ZFP library [2] was chosen and an early implementation of the compression and decompression between each time step was provided by the TSVV. We chose a simulation test case and conducted initial tests to check compression against loss of accuracy.

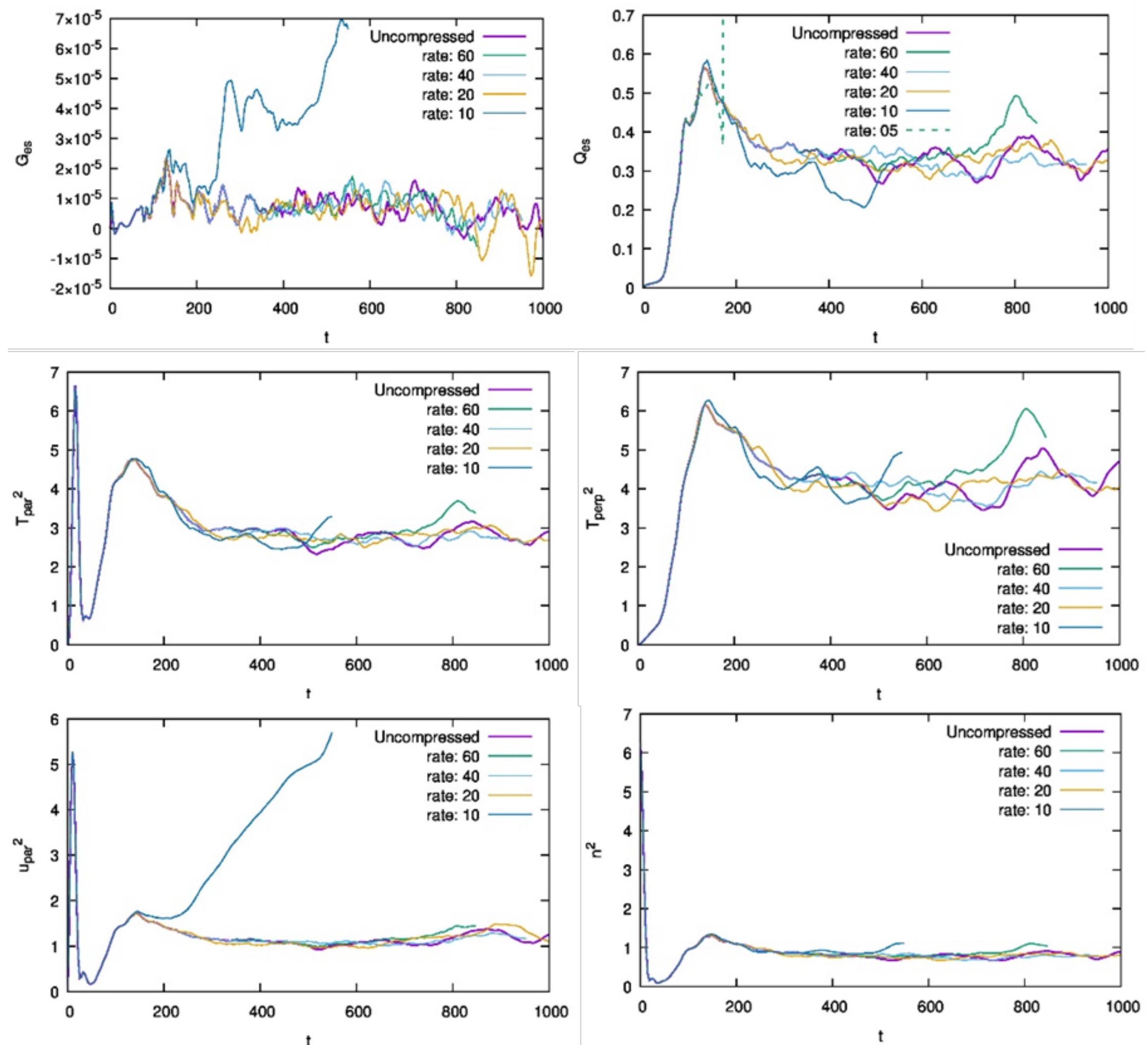

Fig. 50 Time evolution of observables for different compression ratios. Non-dimensional time is used along X-axis. The smaller the compression rate, smaller is the data size and vice versa.

The ZFP library uses a blocked data compression approach. It compresses regularly gridded data by transforming spatially correlated data to a basis that decorrelates it. This results in many near-zero coefficients that are compressed away. Among other similar discrete orthogonal block transforms such as the Haar wavelet transform (HWT), the slant transform (ST), the discrete cosine transform (DCT) and the high correlation transform (HCT), the approach used in ZFP achieves the maximum decorrelation efficiency and coding gain. For the purpose of this study, we employ an initial 'black box' application of this library. The block sizes are 4^d in size where d is the number of dimensions. GENE-3D uses a six-dimensional array. The first three dimensions are spatial dimensions and the next two corresponds to velocity parallel and perpendicular to the magnetic field directions. The sixth dimension is the number of species. We chose to use the ZFP library with four and three dimensions initially. Since we expect greater correlation between the spatial dimensions and since most operators operate on data per velocity dimension, it would make sense to consider compression and decompression of three-dimensional spatial blocks.

Fig. 51 shows the time evolution of different observables of the simulation for different compression rates (CR). Of these, the quantity called 'Q_{es}' is the most representative of the simulations success. This quantity refers to the total heat flux in the system. Clearly, even for a high compression rate of 5 % (only shown in the plot for Q_{es}), the initial linear part of the plots for any of the observables remains very similar to the uncompressed simulation. After this, the compression rate of 5 % results in a 'blow up' in the simulation. However, for up to 10 % compression the results look well within bounds similar to that of the uncompressed simulation. The deviation from this reference simulation occurs increasingly earlier for higher compression rates, as expected. In order to check the effectiveness of the compression rate separately, we show the mean and standard deviation of Q_{es} in the non-linear part of the curve for different compression rates in Fig. 51.

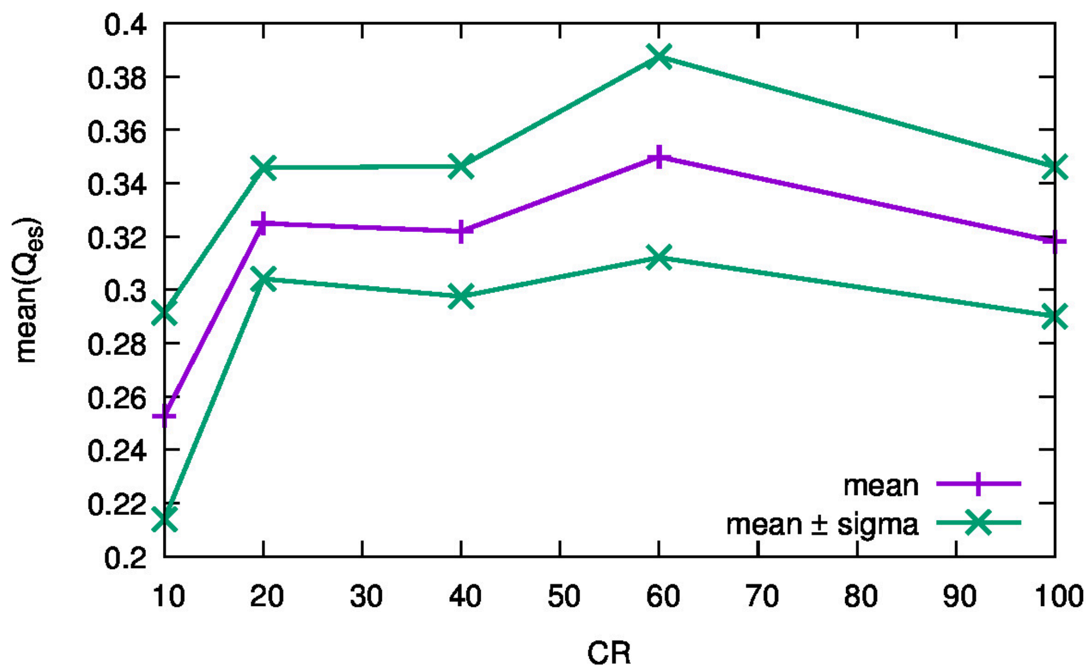

Fig. 51 Mean and standard deviation of the total heat flux Q_{es} for different compression rates (CR).

The mean and standard deviation for each of the compression rates until about 20 % is very similar to the uncompressed value of 100 %. Only for very high compression of 10 % the mean and the standard deviation deviates significantly. Interestingly, we see no steady feature increase with compression rate where the values are close to the uncompressed case. This gives us the confidence that compression rates of up to 20 % does not affect the accuracy of the simulation significantly.

6.4. *Outlook and conclusions*

Initial results give us confidence to proceed with the ZFP library for compression of data for GENE-3D. Currently, the compression-decompression cycle is performed once during the time loop. In the immediate future, we aim to use only compressed data even within the loops. Thus, each matrix-free operation within the time loop would decompress the data 'block by block' and perform computations. This would achieve a reduced memory foot-print and will enable better use of the nodes' performance. The granularity with which the data is compressed as blocks and then decompressed will also determine the memory footprint.

However, the code uses a matrix solver to solve a Poisson equation (part of Maxwell's equations). The use of a blocked compression and decompression approach would be unfeasible for the linear solver that is implemented in an external library. Hence, overall

memory efficiency can only be achieved when the linear system is solved using a matrix-free approach. It is one of the tasks of the Principal Investigator (PI) to implement a matrix-free solver in the near future, which would enable the reduction of the memory footprint of GENE-3D.

In addition to ZFP, other compression libraries with block granularity will be implemented to compare the accuracy for same the compression rate. For example, the SZ [3] library seems to be a good candidate. Other matrix compression libraries which may be more accurate for the system but which compress the entire data structure in one block will not be feasible for this effort. Thus, a comparison across feasible compression libraries will also be performed.

6.5. *Bibliography*

[1] F. Cappello et al., “Use cases of lossy compression for floating-point data in scientific data sets,” The International Journal of High Performance Computing Applications, vol. 33, no. 6, pp. 1201–1220, Nov. 2019.

[2] X. Liang et al., “Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets,” in 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 438–447.

[3] S. Di and F. Cappello, “Fast Error-Bounded Lossy HPC Data Compression with SZ,” in 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 730–739.

7. Final report on HLST project OPT-DG

7.1. *Introduction*

The high order 3D Discontinuous Galerkin code `Fluxo` [1, 2] has been developed to solve the 3D full MHD equations, including nonlinear and resistive terms. It has an explicit time integration and uses unstructured hexahedral meshes. The code aims to improve the scalability of 3D non-linear MHD simulations of fusion plasmas. `Fluxo` is pure-MPI parallelized and production runs of $\mathcal{O}(10,000)$ MPI ranks are possible. `Fluxo` exhibits an ideal weak and strong scaling on current NUMA architectures. This is related to the Discontinuous Galerkin scheme with explicit time integration, having dense local operations, only direct neighbor communication and a low memory consumption. During the HLST assessment project MPI3-DG (2018) [3], the MPI parallelization of `Fluxo`, which employs point-to-point non-blocking MPI communication, was analyzed in detail on the Marconi-Skylake architecture, and the communication/computation overlap was confirmed. In addition, a two-level communication infrastructure separating intra- and inter-node communication was implemented. This led to a reduced communication cost, and potential improvements for increasing the overlap were identified [3].

Current supercomputing hardware is based on multi-core NUMA architectures with increasing sizes of vector execution units, which enable executing single instructions on multiple data (SIMD). Therefore, in order to generate code which runs efficiently on this kind of hardware, ensuring a good level of vectorization becomes fundamental. This is the main subject of the current project, where the single-core performance is analysed on the Intel Skylake architecture in terms of vectorization with AVX-512 instruction set. This allows to identify hotspots and propose changes that need to be tested and, if time allows, implemented back in `Fluxo`. Finally, it is also noteworthy that changes made in the code in order to increase the degree of vectorization will in principle facilitate future porting to multi-core architectures, like GPUs, which use the single instruction multiple threads (SIMT) execution model.

7.2. *Profiling & performance analysis of `Fluxo`*

7.2.1. Single-core profiling

The first task performed in the project concerns the profiling of the current implementation of `Fluxo`, to identify the computational hotspots, which are the best candidates for the planned optimisations. This task, also performed during the previous HLST MPI3-DG project [3], had to be repeated because a different implementation of `Fluxo` is being targeted now. Namely, one that uses a recently implemented entropy-stable discretization [4]. This implementation increases the robustness of the simulations, but does so at the expense of an increase in the cost of the volume terms. The profiling performed with Intel VTune Amplifier on `Fluxo` revealed the subroutines (`VolInt_SplitForm_eqn` and its callees) responsible for the calculation of these terms to be the clear hotspot. Therefore, this is where the vectorization efforts will focus for the remainder of the work.

7.2.2. Vectorization analysis

There are three main approaches for code vectorization. The first, which we shall call *auto-vectorization*, is done automatically by the compiler. This is of course the highest-level approach and also, obviously, the easiest. However, it might not be enough in terms of expected performance for the program at hand, in which case we need to resort to the second approach, namely *explicit vectorization*. This solution supplements the compiler auto-vectorization via the usage of OpenMP SIMD directives, available since the OpenMP v4.0 standard, to decorate the source-code explicitly. They force the compiler to vectorize the decorated loops, even if it would not do so based on its

auto-vectorization heuristics, for instance due to complicated data-dependencies. This means that it is the responsibility of the developer to ensure that the forced vectorizations are safe. The *explicit vectorization* constitutes the main approach being explored in the current work. Finally, if one wishes to push the limits of the vectorization capabilities, the third and lowest-level approach needs to be considered, namely the *manual vectorization*. It uses the so-called *SIMD intrisics*, which resemble assembly language, but written directly inside the C/Fortran program source-code. Being much more involved and implying more source-code changes, which further decrease the overall maintainability of the program, this approach lies clearly beyond the scope of this project, and therefore shall not be pursuit here. Instead, the bulk of the work shall be devoted to the *explicit vectorization* approach.

In practice, with the profiling of `Fluxo` available (recall Subsec. 7.2), the next step comprises checking to what degree the compiler auto-vectorization is able to address the cost-intensive loops in `Fluxo`. This can be done using the vectorization reports generated at compile time. In our case, using the Intel Fortran compiler, this was done with the following compilation flags

```
-qopt-report=5 -qopt-report-phase=vec,
```

which generate an ASCII report file (with a `.optrpt` extension) for each source-code file in the program. Complementary, we used also Intel Advisor that provides the same information in a graphical manner, which can be easier to interpret, at least at an early stage of familiarisation with the code under analysis (`Fluxo`). However, this tool can further extend the analysis by measuring additional metrics for loops or functions in the code, like for instance FLOPs/s and the memory access, from which it can automatically generate the corresponding Roofline model [5]. It also allows to inspect memory access patterns, which can give valuable insight, even though one should keep in mind that such detailed analysis can take a substantial amount of time to measure (several hours in comparison with the wall-clock time of less than a minute of the test case used for the measurements). From this preliminary assessment, a few loops could be decorated with OpenMP SIMD extensions in the subroutines `VolInt_SplitForm_eqn` and `EntropyAndKinEnergyConservingFluxVec_FloGor`, which yielded an overall single-core speedup of the whole `Fluxo` test simulation of about 12 %. This result shows that there is indeed potential for improving the vectorization in this code.

Because the idea of the project is to have a clear picture about the full vectorization potential of `Fluxo`, a finer-grained analysis is in order. Even though the Intel Advisor provides lots of useful information with relatively little effort, it is by design bound to assess each loop in the code as a whole. If one intends to go below this level, like is the case in `Fluxo` due to its large loops containing several computationally heavy operations, another approach is required. In particular, the plan is to use the Likwid toolsuite [6] to measure processor performance counters and build the Roofline model for different operations inside each loop, as deemed necessary. This flexibility should allow for a better assessment of the potential performance achievable gains for each of those operations, revealing the performance upper-bounds, which guide the efforts to be put in the respective optimisation. Such an analysis if often called *performance engineering* in the literature [7].

7.3. *Reduced* `Fluxo`

As explained before, a detailed analysis of `Fluxo` using the Likwid performance suite is planned. However, as shown by the profiling analysis (Subsec. 7.2), only a subset of subroutines in `Fluxo`, which represent the bulk of the computational work, needs to be optimised. Therefore, it was decided to strip down the full version of the code into a reduced implementation which exclusively contains these subroutines, called from a simplified main program that uses only the additional modules that are strictly

necessary. This makes the compilation- and execution-time of the simplified code considerably faster than the original full `Fluxo` code due to the elimination of the unnecessary overhead of subroutines that require no performance optimisation. Also, it makes the source code much simpler to read and the number of variables involved much smaller, greatly facilitating any source code changes, while minimising the risk of these breaking the remaining code execution. Therefore, the performance engineering analysis described before shall be applied to the reduced `Fluxo`.

7.4. *Simplified code to assess explicit SIMD vectorization*

`Fluxo` stores it data using 5D arrays, whose components represent the number of variables, the polynomial degree of the element basis in each of the three spatial dimensions and the number of elements used, respectively. Because of this relatively complex data layout, it is convenient to assess how the explicit SIMD vectorization affects the performance in such cases. To this end, a suite of small programs was developed based on the examples provided by the Intel guidelines for Fortran explicit vectorization [8]. These examples were then modified and tailored to mimic the data layout of `Fluxo`, resulting in a number of different tests with an increasing degree of complexity. Some tests include data transposition operations to exchange the order of the array indexes. The idea here is to assess what is the optimal data ordering for executing vectorized operations. All the developed tests include two versions of the computation intensive subroutine, one explicitly decorated with OpenMP SIMD directives and other without. This allows to directly inspect, for each test, whether or not the explicit vectorization brings advantages compared to the default auto-vectoriation made by the compiler. The default compilation flags used include

`-vecabi=cmdtarget -qopt-zmm-usage=high -align array64byte`.

Four different executables are then generated for each test. The first further uses the compiler flag `-qopenmp-simd` to produce the *SIMD* version of code, which forces the compiler to take into account the OpenMP SIMD directives included in the source code. The second executable further adds the compiler flag `-fno-inline`, which explicit inhibits any inlining of subroutine/function calls inside the loops. This corresponds to the *SIMD_NOIN* version of code, which allows assessing the impact of inlining. Finally, the third and fourth executables are generated similarly, but inhibiting instead the SIMD directives by replacing that flag with `-qno-openmp-simd`, with and without in-lining, respectively, to generate the *NOSIMD* and *NOSIMD_NOIN* versions of the code. Running all four versions for each test program (i.e. data layout) provides valuable guidance for the appropriate choices to be made in `Fluxo` before actually attempting to modify this code, or even its reduced version (see Sec. 7.3).

7.5. *Summary and outlook*

The main goal of the the HLST-OPT-DG project is to assess the current degree of auto-vectoriation in `Fluxo` and then devise and implement a strategy to improve it using explicit vectoriation techniques that invoke OpenMP SIMD compiler extensions. The idea is to use the AVX-512 instruction set together with the Intel compiler to improve the code's single-core performance. The first steps, already performed, include profiling `Fluxo` to expose the computationally heavy routines, to which the performance optimisations will be applied. A first auto-vectoriation analysis was also conducted, using Intel Advisor, from which resulted some speedup after adding just a couple of SIMD directives. This proved the potential for more extensive explicit vectorisation in `Fluxo`, which is planned for the remaining time of the project. However, to facilitate the detailed assessment of the performance improvements available, a reduced version of `Fluxo` was introduced. It includes only the necessary ingredients, namely the hotspot routines, and the necessary variables, leaving out all remaining

parts of the work. This not only reduces the compile- and run-time of the test-cases, but also minimises the effort required when changing the code to implement the explicit vectoriation, or even code refactoring, if that turns out to be required. This task is currently ongoing and constitutes the bulk of the work to be done from now until the project end. Finally, a suite of very simple tests based on Intel's vectoriation documentation webpages [8] was also developed, with the goal of quickly allowing to inspect the effect of explicit vectorization on loops whose data layout in memory resembles that of `Fluxo`. These tests already provided very insightful information to guide our explicit vectorization strategy in this code, and remain as a tool that can be easily further extended, shall more insight be required.

7.6. ***References***

[1] F. Hindenlang, G. Gassner, C. Altmann, A. Beck, M. Staudenmaier and C.-D. Munz, *Explicit Discontinuous Galerkin methods for unsteady problems*, Computers & Fluids **61** (2012) 86

[2] F. Hindenlang, *Mesh Curving Techniques for High Order Parallel Simulations on Unstructured Meshes*, PhD thesis (2014), Universität Stuttgart

[3] T. Ribeiro, *Final report on HLST project MPI3-DG* (2018)

[4] M. Bohm, A. Winters, G. Gassner, D. Derigs, F. Hindenlang and J. Saur, *An entropy stable nodal discontinuous Galerkin method for the resistive MHD equations. Part I: Theory and Numerical Verification* , Journal of Computational Physics (2018)

[5] S. Williams, A. Waterman and D. Patterson, *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*, Communications of the Association for Computing Machinery (2009)

[6] J. Treibig, G. Hager and G. Wellein. *Likwid: A lightweight performance-oriented tool suite for x86 multicore environments.* In Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, (2010). (https://hpc.fau.de/research/tools/likwid/)

[7] G. Hager and G. Wellein *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Inc. (2010)

[8] Martyn Corden, P. Barbara, *Explicit Vector Programming in Fortran* (2018) (https://software.intel.com/en-us/articles/explicit-vector-programming-in-fortran)

8. Final report on HLST project REFMUL3+

8.1. *Introduction*

The finite-difference time-domain (FDTD) method is one of the most popular numerical techniques used to simulate reflectometry. It offers a comprehensive description of the plasma phenomena. However, this method requires a fine spatial grid discretization to keep the error to a minimum, which in turns implies a high-resolution time discretization to comply with the CFL numerical stability condition. Simulations in three-dimensions (3D) are therefore very demanding computationally, in terms of both floating-point operations and memory resources. They become only possible if the problem can be efficiently distributed over large numbers of resources. The REFMUL3 code was developed in 2016 within a collaboration between the Instituto de Plasmas e Fusão Nuclear (IPFN-IST) in Lisbon and the HLST, precisely to meet this goal [1]. REFMUL3 is a 3D full-wave code using the Yee scheme [2] with full polarisation, which simultaneously copes with o- and x-modes, supports a general external magnetic field and a dynamic plasma. REFMUL3 employs a hybrid MPI/OpenMP parallelisation using an explicit 3D domain decomposition, which yields very good scaling properties up to a few thousand cores. This has opened the door to the simulation of much larger domains (grid-counts) which, as expected, exposed new challenges regarding data input/output (I/O) operations. This project aims at addressing those challenges by improving the I/O capabilities of REFMUL3.

8.2. *Parallelisation of REFMUL3*

The domain decomposition implemented in REFMUL3 relies on `MPI_Dims_create` to find automatically a suitable distribution of the resources (cores) over its three spatial dimensions. A Cartesian virtual topology is then created by invoking `MPI_Cart_create` and the process ranks (MPI tasks) are mapped to a 3D coordinate system using `MPI_Cart_coords`. Additionally, a thread-based parallelisation (`#pragma omp parallel for`) can be applied to the slowest varying index (outer loop), which in REFMUL3 corresponds to the z-direction. This choice optimises memory access speed (cache re-usage) and takes into account that the numerical kernel is largely symmetric, in terms of the numerical stencil, in all spatial directions.

The MPI parallelisation of REFMUL3 does not allocate the global computational domain. Instead, it decomposes it into subdomains, each belonging to a different MPI task that locally allocates only the corresponding memory. Each subdomain contains one extra cell per face in all three dimensions, called a ghost-cell, where the values of the corresponding faces of the neighbouring subdomains are stored. This is a necessary and sufficient condition to enforce continuity of the distributed solution across the subdomains for the algorithm under consideration. However, this rule is relaxed for the subdomains that include faces that are part of the global domain boundaries. There, the boundary conditions replace the neighbours and no ghost-cells are needed.

The subdomains are defined using the concept of first and last grid-node global indexes. As the name suggests, these specify the global index values of the first and last grid-nodes of each subdomain in each dimension (x,y,z). They are calculated using the global domain size and the number of MPI ranks used, together with the MPI rank of the task to which the subdomain belongs [1]. The size of each subdomain in each dimension is calculated from the difference between the corresponding last and first grid-node indexes [1]. This technique is quite flexible as it allows the subdomain sizes to differ between MPI tasks, lifting the common constraint that the grid-count must be a multiple of the number of tasks used. This is very welcome when dealing with staggered grids, which by definition differ in size by a grid-node, as is the case with REFMUL3. Moreover, this choice for the domain decomposition adds no extra complication to the implementation of the parallel I/O subroutines, which require

knowing the spatial positions of the subdomains on the global domain. The reason being that any task can easily calculate the set of numbers characterising any other subdomain (size, first and last grid-node indexes) in the simulation [1].

8.3. *Parallel I/O*

8.3.1. HDF5 library: single-file vs. multi-file

As already mentioned in the introductory Sec. 8.1, enhancing the I/O capabilities in REFMUL3 is precisely the motivation for the HLST-REFMUL3+ project. By the end of the HLST-REFMLIO (2018) project [3], the code was equipped with two methods to perform I/O operations, which could be chosen at compile time. In the first, each MPI task independently accesses a different (serial) HDF5 [4] file on disk. Each of these files is a container for the requested data (physical quantity) on the subdomain belonging to the MPI task that accesses it. The file labelling includes the corresponding MPI Cartesian virtual topology coordinates (MPI ranks in each spatial direction) in the filenames. The second method uses the parallel version of the HDF5 library (pHDF5). In this case, a single HDF5 file containing the global domain is accessed collectively by all MPI tasks to read or write the requested dataset. This means that each MPI task must access the spatial region of the global domain contained in the file that corresponds to its subdomain.

The difference between both methods in terms of data layout can be visualised in Fig. 52, which shows the *z*-component of the electric field (E_z) data for a typical REFMUL3 simulation domain. The lower part corresponds to the data stored using a different HDF5 file per subdomain (multi-file method), as can be inferred from the different colours in the figure. The top part of the plot corresponds to the I/O solution that stores all subdomains in a single HDF5 file (single-file method). As expected, a perfect match between the surfaces in the upper and lower parts of the plot is obtained, since both methods store the same data.

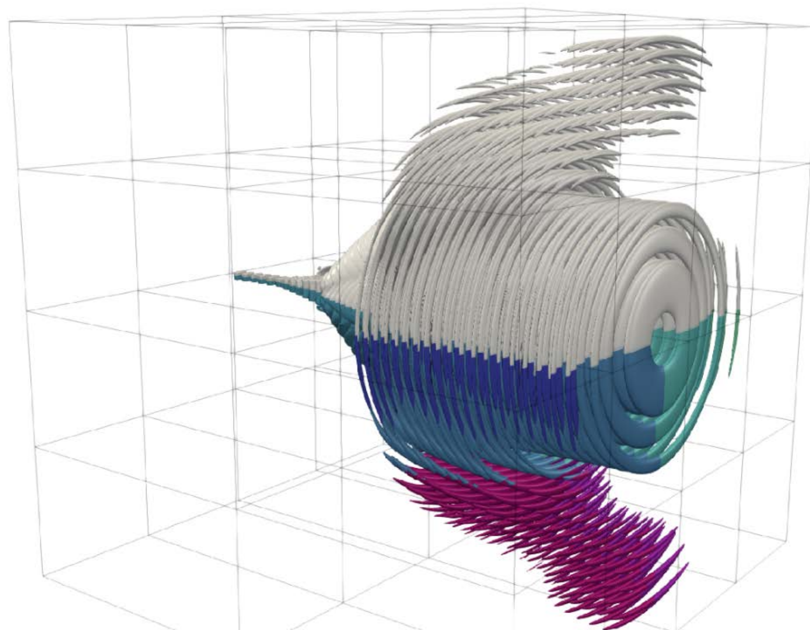

Fig. 52 Iso-surfaces of the *z*-component of the electric field (E_z) after 2000 iterations for a 3D REFMUL3 simulation of a 720x700x700 domain decomposed over 32 MPI tasks. The upper half of the plot (grey) shows the result from the single-file method (pHDF5). The lower half corresponds to the multi-file solution, with the subdomains stored in independent HDF5 files represented by different colours.

Meanwhile, there were some additional refinements made during the current project regarding both I/O methods available. While originally REFMUL3 stored different quantities (e.g. electromagnetic fields) in different sets of HDF5 files, the code has

been changed to produce only three sets of HDF5 files. One to store the static quantities, like the plasma density and the spatial structures of wave-guides, antennas or mirrors. The other two to store the time-evolving electromagnetic fields at different instants and/or acquisition rates. Inside these files, the different quantities are stored as different HDF5 *dataspaces*. This change decreases the overall number of files produced during a simulation, which is especially beneficial for the multi-file method, particularly when high numbers of MPI tasks are used. Along the same lines, a variant of the multi-file method was also introduced in order to include the temporal dimension as the fourth dimension of the dataset stored in the files. Hence, instead of having a different set of HDF5 files per temporal snapshots (iteration), they are combined into the same set of HDF5 files.

8.3.2. HDF5 performance assessment and tuning

In terms of practical usability, both multi-file and single-file I/O methods have advantages and disadvantages, in a fashion that renders them complementary. The single-file solution is independent of the domain decomposition, which is very convenient if one thinks of check-pointing to create simulation restart files. It is also in general easier to handle the data stored on disk, since a single file contains the global simulation domain. Conversely, the multi-file solution produces a much larger number of files that depends on the choice of domain decomposition. However, it has also a big advantage in that it performs much better when a considerable number of MPI tasks (more than 64) is used, showing some degree of scalability with the amount of resources used. This can be seen in Fig. 53, which compares the strong scaling of both I/O methods for a representative REFMUL3 problem size.

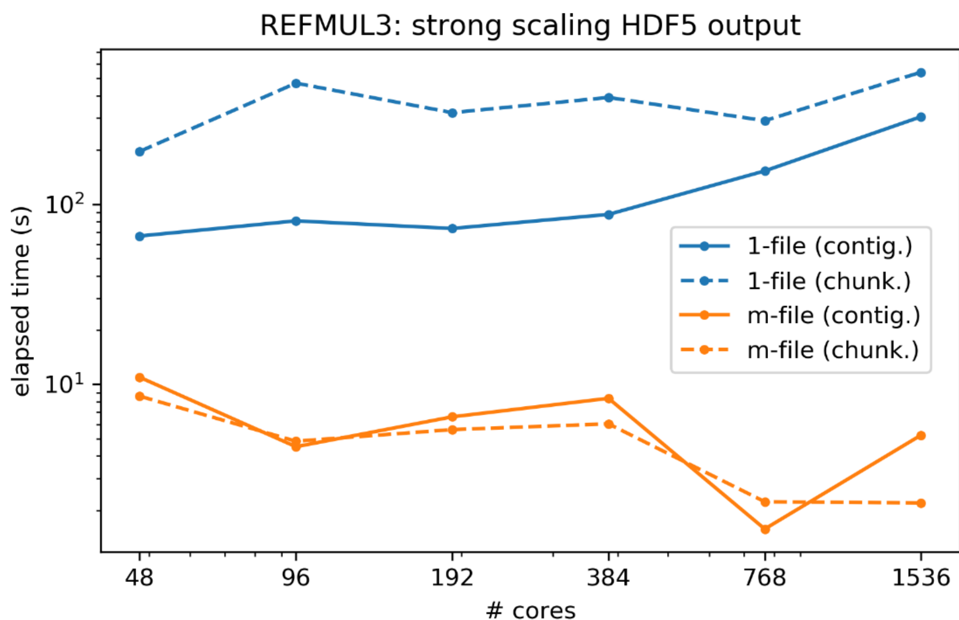

Fig. 53 Strong scaling of the I/O operations on a pure MPI REFMUL3 simulation using a 720x700x700 (*x*,*y*,*z*)-grid decomposed in 3D. The single file solution accessed concurrently by all MPI tasks via the parallel HDF5 library is shown in yellow. The blue curve shows the multi-file case, where each MPI task stores its data in a different (serial) HDF5 file. The measurements were made on the Skylake nodes of Marconi.

At this point it suffices to note that the blue curves (single-file method) have always higher values than the yellow counterparts (multi-file method), without elaborating on the differences between solid and dashed lines. This behaviour is not unexpected. On one hand, parallel file systems, like the IBM Spectrum Scale (formerly known as GPFS) installed on Marconi, require concurrent disk I/O access to achieve peak bandwidth, which nicely fits the multi-file solution. On the other hand, achieving similar performance using a parallel I/O library, like HDF5, is much harder. Firstly, it depends

on the particular installation setup used for that library, which is not the responsibility of the user but rather of the HPC administrators. Secondly, it depends on the degree of optimisation made by the user when calling the I/O library API from its code. The latter is not in general a trivial task to achieve. It involves a significant effort in parameter tuning within the parallel I/O library, followed by measurements to assess their impact on I/O performance, which consume HPC resources. Knowledge about the parallel file system configuration parameters (e.g. stripe size and count) is also of key importance to achieve good performance.

Within the HDF5 library, the performance optimisation is done via the concept of *data chunking*, which is illustrated in Fig. 54 for the 3D case. By default the HDF5 library writes the data contiguously along the fast-varying index (column-index in our example). While this provides an optimal data access in that dimension, it is quite inefficient for accessing data along the remaining dimensions. Such data accesses are discontiguous and require a lot of seek operations within the file. Therefore, for a general access pattern, it can be advantageous to group the data into *chunks*, within which the data is accessed contiguously along the fast-varying index, as depicted in the right side of Fig. 54.

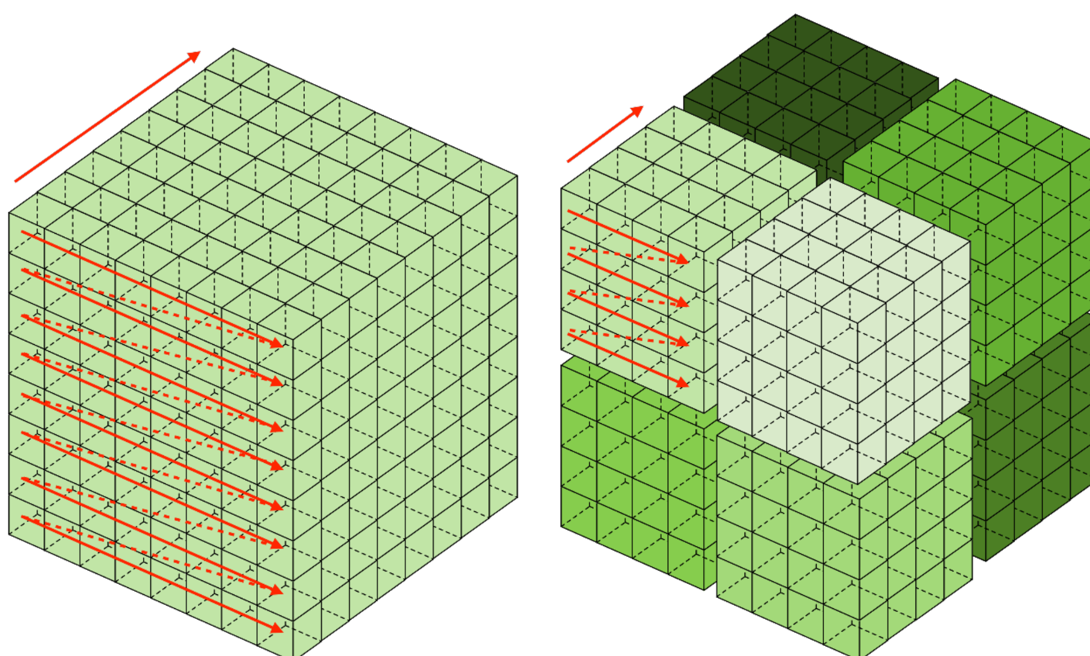

Fig. 54 Illustration of HDF5 I/O data chunked access (right) compared to contiguous access (left).

The HDF5 library requires the chunk size to be constant for all MPI tasks. However, it is not required that the global domain stored in an HDF5 file (*dataset*) is covered by an integral number of chunks. This is what enables chunking to be used in REFMUL3, whose domain decomposition allows subdomains with slightly different sizes (recall Sec. 8.2). The current implementation calculates the chunk size in each of REFMUL3's three spatial dimensions as an integer fraction of the largest subdomain in that direction. This means that the chunked global domain stored in the HDF5 file (*dataset*) may not be an integral number of chunks, i.e., the chunks cover the whole dataset, but some overhead space (smaller than the chunk size in each dimension) might be allocated within the file by the HDF5 library. This apparent drawback is the price to pay for the added flexibility, which is justified by the potential gains in performance that can result from chunking, as discussed before. Moreover, the file storage overhead can be mitigated by using data compression techniques from within the HDF5 library, which is anyway only possible if the data is stored using a chunked layout. Currently the compression is not activated in the HDF5 I/O subroutines of REFMUL3, but this can in principle be easily changed in the future.

The dashed lines in Fig. 53 show the results obtained with the same chunked data layout for both single-file (blue) and multi-file (yellow) methods. In particular, the chunk size used along each dimension $\alpha \in (x,y,z)$ is given by the half of the largest grid-count (L_α) of all N_α subdomains in that direction, namely,

$$L_\alpha = \frac{\max_n[l_\alpha(n)] - 1}{2} + 1 \qquad \& \qquad n \in [0, N_\alpha - 1]$$

where $l_\alpha(n)$ represents the grid-count of the subdomain with the (Cartesian) MPI rank n in the direction α. This implies that the chunk sizes along each dimension depend on the domain decomposition, i.e., on how many cores are allocated to that dimension. Using examples from Fig. 53, the chunk sizes in x,y,z for files produced on 96 cores is (91,89,60), whereas on 768 cores is (46,45,31), which reflects the fact that the latter uses double the number of cores per dimension compared to the former. The comparison to the solid lines (contiguous data layout) reveals that the choice of chunking layout used has little impact on the multi-file method. This is the expected behaviour because this method stores each subdomain in a different file, so that the chunked volumes inside each file are just a factor of 2^3 smaller than the whole subdomain volume. In other words, each chunked layout file contains only 8 chunked sub-volumes, which happens to be the case represented in Fig. 54. On the other hand, using the same chunking data layout on the single file method clearly impacts its performance. The reason being that, in this case, the same chunking volumes are much smaller relative to the dataset stored in the file, which now corresponds to the global domain volume. However, contrary to our initial hopes, a performance degradation is observed for the chunked data storage layout (dashed blue lines) compared to the contiguous counterpart (solid blue lines). Note that this does not contradict our argumentation in favour of the former because the I/O performance greatly depends on the choice of the chunking sizes. Therefore, the current result is not a show stopper. It simply means that more effort has to be put in searching for chunking size values that would improve the overall I/O performance for the REFMUL3 case. The way the chunking has been implemented in the code should allow this to be easily done. Had we had more time available within the current project, we would do it ourselves. Since that is not the case, such parameter scan has to be left for Filipe da Silva to perform in the future. One related topic refers to the concept of *chunk caching*, which according to the HDF5 documentation [5], can also impact the I/O performance. Also due to lack of time, this topic could not be explored, but its potential relevance suffices to recommend investigating it in the future.

To finalise the current session, a few words on the currently available I/O methods in REFMUL3 are in order. They are the multi-file method, the multi-file with time as fourth dimension in the dataset and the single-file method. Because of their complementary characteristics, it was decided together with Filipe da Silva to keep them all available at compile time via the `make` options `HDF5=1`, `H5T4D=1` and `PHDF5=1`, respectively. They activate specific preprocessor variables that are used to define a macro that chooses which I/O set of functions to use, respectively. This can be seen as an overloading of the I/O functions in REFMUL3. It provides maximal flexibility since it allows to make the decision on a case-by-case basis depending on the characteristics of the I/O operations at hand, as well as on the characteristics of the parallel file system being used. Another possibility that became clear during the course of the project refers to usage of both the multi-file and single-file methods simultaneously in a simulation, but at different places in the code. For instance, the time independent quantities, which require disk access only once during the initialisation phase of any simulation, are good candidates to always use the single-file method. They include the wave-guide and antenna structures that are more conveniently stored in a domain-decomposition-free layout, to allow them to be directly re-used in subsequent simulations using different numbers of resources. This has not been implemented in the final version of REFMUL3 handed over to Filipe da Silva. However, to do so is very simple. Namely, for those quantities, simply replace the I/O function calls done via the macro definition explained before with the explicit I/O function corresponding to the single-file case (`PHDF5=1`).

Let's consider as an example the overloaded function call `escrvCubeH5star`. This function is defined as `escrvCubeH5`, `escrvCubeH5t4D` or `escrvCubePH5` when REFMUL3 is compiled with the `make` option `HDF5=1`, `H5T4D=1` or `PHDF5=1`, respectively. To force the code to always use the single-file I/O method for the antennae description (`zFrm`), independently of the compilation options used, one simply replaces explicitly the existing calls to `escrvCubeH5star` for that quantity with `escrvCubePH5`.

8.3.3. Hybrid simulations: impact on I/O

So far we have emphasised the performance advantages of the multi-file implementation in REFULM3 over the current single-file counterpart, for which the HDF5 chunking is not yet providing the desired results, for moderately large numbers of MPI tasks (up to a few thousands). For the sake of completeness, it is necessary to discuss also the cases which use tens of thousands or more cores. Such scenarios become realistic when much bigger domain sizes are considered. Then, the pressure put on the file system by the multi-file method necessarily builds up due to the need to handle very large numbers of files being accessed concurrently, which will ultimately hinder the I/O performance. A mechanism available to ameliorate this issue refers to the usage of the MPI/OpenMP parallelisation capabilities of REFMUL3. It reduces the total number of MPI tasks employed in a simulation by the number of threads spawned. In the limiting case, this factor can go up to the number of cores per node, which on Marconi SKL partition is 48. The good scalability of the code when running with many threads per MPI task [1] supports this recommendation. From the I/O point of view, it should be considered whenever the amount of files produced by the pure MPI version of the code is so large that it breaks the scalability shown by the yellow lines in Fig. 53.

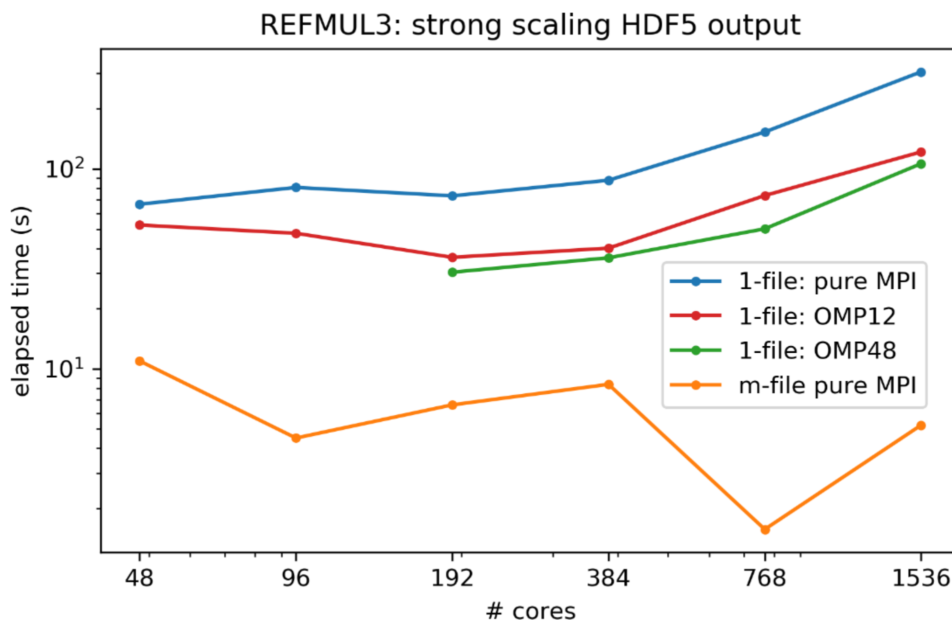

Fig. 55 Same strong scaling of the I/O operations shown in Fig. 53 for the contiguous HDF5 data layout on a pure MPI (solid blue and yellow lines). The lines for the single-file method with contiguous data layout when running REFMUL3 in hybrid mode are also show in red and green for 12 and 48 OpenMP threads per MPI task, respectively.

Even though the previous argumentation applies to the multi-file I/O method, the single-file counterpart also benefits from running REFMUL3 in hybrid mode. From the solid blue line in Fig. 53, it is clear that, as the amount of concurrency in accessing the (single) HDF5 file increases with the number of MPI tasks (cores), the scalability breaks down. If one uses several OpenMP threads per MPI task, the total amount of MPI tasks decreases for the same number of requested cores, and so does the degree of concurrency in the file access. Therefore, running in hybrid mode should bring I/O

performance advantages for the single-file I/O method, even when moderate numbers of resources are used, which is precisely what is shown in Fig. 55.

8.3.4. Check-point and restart-file

The milestone of having check-points in REFMUL3 refers to the storage in a restart-file of the necessary information about the state of a simulation at a given iteration. The ability to read this information allows to restart the simulation from that iteration and continue its evolution through a subsequent run. The current section describes the implementation and usage of the corresponding infrastructure, which was implemented in REFMUL3.

REFMUL3's restart-file must contain the full 3D electromagnetic fields and current density stored at the desired iteration. The whole impulse calibration time series up to that same iteration is also required, as is the iteration number corresponding to the restart state and to the next scheduled check-point. By default, the check-points are switched off in REFMUL3, so in order to use them, the first thing to do is to tell the code that check-points are required. This is signalled via a new command line input parameter: `--wrtrst=1`. Moreover, the check-points can be made periodically during a run, to ensure that, in case of failure, the simulation needs only to be restarted from the last successful check-point. This required introducing two additional command line input parameters to control the iteration of the first check-point (`--wrtrsti`) and the number of iterations between subsequent check-points (`--dwrtrst`), respectively. Alternatively, check-points can be used to split a long run, which does not fit within the maximum wall-clock time available in the computational queue, into a chained set of shorter runs, each fulfilling that condition, whose execution reaches the final desired iteration. This scenario required yet another input parameter to specify the iteration that triggers a check-point followed by a clean exit of the code (`--pitstop`). Obviously, both scenarios are not mutually exclusive and can therefore be setup together, i.e., several intermediate check-points are made before the last one that happens when the `--pitstop` iteration is reached. Either way, each check-point overwrites the restart-file created in the previous one, since only the last successful one is of interest.

By default, any REFMUL3 simulation starts from the initial iteration (t=0). So, in order to continue a simulation from a subsequent state (t>0) stored in a restart-file created during a previous run, it is necessary to signal this information to REFMUL3. This is done via the input parameter `--restart`, which triggers, right at the beginning of the first iteration, the opening of the restart-file and the subsequent loading of all the data stored therein to the corresponding REFMUL3 variables. One of these variables is the iteration counter, which is then updated to the value corresponding to the check-point iteration of the restart-file. The remaining data corresponds to the 3D physical quantities together with the impulse calibration data, as mentioned before.

To consolidate the concept behind the check-point infrastructure of REFMUL3, an example showing the corresponding command line options is in order. Suppose we want to evolve a particular case for 100 000 iterations. On an ideal HPC system, this could be done with

```
mpirun ./refmul3 -n=100000.
```

However, suppose our simulation needed a large number of cores and the we did not want to risk losing many node-hours due to a possible failure of a single node after some time into the simulation. We then decide to make check-points every 20 000 iterations, starting at iteration 39 999. Then our command line would look like

```
mpirun ./refmul3 -n=100000
        --wrtrst=1 --wrtrsti=39999 --dwrtrst=20000.
```

Finally, suppose that the wall-clock limit in our HPC queue allowed only for 60.000 iterations to be executed during a single run. Then, we can split the previous simulation into a first run that stops with a restart-file at iteration 49 999

```
mpirun ./refmul3 -n=100000
    --wrtrst=1 --wrtrsti=39999 --dwrtrst=20000 --pitstop=49999
```

followed by the restart run that reads the state of iteration 49 999 from the restart-file and continues until the end of the simulation

```
mpirun ./refmul3 -n=100000
        --wrtrst=1 --wrtrsti=39999 --dwrtrst=20000 --restart=1.
```

The complete check-point infrastructure described before was implemented in REFMUL3 using the single-file I/O method, independently of the choice made at compilation time for the standard code I/O (`HDF5=1`, `H5T4D=1` or `PHDF5=1`). Similarly to the quantities that require disk access during the initialisation phase of a REFMUL3's simulation, which we discussed in Subsec. 8.3.2, the check-pointing operations were obvious candidates for using exclusively the single-file method, even if at the cost of some global code scalability. Although being called multiple times from the main-loop of the code, depending on how often this is set to happen, having domain-decomposition-free restart files was agreed with Filipe da Silva to be a more important advantage than having the highest possible I/O performance. Not to mention that there is still, in principle, room for improvement of the single-file I/O bandwidth through a more careful tuning of the chunked layout data access (recall Subsec. 8.3.2).

As already mentioned, all remaining output that is generated from within the REFMUL3's main-loop at high frequencies retains the flexibility to use any of the three available I/O methods at compile time via the make variables explained previously (`HDF5=1`, `H5T4D=1` or `PHDF5=1`). Typically, this type of output constitutes the bulk of the data to be stored, often meant for making time-evolution movies, so disk access bandwidth if of utmost importance. This means that it benefits mostly from the multi-file I/O method (`PHDF5=1`), which in the current implementation of REFMUL3 offers the highest I/O performance on Marconi (see Fig. 53). Finally, it is important to note that the check-point and restart infrastructure preserves these I/O operations in the sense that the same output is generated for all I/O methods, independently of executing the simulation using a single run or splitting it over several chained runs via the usage of the `pitstop` input parameter.

8.4. *PDI library*

The PDI library [4] has been developed as a tool to decouple high-performance simulation codes from I/O concerns. Rather than having the code explicitly calling I/O libraries, like HDF5, PDI provides a simple and general interface to make these libraries available to the code. This is done by specifying the I/O operations in a dedicated YAML [5] file, which invokes a system of PDI plugins, one for each different I/O library. The result is that the low-level details on I/O operations can be moved out of the main simulation source code and replaced by the simplified declarative API of PDI. In principle, this makes the code more portable and maintainable, provided of course, that the PDI library is available on the host HPC machine. Another consequence is that switching between I/O libraries or even using a mix of different libraries in a single execution requires minimal code changes, and therefore becomes very easy to achieve. The plugins for HDF5, SIONlib [6] or FTI [7], which are standard I/O libraries, are provided with the current PDI installation. Since REFMUL3 currently uses the HDF5 library for I/O operations, we have in theory all ingredients needed to use PDI within REFMUL3 to replace the existing explicit calls to the HDF5 library with an interface using the PDI library. In practice however, to enable this plan, several conditions had to be met beforehand, as described below.

The first step was to learn the basics about the PDI library and its usage. To that end, Tiago Ribeiro participated in the PRACE training event on "High Performance Parallel IO and post-processing" held at the *Maison de la Simulation* (France) in March 2019 [8]. This was very useful since one of the speakers was the main developer of PDI. Attending this training allowed not only to obtain a deeper understanding of the main concept behind this library, but also to try it on some hands-on exercises. Another important benefit of the visit was that it allowed establishing a direct contact to the PDI team of developers via their dedicated Slack channel [9]. This proved to be instrumental to achieve a working installation of PDI on the Marconi machine, from which resulted, additionally, a new release of the PDI library including some fixes. Currently, the PDI library is available on Marconi using the module system via the commands:

```
module load profile/candidate

module load autoload pdi/0.5.0--openmpi--4.0.1--gnu--7.3.0
```

This particular module depends on the GNU C compiler v7.3 and OpenMPI library v4.0.1. Having PDI working with the Intel compiler and MPI library is something not available on Marconi at the time of writing of this report.

The PDI installation available on Marconi allowed running successfully the PDI hands-on exercises mentioned earlier. They provide example codes that use PDI to interface the parallel HDF5 library, which is essentially the main ingredient needed in REFMUL3. The next step was to create a branch of REFMUL3 that uses PDI as the I/O interface to the HDF5 library. Initially, it was thought to call PDI's API directly from the main source code of REFMUL3, as is suggested by the philosophy of this library. This implies exposing the memory buffers (pointers) of the variables involved in I/O operations (electromagnetic fields), as well as their properties, called metadata in PDI. The metadata comprises mostly the subdomain dimension sizes and MPI task ranks. However, for the single-file method, it also includes the coordinates of each subdomain within the global domain, as well as the dimensions of the latter. Because the field variables in REFMUL3 are evolved in staggered grids, their sizes differ. For this reason, the data for each field variable is stored in a structure that includes the metadata as additional members. This allows passing easily the whole structure to functions in a compact manner within REFMUL3, simplifying its source code. However, as the API of PDI does not support the C-structure type, each metadata component needs to be exposed separately. This implies a long list of instructions in the source code which was undesirable. Therefore, it was decided to wrap the PDI exposing instructions inside a new set of functions, which should further overload the macro function used before to choose between the different I/O methods (recall Subsec. 8.3.2). The current implementation of REFMUL3's PDI branch supports all three previously available I/O methods that use explicitly the HDF5 API (`HDF5=1`, `H5T4D=1` and `PHDF5=1`), plus the multi-file method and the single-file method using instead the PDI API. These two additional variants can be chosen at compile time using the following make options `HDF5=1 PDI=1` or `PHDF5=1 PDI=1`, respectively. In both cases, due to lack of time, only the output operations use the PDI library. This means that only the overloaded function call `escrvCubeH5star` becomes defined as `escrvCubePDI` when using the `make` option `PDI=1`. Further, the function `escrvCubePDI` is used by both multi-file and single-file methods because it merely exposes variables and their metadata to PDI. It is the YAML file which specifies the details about the I/O operations. Therefore, we have two such files, namely, `refmul3_hdf5.yml` and `refmul3_phdf5.yml`, to be used for the multi-file and single file methods, respectively. What is missing is the PDI interface for the input operations (overloaded macro function `leCubeH5star`). So far, even when the PDI usage is requested at compile time, these operations use instead the previously developed functions that invoke directly the HDF5 API. Nevertheless, developing the function `leCubePDI` and extending the existing YAML files to interface the input

operations using PDI should be straightforward if one uses what is already done for the output counterpart.

The motivation to use PDI in the framework of the HLST-REFMUL3+ project was, to a large extent, to assess this library whose concept offers a simplified implementation of the I/O operations. In the framework of REFMUL3, this could be successfully confirmed. To produce the same HDF5 output, the operations that were interfaced with PDI could be implemented using a much simpler code than the original functions relying directly the HDF5 API. While this benefit would have been more significant if the HDF5 I/O operations were not already implemented in REFMUL3, the fact that the newly implemented PDI interface allows a straight forward extension to different I/O libraries remains. With this in mind, one obvious suggestion for the future refers to the usage of the PDI to interface the check-point infrastructure of the code, currently bound to the HDF5 library. For these specific I/O operations there are dedicated libraries, like FTI [9], that could be easily used by REFMUL3 via a dedicated PDI plugin. However, there are also some disadvantages of PDI that became clear. On the one hand, there is the dependency on the PDI library that is not yet part of the standard software stack of HPC centers. Hence, its installation needs to be triggered by the user, as was done by us for the case of Marconi. On the other hand, it is known that PDI does not support the complete API of the HDF5 library, so more advanced usage of this library could be barred when using PDI. For instance, at the time of writing of this report, we do not know if the current PDI API allows specifying HDF5 data chunked layouts (see Subsec. 8.3.2). Finally, it should be mentioned that, because the PDI branch constitutes a extension of the original code that retains all the previously available I/O methods and adds two additional ones, it was merged back into the trunk.

8.5. *Other performance related topics*

Besides the topics covered before on I/O operations, which constitute the bulk of the work requested in the project proposal, some time was dedicated to additional activities related to the overall performance of REFMUL3. In particular, the topic of the impulse calibration response in REFMUL3 is worth mentioning here. Due to its nature, it involves very anisotropic 3D arrays. They comprise a relatively small grid-count commensurate with the wave-guide cross section, of about $\mathcal{O}$ (10^1x10^1) grid-nodes in the *yz*-plane. The third dimension stores the temporal evolution of the impulse response and is larger in size, with typically $\mathcal{O}$ $(10)^3$ grid-nodes. The volume of such quantities is therefore much smaller than the global simulation domain and their anisotropy prevents directly using the same 3D decomposition scheme. For these reasons, it was decided during the original HLST-REFMUL3 (2017) project [1] not to decompose them, but rather to replicate the information on each task. While this solution made sense at the time for practical reasons, the implied serialisation of the operations performed on these quantities naturally impairs the scalability of the global code, especially when larger numbers of resources are involved. This can be inferred from the trace analysis of one single iteration of the REFMUL3 code on 8 cores shown in Fig. 56. There, the calibration operations are highlight in light blue, the remaining (parallelised) operations in blue and the MPI operations in red. As the number of resources is increased, the cost of the last two added together decreases, due to the known good strong scaling behaviour of REFMUL3 [1]. However, the cost of the calibration operations (light blue) will remain unchanged due to their serial nature. Hence, the relative cost of these operations will increase and therefore potentially degrade the global scalability of the code.

Fig. 56 Snapshot of the Intel Trace Analyzer and Collector (ITAC) [10] analysis of a single iteration of the REFMUL3 code on a small test domain of 220x70x70 in (*x*,*y*,*z*), excluding I/O. The very small global domain size used exacerbates the weight of the serial calibration operations. This was done specifically to highlight the problem for the sake of the discussion.

Devising a parallelisation for the calibration operations goes beyond the scope of the current project proposal and will therefore not be attempted. Not only is there a lack of time available to do so, but also the actual calibration algorithm used in the 3D code is still a topic under investigation, and might therefore still be changed. Nevertheless, it is important to recommend revisiting the issue in the future, as it can potentially affect the overall code scalability. In fact, a small change was already made in REFMUL3, which involves the same arrays. Namely, during the initialisation phase, the files for the impulse response calibration need to be read from disk. Until now, this was done by the master MPI rank, which would sequentially read each file and then broadcast the data to the remaining MPI ranks. Because there are a total of four different files to be read, this was changed in order to have four MPI tasks do that in parallel. Consequently, these initialisation costs were cut by roughly a factor of four.

8.6. *Conclusions and outlook*

The project REFMUL3+ (2019) requested support for the assessment and extension of parallel I/O capabilities in the code REFMUL3. The corresponding activities comprised I/O bandwidth scaling measurements on Marconi and a subsequent effort to tune the existing parallel HDF5 implementation for better performance. They further comprised the implementation of a check-point and restart file infrastructure and an assessment regarding the interfacing of the I/O operations in REFMUL3 using the Portable Data Interface (PDI) library [6]. The work progressed as planned, and the milestones achieved are summarised in the paragraphs below.

Strong scaling measurements of REFMUL3 on Marconi revealed that the two main I/O methods currently available in the code have complementary characteristics. The multi-file solution, with each MPI task independently accessing its own file, yields the best performance when larger numbers of resources are used. However, it produces a large number of files (even with the variant that stores the iterations as the fourth dimension in the datasets), that moreover are domain decomposition dependent. On the other hand, the single-file method relies on the pHDF5 library to store the data in a domain independent fashion. While this is very flexible in terms of data input for subsequent simulations, in particular for restart files, its measured poorer scaling performance makes it less appropriate for cases where disk access bandwidth is of

utmost importance. For this reason, an tuning effort was made to use HDF5 chunked data layouts. It proved that such techniques were applicable to REFMUL3, despite the fact that its subdomains might differ slightly in size. Even if the chunk size choice made during the project did not improve the I/O performance, its implementation easily allows for future studies to scan this parameter and find the optimal value, therefore potentially improving the single-file performance of the method. Nevertheless, even considering that some gains can be obtained, we believe that the multi-file method should still have a performance edge, at least up to moderately large numbers of MPI tasks (few thousands). The reason simply being that the current parallel file systems require multiple concurrent disk accesses to obtain peak performance, which is exactly what the multi-file method does. It is worth mentioning also that using the hybrid version of REFMUL3, with large numbers of threads per MPI task improves the performance of the single-file method by reducing the degree of concurrency in the file access. In terms of the multi-file method, the advantages of running REFMUL3 in hybrid mode stem from the fact that it reduces the overall number of output files produced, which is beneficial, especially when very large numbers of resources are used. From the previous discussion it is clear why it was decided, together with Filipe da Silva, to keep both multi-file (including its variant using a 4D dataset to accomodate the time domain) and single-file methods available to be chosen at compile time.

The check-point and restart infrastructure was implemented in REFMUL3 based on the single-file method, independently of the choice made at compile time for the remaining I/O operations of the code. Even if at the cost of some global code scalability, this is justified by the flexibility that having domain decomposition independent restart-files provides. It is now possible to split large simulations that might not fit within the wall-clock limits of the HPC queues at hand into a number of smaller runs that need to be executed in a chained fashion. Also, periodic check-pointing is also available, which offers much more confidence against possible hardware failures, whose probability increase with the number of resources utilised.

The work on the PDI library involved participating in a PRACE training dedicated to the topic of high performance I/O techniques [10], which specifically covered the PDI library. From this it was possible, in collaboration with the developers of PDI and the Marconi Support Team, to have this library available on Marconi via the standard module system. The hands-on exercises provided during the training include code examples with all the ingredients required to deploy PDI to REFMUL3, which was also done. Due to lack of time, PDI is currently only available for the output functions of REFMUL3, but for both multi-file and single-file methods. This provides the proof of concept demanded in the projects milestones and serves as a reference to extend it to the input operations, whose implementation should be straight forward. Moreover, it confirmed also the claim that the PDI interface to HDF5 is much simpler that the HDF5 counterpart. This means that implementing the same HDF5 I/O operations via the PDI library was much easier. Moreover, because the description of those operations is pushed to a YAML file, which uses a HDF5 specific plugin (included with PDI), it should be straight forward to use an alternative I/O library by simply choosing the new PDI plugin. This applies also to the possibility of using the PDI library to interface the check-point operations, for which there are dedicated libraries (e.g. FTI [9]) that could prove more efficient than the currently used HDF5 library. This has not been done due to lack of time, but could be something worth considering for the future. Once more, the PDI interfacing already implemented in REFMUL3 should render this task easily feasible. The drawbacks of using PDI on REFMUL3 refer to its dependency on this new library, which must be available as part of the software stack of the HPC centers. Additionally, the fact that the simplified API of PDI might prevent direct usage of more advanced features of the I/O libraries being interfaced, at least without explicit extension of the provided HDF5 plugin, is also something to consider.

In parallel to the tasks described before, for which explicit support has been requested, some time was also put into supporting debugging activities. Filipe da Silva investigated the issue that was found towards the end of the REFMULIO (2018) project

[3] in more detail. It could be concluded that it was not a bug *per se*, but rather a characteristic of the numerical method used. Further investigations in this topic will follow, but outside the framework of this project. Additionally, a trace analysis using the ITAC software [12] on REFMUL3 revealed an inefficiency in the initialisation of the impulse response calibration. A simple change made in this part of the code alleviated the problem, resulting in a factor of four reduction in its cost. A recommendation for the future, also beyond the current project's scope, concerns the need to devise a proper parallelisation for the operations related to the impulse response calibration. The small and highly anisotropic arrays involved, which are incompatible with the MPI communicators used for the global domain, are the reason why this was not done originally. However, together with the exploitation of techniques for overlapping communication and calculations throughout the code, they constitute changes that are expected to significantly extend the scalability of REFMUL3 on higher numbers of cores.

8.7. *References*

[1] T. Ribeiro, *Final report on HLST project REFMUL3* (2017)

[2] K.S. Yee, *Numerical solution of inital boundary value problems involving maxwell's equations in isotropic media*, IEEE Trans. on Antennas and Propagation **14** (1966) 302

[3] T. Ribeiro, *Final report on HLST project REFMULIO* (2018)

[4] The HDF Group, *Hierarchical data format version 5* (2000–2017) http://www.hdfgroup.org/HDF5

[5] The HDF Group, Chunking in HDF5 http://portal.hdfgroup.org/display/HDF5/Chunking+in+HDF5

[6] J. Bigot et. al, *PDI, the portable data interface* (2018–2019) https://gitlab.maisondelasimulation.fr/jbigot/pdi

[7] *YAML Ain't Markup Language* https://yaml.org

[8] *SIONlib: Scalable I/O library for parallel access to task-local files* (2008–2018) http://www.fz-juelich.de/jsc/sionlib/

[9] *FTI: Fault Tolerance Interface* (2008–2018) http://leobago.github.io/fti

[10] PRACE Training, *High Performance Parallel IO and post-processing*, Maison de la Simulation, France, March 11-13, 2019 https://events.prace-ri.eu/event/814/

[11] *Slack collaboration hub*, https://pdidev.slack.com/

[12] *Intel Trace Analyzer and Collector* (2019) https://software.intel.com/en-us/trace-analyzer/documentation

9. Report on HLST project JORSOLV

9.1. *The JOREK code*

JOREK is a nonlinear MHD code, which is used widely in Europe to study large-scale instabilities in X-point tokamak plasmas. These studies are very expensive numerically, especially when the interaction among a large number of harmonics is involved. Thus, complex and therefore more realistic problems require a substantial amount of computational resources, for example, a significant amount of memory. As a consequence, it is challenging to investigate these realistic configurations. Thus, the present project aims to optimize the solver part of the JOREK code in a way that memory consumption decreases significantly, thereby making the solver more efficient.

9.2. *Current Status of the JOREK solver*

JOREK is an MPI+OpenMP parallel code written in Fortran 90 that solves the reduced-MHD equations using a finite-element method. The code uses a fully implicit scheme in order to employ a substantial time step. As a result, a set of linear equations $Ax = b$, in the form of a large sparse matrix, needs to be solved for every time step. JOREK utilizes the iterative method GMRES (Generalized Minimal RESidual) to solve such a system. GMRES requires a preconditioned matrix for a faster convergence of the solution. The criteria for selecting a preconditioned matrix are that it must be comparable to the matrix A and must also be easily invertible. Once a suitable B is available, its inverse can be left-multiplied in the equation $Ax = b$ to acquire an optimal starting point for the GMRES iterations. The preconditioning in JOREK is achieved through a direct solver: either PASTIX (PArallel Sparse maTrIX) or MUMPS (MUltifrontal Massively Parallel Sparse). In the present project, we focus on the use of PASTIX in the JOREK solver.

PASTIX is an open source scientific library that contains a high-performance parallel solver for extensive sparse linear systems. It uses LU factorization with static pivoting for asymmetric matrices. Note that, for the preconditioning, we usually do not obtain the LU decomposition of the whole matrix but only of the blocks corresponding to the toroidal harmonics. These blocks are linearly independent of each other and lie along the main diagonal of the matrix. As an example, for $n_{tor} = 5$ we need to obtain the LU decomposition of the three blocks corresponding to toroidal harmonics 0, 1 and 2 for the preconditioning. The matrix A is distributed among the MPI tasks and each task has access to a part of the global matrix, which is denoted by A_{glob}. At present, these harmonic blocks are being extracted from the global matrix A_{glob} using MPI all-to-all communication because each A_{glob} has contributions from all harmonics. However, the drawback of this approach is that the communication time among the MPI tasks goes up dramatically if the problem size is increased by enhancing the resolution and the number of harmonics.

A second drawback is that the direct solver consumes a tremendous amount of memory in the LU decomposition of the preconditioning matrix. The reason for such a large memory consumption is that for each non-zero harmonic, the sine and the cosine parts are being treated separately, which makes the block size much more extensive. In this project, we are going to address these two drawbacks in an effort to improve the efficiency of the JOREK solver.

9.3. *The PASTIX solver on Marconi*

Since we are planning to use the PASTIX solver in the JOREK code, it is necessary to have a thorough understanding of it. The project thus began by installing PASTIX and other supporting libraries on the Marconi supercomputer.

PASTIX makes use of the SCOTCH library in order to reduce the fill-in generated during the LU decomposition. Thus, as a first step, 'scotch_5.1.12' and 'pastix_release_4492' are compiled for both MPI_THREAD_FUNNELED (only one thread can make MPI calls) and MPI_THREAD_MULTIPLE (all threads can make MPI calls) in separate folders on Marconi. The purpose of compiling PASTIX in two different directories is that both threading options are being used in JOREK. The proper installation of the libraries is then verified by running the included examples in the PASTIX package.

In order to gain further insight and to double check the newly installed PASTIX libraries, a small code was written to solve a couple of test cases for non-symmetric matrices with real coefficients. The numerical results have shown a good agreement with the analytical ones. For the final verification, the JOREK code has been compiled with the new PASTIX libraries and the results from the old and new libraries are found to be identical.

9.4. *Direct extraction of harmonic matrix blocks*

In order to extract the preconditioning matrix, i.e., harmonic blocks, the present solver makes use of MPI all-to-all communication, which gets very expensive computationally for more realistic problems. This drawback can be avoided if the harmonic blocks are extracted directly from the subroutine mod_elt_matrix.F90. Therefore, the subroutine has been modified in a way that we can extract the elementary matrix for each harmonic. Then, in the new subroutine construct_harmonic_matrix_mod.F90, these elementary matrices are looped over all the elements to get the desired harmonic blocks. Note that each block can be addressed by one or more MPI tasks. However, as of now we have just considered one MPI task per harmonic block. We have also done a one-to-one comparison of matrix entries from the direct extraction and the MPI all-to-all communication. The deviation between the two is found to be negligible.

9.5. *Benchmarking*

In Fig. 57, we compare the time evolution of the energy between the original (blue line) and the new solver (red line) for an arbitrarily chosen physics model = 303, n_{tor} = 17 and period = 1. The new solver, extracts the harmonic blocks directly from the elementary matrix (without domain decomposition for now, i.e., only one MPI task per block).

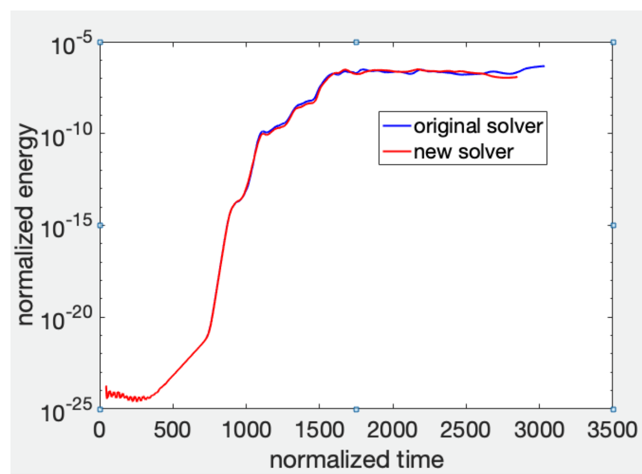

Fig. 57 Comparison of the energy evolution of a single harmonic between the original solver (blue line) and the new solver (red line) for model = 303, n_{tor} = 17 and period = 1.

In Fig. 57, both the red and the blue lines are overlapping in the linear regime. This

confirms that the growth rate of the linear instability remains unaffected with the new solver. Nevertheless, a deviation in the nonlinear regime is evident. This is expected, because the system is highly nonlinear, and any minor change in the numerics would affect the solution in the nonlinear regime. However, the physics of the system, i.e., the nonlinear saturation level of the energy, remains unchanged with the new solver. This validates the new method of harmonic block extraction.

9.6. *Performance comparison*

In Table 9, we compare the elapsed time in the MPI all-to-all communication (original solver) and the elapsed time in the direct extraction (new solver without domain decomposition). Here we consider model = 303 with period= 1 and n_{tor} = 3, 17, 23. It is found that, although the direct extraction is quite efficient for the small case (n_{tor} = 3) it becomes more or less comparable to MPI all-to-all communication for larger cases n_{tor} = 17, 23. In Table 10, we consider only the n_{tor} = 17 case with an increased resolution. Here we notice that for larger problems the MPI all-to-all communication gets quite expensive, however, the direct extraction without domain decomposition gets even more expensive.

n_{tor}	**No. of MPI Tasks**	**Elapsed time in MPI all-to-all communication (in seconds)**	**Elapsed time in direct extraction (in seconds)**
3	2	3.52	1.29
17	9	12.17	7.85
23	12	16.51	12.27

Table 9 Performance comparison between the MPI all-to-all communication and the direct extraction of harmonic blocks for model = 303 with period = 1 and n_{tor} = 3, 17, 23 at a normal resolution of n_{radial} = 81, n_{pol} = 128, n_{flux} = 40, n_{tht} = 64, n_{open} = 15, n_{leg} = 15 and $n_{private}$ = 9.

n_{tor}	**No. of MPI Tasks**	**Elapsed time in MPI all-to-all communication (in seconds)**	**Elapsed time in direct extraction (in seconds)**
17	90	38.78	71.61
17	180	29.55	72.10
17	234	28.80	71.98

Table 10 Performance comparison between the MPI all-to-all communication and the direct extraction of harmonic blocks for model = 303 with period = 1 and n_{tor} = 17 at a resolution of n_{radial} = 137.7, n_{pol} = 217.6, n_{flux} = 120, n_{tht} = 192, n_{open} = 45, n_{leg} = 45 and $n_{private}$ = 27.

9.7. *Domain decomposition*

The next target is to implement a domain decomposition so that each block can be addressed by multiple MPI tasks. We can thereby reduce the time spent in the direct extraction of the harmonic blocks significantly. However, in the present form the direct solver PASTIX cannot deal with a domain decomposition. We thus need to modify the PASTIX implementation in a way that it can solve distributed matrices as well. This work is still in progress

9.8. *Memory consumption in the LU decomposition*

As we have discussed in Section 9.2, the second drawback of the current JOREK solver is that it consumes a large amount of memory for the LU factorization, a necessary step to obtain good preconditioning. Theoretically, the matrix elements corresponding to the sine and cosine components of the harmonics should have the pattern of matrix *P* as in Eq. 1, which would allow us to replace these four real entries by one complex entry $z = a + ic$. Thus, we could reduce the size of the matrix block, and hence the memory consumption by one half. We could then employ a complex sparse matrix solver (e.g., PASTIX) for the preconditioning.

$$P = \begin{pmatrix} a & -c \\ c & a \end{pmatrix}, Q = \begin{pmatrix} a & -c \\ b & d \end{pmatrix}, R = \begin{pmatrix} (a+d)/2 & -(b+c)/2 \\ (b+c)/2 & (a+d)/2 \end{pmatrix} \quad (1)$$

9.9. *The complex PASTIX solver*

It is important to note that the PASTIX package provides support for codes written in `C` as well as `Fortran`, and that there are different functions for each programming language. Nevertheless, we found that for complex matrices there are no special functions for `Fortran` as there are for `C`, e.g., `z_pastix()`. Instead, for `Fortran`, PASTIX calls the `C` functions, and in order to enable those calls, PASTIX needs to be compiled with the flags '`-DFORCE_COMPLEX -DTYPE_COMPLEX`'. After trivial modifications in the test code written for real matrices, a couple of examples with complex coefficients were solved to verify the solver. A good match between the numerical and the analytical results is found here as well.

When we looked into the subroutine '`solve_mat_n.f90`' and printed the arrangement of the matrix elements for the case `n_tor = 3` with toroidal mode number `1`, the matrix entries corresponding to the sine and cosine components were found to be in the form of matrix *Q* in Eq. 1, which cannot be replaced by one complex variable. We believe that this peculiar pattern is due to rounding errors and can be fixed by enforcing the desired symmetry in the preconditioning as expressed by the matrix *R* in Eq. 1.

In order to verify this assumption, we have considered the same physics model = 303, n_{tor} = 3, and compared the energy evolution for a single harmonic, with and without the enforced symmetry. As we can see in Fig. 58, both solutions (red and blue) are completely overlapping and thus confirm our assumption.

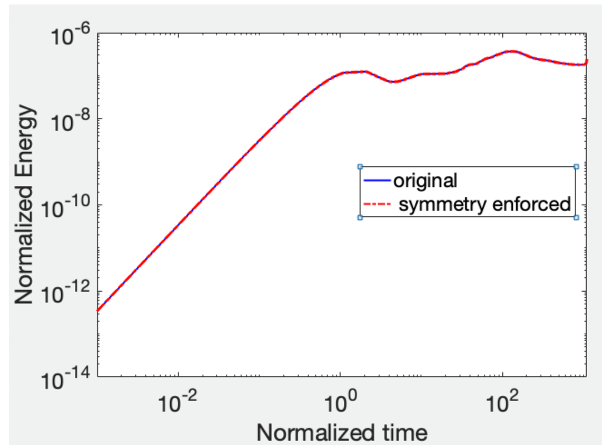

Fig. 58 Comparison of the energy evolution of a single harmonic between the original solver (blue line) and the solver with enforced symmetry (red line) for model = 303, n_{tor} = 3 and period = 1.

9.10. *Conclusions*

The project aims at optimizing the solver part of the JOREK code by reducing its memory consumption and enhancing its efficiency.

Presently, the preconditioning matrix in JOREK is being constructed from a global matrix. This matrix is distributed among the MPI tasks, which makes it necessary to use expensive MPI all-to-all communication to extract the preconditioning matrix. In order to reduce the computational cost, we are extracting the preconditioning matrix directly from the elementary matrix as blocks of harmonics. Each harmonic block can be addressed by one or more MPI tasks. Nevertheless, in the beginning, we have only considered one MPI task per harmonic block and found that for realistic problems, the direct extraction (without domain decomposition) exhibits much higher computational costs in comparison to the MPI all-to-all communication. Thus, the domain decomposition is necessary to increase the efficiency of the solver, i.e., each harmonic block needs to be addressed by multiple MPI tasks.

In the present form, the JOREK solver does not support this domain decomposition because the currently used PASTIX solver `pastix_fortran()` cannot solve distributed matrices. Therefore, a distributed PASTIX solver `dpastix_fortran()` needs to be implemented in JOREK. However, there seems to be a bug in the distributed PASTIX library, which has recently been reported to the PASTIX team. Once the bug is fixed, we will implement `dpastix_fortran()`in JOREK, and compare the results.

At the same time, we are working on lowering the memory consumption. The memory consumption can be reduced by converting the real-valued preconditioning matrix into a complex one and then employing the complex PASTIX solver for it. However, this conversion is only possible if the real matrix exhibits the desired symmetry, i.e., diagonal elements should be identical and off-diagonal elements should only have opposite signs. We have found that the symmetry, which exists analytically, may be broken by numerical errors. Therefore, the real matrix can no longer be converted into a complex one. Nevertheless, it has been confirmed that if the symmetry is lost due to numerical errors, one can enforce it again, by replacing the diagonal and off-diagonal entries by their respective average values, without meaningfully affecting the results. Thus, the next steps are to convert the preconditioning matrix into a complex one, employ the complex PASTIX solver, and finally compare the results.