E-Infrastructures

H2020- INFRAEDI-2018-2020

INFRAEDI-01-2018: Pan-European High Performance Computing infrastructure and services (PRACE)

PRACE-6IP

PRACE Sixth Implementation Phase Project

Grant Agreement Number: INFRAEDI-823767

D8.5

Final report: Including performance results on (pre)Exascale systems

Final

Version: 0.7
Author(s): Alex Upton, ETH Zürich
Joost VandeVondele, ETH Zürich
Fabio Affinito, CINECA
Date: 30.06.2022

Project and Deliverable Information Sheet

PRACE Project	**Project Ref. №: INFRAEDI-823767**	
	Project Title: PRACE Sixth Implementation Phase Project	
	Project Web Site: https://www.prace-ri.eu/about/ip-projects/	
	Deliverable ID: D8.5	
	Deliverable Nature: Report	
	Dissemination Level: PU*	**Contractual Date of Delivery:** 30/06/2022
		Actual Date of Delivery: 30/06/2022
	EC Project Officer: Leonardo Flores Añover	

* - The dissemination level are indicated as follows: **PU** – Public, **CO** – Confidential, only for members of the consortium (including the Commission Services) **CL** – Classified, as referred to in Commission Decision 2005/444/EC.

Document Control Sheet

Document	**Title: Final report: Including performance results on (pre)Exascale systems**	
	ID: D8.5	
	Version: 0.7	**Status: *Final***
	Available at: https://www.prace-ri.eu/about/ip-projects/	
	Software Tool: Microsoft Word 2016	
	File(s): D8.5_v0.7_final.docx	
Authorship	**Written by:**	Alex Upton, ETHZ Joost VandeVondele, ETHZ Fabio Affinito, CINECA

	Contributors:	Fabio Affinito, CINECA Momme Allalen, LRZ Simone Bacchio, CaSToRC Vicenç Beltran, BSC Marco Bettiol, ETH Zürich Mauro Bianco, ETH Zürich John Biddiscombe, ETH Zürich Ricard Borrell, BSC Fabian Bösch, ETH Zürich David Brayford, LRZ John Brennan, ICHEC Dirk Brömmel, JUELICH Tomáš Brzobohatý, IT4I Mark Bull, EPCC Zahra Chitgar, JUELICH Laurent Chôné, CSC Olivier Coulaud, CENAERO Tilman Dannert, MPCDF Edoardo Di Napoli, JUELICH Myles Doyle, ICHEC Jacob Finkenrath, CaSToRC Christophe Geuzaine, ULIEGE Paul Gibbon, JUELICH Luc Giraud, INRIA Aleksander Grm, UL Kenneth Hanley, ICHEC Berk Hess, KTH Koen Hillewaert, ULIEGE Victor Holanda, ETH Zürich Guillaume Houzeaux, BSC Luigi Iapichino, LRZ Alberto Invernizzi, ETH Zürich Niclas Jansson, KTH Joe Jordan, KTH Prashanth Kanduri, ETH Zürich Sebastian Keller, ETH Zürich Leon Kos, UL Marcin Krotkiewski, UiO Chiara Latini, CINECA Carlos Lopez, MPCDF Martti Louhivuori, CSC Georgi-os Markomanolis, CSC Michele Martone, LRZ Michal Merta, IT4I Teodor Nikolov, ETH Zürich Henrik Nortamo, CSC Lee O'Riordan, ICHEC Adam Peplinski, KTH

		Janes Povh, UL Lara Querciagrossa, CINECA Michel Rasquin, CENAERO Cristóbal Samaniego, BSC Mikael Simberg, ETH Zürich Matthieu Si-monin, INRIA Ujjwal Sinha, JUELICH Raffaele Solcà, ETH Zürich Thomas Toulorge, CENAERO Alex Upton, ETH Zürich Joost VandeVondele, ETH Zürich Ivona Vasileska, UL Radim Vavřík, IT4I Jonathan Vincent, KTH Xinzhe Wu, JUELICH Shuhei Yamamoto, CaSToRC Artem Zhmurov, KTH
	Reviewed by:	Thomas Eickermann, JUELICH Pedro Alberto, UCoimbra
	Approved by:	MT/TB

Document Status Sheet

Version	Date	Status	Comments
0.1	04.10.2021	1st Draft	1st version with input from all projects
0.2	19.10.2021	2nd Draft	2nd version following internal review
0.3	25.10.2021	3rd Draft	Final 'pre-submission' version following internal review, will be updated with annexes in May 2022
0.4	10.06.2022	4th Draft	Updated version with annex following WP8 extension
0.5	21.06.2022	5th Draft	Updated version following internal review
0.6	22.06.2022	6th Draft	Updated version
0.7	30.06.2022	7th Draft	Final version for EC

Document Keywords

Keywords:	PRACE, HPC, Research Infrastructure, Exascale, Forward-looking software solutions

Table of Contents

Project and Deliverable Information Sheet i
Document Control Sheet i
Document Status Sheet iii
Document Keywords iv
List of Figures vii
List of Tables xi
References and Applicable Documents xii
List of Acronyms and Abbreviations xv
List of Project Partner Acronyms xvii
Executive Summary 1
1 Introduction 2
2 Compatibility of PRACE-6IP WP8 projects on leading European HPC systems 3
3 PiCKeX: Particle Kinetic codes for Exascale plasma simulation 7
3.1 Introduction and summary 7
3.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 8
3.3 Interactions with stakeholders, users, outreach and publications 12
3.4 Overall assessment of achievements and future developments 13
4 MoPHA: Modernisation of Plasma Physics Simulation Codes for Heterogeneous Exascale Architectures 14
4.1 Introduction and summary 14
4.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 15
4.3 Interactions with stakeholders, users, outreach and publications 24
4.4 Overall assessment of achievements and future developments 25
5 LoSync: Synchronisation reducing programming techniques and runtime support 27
5.1 Introduction and summary 27
5.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 28
5.3 Interactions with stakeholders, users, outreach and publications 32
5.4 Overall assessment of achievements and future developments 33
6 FEM/BEM based domain decomposition solvers 35
6.1 Introduction and summary 35
6.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 36
6.3 Interactions with stakeholders, users, outreach and publications 39

6.4 Overall assessment of achievements and future developments 40
7 Performance portable linear algebra 41
7.1 Introduction and summary 41
7.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 42
7.3 Interactions with stakeholders, users, outreach and publications 48
7.4 Overall assessment of achievements and future developments 49
8 LyNcs: Linear Algebra, Krylov methods, and multi-grid API and library support for the discovery of new physics 51
8.1 Introduction and summary 51
8.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 52
8.3 Interactions with stakeholders, users, outreach and publications 58
8.4 Overall assessment of achievements and future developments 60
9 QuantEx: Efficient Quantum Circuit Simulation on Exascale Systems 62
9.1 Introduction and summary 62
9.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 63
9.3 Interactions with stakeholders, users, outreach and publications 67
9.4 Overall assessment of achievements and future developments 68
10 GHEX: Generic Halo-Exchange for Exascale 70
10.1 Introduction and summary 70
10.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 72
10.3 Interactions with stakeholders, users, outreach and publications 77
10.4 Overall assessment of achievements and future developments 79
11 ParSec: Parallel Adaptive Refinement for Simulations on Exascale Computers 81
11.1 Introduction and summary 81
11.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 82
11.3 Interactions with stakeholders, users, outreach and publications 87
11.4 Overall assessment of achievements and future developments 89
12 NB-LIB: Performance portable library for N-body force calculations at the Exascale 91
12.1 Introduction and summary 91
12.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems 92
12.3 Interactions with stakeholders, users, outreach and publications 95
12.4 Overall assessment of achievements and future developments 97
13 Conclusions 99

Annex A: Benchmarking and performance results obtained on leading HPC systems during WP8 extension 101

A.1 PiCKeX: Particle Kinetic codes for Exascale plasma simulation 101

A.2 MoPHA: Modernisation of Plasma Physics Simulation Codes for Heterogeneous Exascale Architectures 103

A.3 Performance portable linear algebra 105

A.4 LyNcs: Linear Algebra, Krylov methods, and multi-grid API and library support for the discovery of new physics 108

A.5 QuantEx: Efficient Quantum Circuit Simulation on Exascale Systems 110

A.6 GHEX: Generic Halo-Exchange for Exascale 113

A.7 ParSec: Parallel Adaptive Refinement for Simulations on Exascale Computers 115

A.8 NB-LIB: Performance portable library for N-body force calculations at the Exascale 117

List of Figures

Figure 1: Proton acceleration by circularly polarized, multi-petawatt laser pulse 7
Figure 2: The strong scaling using the moving window algorithm with a simulation window comprising 25000 X 6400 cells with 10 particles per cell for 8000 timesteps. (a) Comparison of the strong scaling results between the initial and new version of EPOCH from 32 to 1536 nodes (1536 to 73728 CPU cores) on JUWELS. (b) Comparison of the strong scaling results between the initial and the new version of EPOCH from 32 to 512 nodes (4096 to 65536 CPU cores) on JURECA-DC. 9
Figure 3: (a) Speed-Up obtained by the new version of EPOCH over the initial version from 32 to 1536 nodes (1536 to 73728 CPU cores) on JUWELS. (b) Speed-Up obtained by the new version of EPOCH over the initial version from 32 to 512 nodes (4096 to 65536 CPU cores) on JURECA-DC. 10
Figure 4: a) CPU version b) GPU version of the particle mover algorithm in OOPD1 11
Figure 5: Test case in OOPD1 11
Figure 6: Benchmark of the CPU and full particle mover GPU version of OOPD1 12
Figure 7: Turbulent flow in a fusion plasma simulation 14
Figure 8: Gantt diagram of the tasks performed during the computation of the right-hand side vector (rhs) required in the time-step and how they were distributed on the available hardware. . 16
Figure 9: Gantt diagram of the computation of the rhs. The new task-based parallelism allows to overlap the computation of entire terms (which is not possible in the MPI implementation) and now they can be computed concurrently in both the CPU (upper part) and GPU (lower part). 17
Figure 10: Scalability study of Vlasiator on the Mahti supercomputer with total time (blue) and propagation time (red) in seconds. 19
Figure 11: Scaling of StruGePiC on Puhti 20
Figure 12: SIMPIC workflow diagram. It shows the general algorithm flow of the PIC codes which consists of two algorithms: Particle mover and field solver. 21

Figure 13: Comparing performance of various accelerated versions of SIMPIC on the VIZ cluster. Left: Runtime Plot of the Particle Mover against number of particles. Right: Runtime plot of field solver against number of cells. 23
Figure 14: Timeline of One Time Step of the full SIMPIC GPU version. 23
Figure 15: Timeline for StarPU particle mover task. 24
Figure 16: Software architecture for hybrid applications using TAMPI and TAGASPI libraries.27
Figure 17: State transition diagram for tasks (blocking mode) 28
Figure 18: Gauss-Seidel strong scaling with a 256Kx128K matrix and 1000 timesteps in MN4 from 1 to 256 nodes. Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input (64Kx32K matrix and 1000 timesteps) for the experiments from 1 to 8 nodes. 29
Figure 19: Gauss-Seidel throughput varying the block size with a 128Kx128K matrix and 500 timesteps in Marenostrum4 with 128 nodes. 30
Figure 20: miniAMR strong scaling in Marenostrum4 from 1 to 256 nodes. The lower shows the efficiency for both the total time and assuming a negligible refinement time (NR). Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input for the experiments from 1 to 8 nodes. 31
Figure 21: miniAMR throughput varying the number of computed variables in Marenostrum4 with 128 nodes. The figure shows the throughput for both the total time and assuming a negligible refinement time (NR). Notice that the vertical axis (throughput) starts at 1000 GUpdates/s. 32
Figure 22: Capabilities of the ESPRESO library. 35
Figure 23: Frequency response of the electric motor case computed using 450 nodes of the Salomon cluster at IT4Innovations in 714 s (15 million DOFs, 60 frequency samples). 36
Figure 24: Strong parallel scalability of the harmonic analysis solver on JUWELS Booster module. 37
Figure 25: Scalability of the individual phase of mesh manipulation on the JUWELS cluster module 38
Figure 26: Scalability of the READ + PARSE and MESHING phases on the Karolina system ...39
Figure 27: Overview of DLA-Future 41
Figure 28: Cholesky factorization on Daint MC. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node. 43
Figure 29: Cholesky factorization on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node). 43
Figure 30: Cholesky factorization on Marconi 100. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node). 44
Figure 31: Triangular solver on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node). 44

Figure 32: Transformation from generalized to standard eigenproblem on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node). 45
Figure 33: Trace of 1 rank (over a total of 4) for the execution of 4 independent Cholesky decompositions. The tasks of each factorization are depicted with a different color. (Due to a limitation of the trace utility, MPI communications cannot be identified, therefore they are all colored in light-green). The trace above shows the case in which after each factorization a synchronization point is added, the trace below shows the case in which the factorizations are allowed to overlap. 45
Figure 34: Runtime of ChASE as a vertical stacked bar plot, including includes the fractions of runtime of numerical modules 46
Figure 35: Speedup of the ChASE GPU implementation compared with the ChASE CPU implementation 47
Figure 36: ChASE weak scaling results across increasing number of nodes 48
Figure 37: Important algorithmic steps in the Krylov accelerated multigrid solver developed in the LyNcs project 51
Figure 38: (Left) Strong scaling study of the QUDA Dirac operators on the fine (D), intermediate (Dc) and coarsest (Dcc) grids for a lattice of size 963 x 128. (Right) Strong scaling of the coarsest operator Dcc varying the number of right-hand sides (rhs) inverted at the same time. 55
Figure 39: (Left) Median of speedup ratio between SpMM measurements with by-rows operands layout and by-columns, on different machines and for different right hand sides count. The by-rows layout is recommended in LIBRSB-1.3 because of its better locality in the lower level loops improves performance. Notice how for NRHS=1, that is SpMV, the layout is the same, and so the performance difference vanishes. (Right) Median of speedup when comparing LIBRSB-1.3 samples to LIBRSB-1.2 ones. Notice how with one exception, each machine/NRHS combination has been (overall) improved over LIBRSB-1.2. 56
Figure 40: Strong scaling of DDalphaAMG with several lattice sizes and number of right hand sides. 57
Figure 41: Comparison between the total iteration count of different Block-Krylov methods at the coarsest level of an 3 level multi-grid approach using a lattice with volume 64*32*32*32 employing twisted mass fermions. 58
Figure 42: Expected area of applicability of simulation methods 62
Figure 43: Computation of 2048 amplitudes for the 5x5x24 RQC, with a single sliced bond, on 4 nodes with an increasing number of processes 64
Figure 44: NVIDIA Nsight profiler analysis report of code on heterogeneous Marconi100 System, with an overall summary highlighting the bottlenecks. 67
Figure 45: Overview of GHEX 71
Figure 46: Transport layer benchmark results on Betzy 74
Figure 47: Impact of HWCART on the performance of halo exchange on Betzy 75
Figure 48: Results obtained on the Piz Daint Multicore partition 75
Figure 49: Benchmark results on the Piz Daint GPU partition 76
Figure 50: GHEX scaling on the Piz Daint multicore and hybrid partitions 77
Figure 51: Weak scaling benchmarks on the Piz Daint GPU partition using unstructured meshes 77

Figure 52: A vortical structure of the flow around a simplified rotor (left) and the refinement level structure for the same simulation (right). Work performed in collaboration with CINECA.81
Figure 53: Scaling of the OpenACC version of Nek5000 on several GPU enabled machines. (left) Results in JUWELS Booster. (right) Results including JUWELS Booster at Jülich in Germany, Berzelius at NSC in Sweden, Longhorn at TACC in the USA and Piz Daint at CSCS in Switzerland. 82
Figure 54: Cube adapted mesh with respect to a radial anisotropic size-field, obtained with a 768 cores parallel mesh adaptation. Left: the final mesh with around 7.2 million elements. Right: partition represented by different colours. 83
Figure 55: Strong scaling test for anisotropic refinement of a mesh within a Cube, through two refinement steps the mesh size is multiplied by 2.25x. 84
Figure 56: Mesh of a nozzle (courtesy NASA Glenn Research Center) with a priori mesh size constraints, leading to about 420 million tetrahedra. Three zoom levels on a visualization of the mesh. 84
Figure 57: Parallel AMR workflow implemented in Alya. 86
Figure 58: Strong scaling of the AMR implementation of Alya in a tetrahedral mesh of 16M elements, using the Hawk supercomputer from HLRS. 86
Figure 59: 20M elements mesh of the Preccinsta burner obtained through adaptive mesh refinement from 3M elements mesh. Test case studied in the CoEC Center of Excellence. 88
Figure 60: NB-LIB interface and data flow. A series of pre-processing steps prepares user supplied data so that the NB-LIB interface is generic, accepting only elementary types. 91
Figure 61: MPI communication time for 100,000 particles MPI-OpenMP vs MPI-OpenACC ..101
Figure 62: Particle mover time for 100,000 particles single GPU vs multiple GPU 102
Figure 63: Vlasiator scaling on LUMI-C 103
Figure 64: Scaling of StruGePiC on LUMI-C 104
Figure 65: The trace of the execution of a generalized eigensolver for a 10240×10240 matrix. Above the algorithms are executed sequentially, below they are allowed to overlap. 105
Figure 66: Triangular solver on LUMI-EAP. We present the strong scaling for a matrix of size 40k. 106
Figure 67: Cholesky decomposition on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node). 106
Figure 68: Triangular solver on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node). 107
Figure 69: Transformation from generalized to standard eigenproblem on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node). 107
Figure 70: Strong scalability of DDalphaAMG-multiple rhs on IT4I system Karolina 108
Figure 71: librsb 1.3 vs librsb 1.2 109
Figure 72: Weak scaling of a QXContexts simulation on JUWELS Booster. As test cases, 1024 probability amplitudes per node were computed for both a 53 qubit rochester circuit and a 70 qubit bristlecone circuit. 110

Figure 73: Strong scaling of a QXContexts simulation on JUWELS Booster. A total of 65536 probability amplitudes of a 53 qubit rochester circuit were computed as a test case.....111
Figure 74: Strong scaling of a simulation within a single node on JUWELS Booster. As test cases, both 64 and 512 probability amplitudes were computed for a 49 qubit random quantum circuit with 32 layers of entangling gates.....111
Figure 75: Strong scaling results for a simulation of a 70 qubit circuit on a single node on JUWELS Booster. Results are shown for both the computation of 1024 and 2048 probability amplitudes. We observe close to ideal scaling with respect to the number of GPUs used on a single node.....112
Figure 76: Bi-directional communication bandwidth on LUMI-C (Slingshot 10, 100Gb/s)for messages of different sizes. Left: 1 thread, 1 message in-flight. Right: 16 threads, 10 messages in-flight 113
Figure 77: Weak scaling of HE with GHEX and native MPI on LUMI-C. Each rank handles double-precision fields of size 128^3. Left: 1 data field, halo width 1. Right: 5 data fields, halo width 5.....114
Figure 78: Weak scaling of BIFROST with GHEX and native MPI halo exchange on Betzy and LUMI-C. Single-precision, 64^3 grid points per rank 114
Figure 79: Strong scaling of Argo on LUMI-C@CSC (left) and Hawk@HLRS (right), it should be noted that the dashed lines refer to efficiency 116

List of Tables

Table 1: WP8 Project compatibility on EuroHPC systems (++ actual results obtained, + architecture supported, 0 untested architecture, - architecture unsupported).....3
Table 2: WP8 Project compatibility on PRACE Tier-0 systems (++ actual results obtained, + architecture supported, 0 untested architecture, - architecture unsupported).....5
Table 3: Hardware configuration of JSC supercomputers benchmarks performed on 9
Table 4: Results for Vlasiator scalability on Mahti. The propagation time does not include initialization, IO, and load balancing. 18
Table 5: Computation of the harmonic analysis with combined spatial-frequency domain parallelization. Solution of the system using iterative solver in [s]. 37
Table 6: Overview of the major achievements of the project in software development.....52
Table 7: Key features of the different architectures evaluated on BEAST 65
Table 8: Time to compute various quantum circuits on Marconi100 system 66
Table 9: Scaling test for the generation of a 420M elements mesh of a nozzle. Parallelization base on threading using OpenMP on an AMD Epyc Rome 7542 CPUs at 2.9 GHz. 85
Table 10: Comparison of Alya SFC-based mesh partitioning vs Zoltan v 3.8.3. Partition of a 250M elements mesh around an airplane. 87
Table 11: Benchmark performance of NB-LIB vs GROMACS using a system of 157464 Argon atoms 95
Table 12: Vlasiator scalability on LUMI-C 103
Table 13: Strong scaling of Argo on AMD Rome architecture with 4 threads per MPI and 32 MPI per node: LUMI-C@CSC (left) and Hawk@HLRS (right). 115

References and Applicable Documents

[1] https://www.lumi-supercomputer.eu/lumi-one-of-the-worlds-mightiest-supercomputers/

[2] https://www.cineca.it/en/hot-topics/Leonardo-announce

[3] https://eurohpc-ju.europa.eu/news/deucalion-new-eurohpc-world-class-green-supercomputer-portugal

[4] https://siliconcanals.com/news/meet-vega-eurohpcs-first-supercomputer/

[5] https://sofiatech.bg/en/petascale-supercomputer/

[6] https://www.cesnet.cz/wp-content/uploads/2021/04/branislav_jansik_CESNET25-1.pdf

[7] https://luxprovide.lu/technical-structure/

[8] https://www.cscs.ch/computers/piz-daint/

[9] https://apps.fz-juelich.de/jsc/hps/juwels/configuration.html/

[10] https://www.hpc.cineca.it/hardware/marconi100

[11] https://www.bsc.es/support/MareNostrum4-ug.pdf

[12] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG

[13] http://www-hpc.cea.fr/en/complexe/tgcc-JoliotCurie.htm

[14] https://www.hlrs.de/systems/hpe-apollo-hawk/

[15] https://prace-ri.eu/hpc-access/hpc-systems/

[16] https://gitlab.jsc.fz-juelich.de/SLPP/epoch/test-cases

[17] I. Vasileska, P. Tomsic and L. Kos, Modernization of the PIC codes for exascale plasma simulation, 43rd International Convention on Information, Communication and Electronic Technology MIPRO 2020 https://doi.org/10.23919/MIPRO48935.2020.9245299

[18] I. Vasileska, L. Bogdanovic and L. Kos, Particle-in-Cell Code for GPU Systems, 44rd International Convention on Information, Communication and Electronic Technology MIPRO 2021, in press

[19] I. Vasileska, Task based parallelisation of the Particle-In-Cell codes, ASHPS (First Austrian-Slovenian HPC meeting) https://ashpc21.si/booklet-of-abstracts/#dearflip-df_2168/

[20] I. Vasileska, Plasma physics simulations with PIC codes, Autumn PRACE School 2020 https://events.prace-ri.eu/event/1049/timetable/

[21] P. Gibbon, U. Sinha, D. Brömmel, P. Otte, Z. Chitgar and J. Chew, Exascaling Strategies for the EPOCH Community Code, invited talk at the 40th Hirschegg Meeting on High-Energy Density Physics with Intense Ion and Laser Beams (26th January - 1st February, 2020): special session on PIC codes. https://indico.gsi.de/event/8925/attachments/27830/34747/Hirschegg_2020_Program_V5.pdf

[22] Palmroth et al., 2018, Vlasov methods in space physics and astrophysics, Living Reviews in Computational Astrophysics, https://doi.org/10.1007/s41115-018-0003-2

[23] github.com/fmihpc/vlasiator

[24] J. Xiao and H. Qin, Explicit structure-preserving geometric particle-in-cell algorithm in curvilinear orthogonal coordinate systems and its applications to whole-device 6D kinetic simulations of tokamak physics, Plasma Sci. Technol. 2021, https://doi.org/10.1088/2058-6272/abf125

[25] Kevin Sala, Alejandro Rico and Vicenç Beltran, Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications. CLUSTER 2020: 314-325 10.1109/CLUSTER49012.2020.00042

[26] Marcos Maroñas, Xavier Teruel, J. Mark Bull, Eduard Ayguadé, and Vicenç Beltran, Evaluating Worksharing Tasks on Distributed Environments. CLUSTER 2020: 69-80 https://doi.org/10.1109/CLUSTER49012.2020.00017

[27] Kevin Sala, Sandra Macià and Vicenç Beltran, Combining One-Sided Communications with Task-Based Programming Models, CLUSTER 2021, in press.

[28] https://github.com/It4innovations/espreso/

[29] Riha L., Brzobohaty T., Markopoulos A., Meca O. and Kozubek T. Massively Parallel Hybrid Total FETI (HTFETI) Solver. In: Platform for Advanced Scientific Computing Conference, PASC. ACM. 2016 https://pasc16.pasc-conference.org/program/index-of-contributors/

[30] http://www.msca-expertise.eu

[31] https://www.it4i.cz/en/welcome-to-the-national-competence-center-in-hpc

[32] www.dihostrava.cz/en

[33] http://www.netlib.org/scalapack

[34] https://github.com/STEllAR-GROUP/hpx

[35] https://en.cppreference.com/w/cpp/thread/future

[36] ACM Trans. Math. Softw. 45, 2, Article 21. doi: 10.1145/3313828

[37] Comp. Phys. Comm. 267 (2021) 108081 doi.org/10.1016/j.cpc.2021.108081

[38] https://wg21.link/p2300

[39] SIAM J. Sci. Comput., 43(4), A2660–A2684. doi: 10.1137/20M1313933

[40] https://github.com/Lyncs-API

[41] https://gitlab.inria.fr/solverstack/fabulous

[42] https://github.com/sy3394/DDalphaAMG/tree/multirhs

[43] http://librsb.sourceforge.net/

[44] https://github.com/michelemartone/pyrsb

[45] https://octave.sourceforge.io/sparsersb/

[46] https://gitlab.inria.fr/solverstack/maphys/maphyspp

[47] W. T. L. P. Lavrijsen and A. Dutta, "High-Performance Python-C++ Bindings with PyPy and Cling," 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2016, pp. 27-35, doi: 10.1109/PyHPC.2016.008

[48] https://doi.org/10.25080/majora-1b6fd038-00e

[49] Villalonga, et al., "Establishing the Quantum Supremacy Frontier with a 281 Pflop/s Simulation", Quantum Sci. and Tech., 5 034003, (2020), https://doi.org/10.1088/2058-9565/ab7eeb

[50] Bridgeman and Chubb, "Hand-waving and interpretive dance: an introductory course on tensor networks." Journal of Physics A: Mathematical and Theoretical, 50(22):223001, (2017), https://doi.org/10.1088/1751-8121/aa6dc3

[51] Biamonte and Bergholm, "Tensor networks in a nutshell," 2017, arXiv:1708.00006

[52] https://github.com/JuliaQX

[53] Bezanson et al., "Julia: A Fresh Approach to Numerical Computing". SIAM Review, 59(1):65-98, (2017). Publisher: Society for Industrial and Applied Mathematics, https://doi.org/10.1137/141000671

[54] https://hpc.fau.de/research/tools/likwid

[55] Boixo et al., "Characterizing quantum supremacy in near-term devices". Nature Phys 14, 595–600 (2018). https://doi.org/10.1038/s41567-018-0124-x

[56] Arute et al., "Quantum supremacy using a programmable superconducting processor". Nature 574, 505–510 (2019) https://doi.org/10.1038/s41586-019-1666-5

[57] JuliaCon Virtual Poster "Distributed Quantum Circuit Simulation" https://www.youtube.com/watch?v=IuZ2b-b4baY&ab_channel=TheJuliaProgrammingLanguage

[58] JuliaCon Virtual Poster "Introducing QXGraphDecompositions" https://www.youtube.com/watch?v=h6FeH4krtJY&t=4s&ab_channel=TheJuliaProgrammingLanguage

[59] https://openhpc.github.io/cloudwg/tutorials/sc20/exercise4.html

[60] https://fosdem.org/2021/schedule/event/containerized_hpc

[61] https://github.com/QuantumBFS/Yao.jl

[62] https://github.com/JuliaQX/YaoQX.jl

[63] Pednault et al., "Pareto-Efficient Quantum Circuit Simulation Using Tensor Contraction Deferral", http://arxiv.org/abs/1710.05867

[64] R. Shutski et al., "Simple heuristics for efficient parallel tensor contraction and quantum circuit simulation". Phys. Rev. A 102, 062614, (2020). Doi: https://doi.org/10.1103/PhysRevA.102.062614

[65] Cupjin Huang et al., "Classical Simulation of Quantum Supremacy Circuits" arXiv:2005.06787, 2020

[66] Pan, Feng and Zhang, Pan "Simulating the Sycamore quantum supremacy circuits" https://arxiv.org/abs/2103.03074

[67] https://github.com/boeschf/hwmalloc

[68] https://github.com/NordicHPC/hwcart

[69] https://github.com/boeschf/oomph

[70] https://github.com/GridTools/GHEX

[71] https://github.com/GridTools/ghexbench

[72] https://gridtools.github.io/gridtools/latest/index.html,

[73] SIGMA2 LUMI system researcher meeting, 31 March - 1st April 2020, https://www.sigma2.no/lumi-system-researcher-preparation#program

[74] Fifth Workshop on Programming Abstractions for Data Locality, September 9-11, 2019, https://sites.google.com/a/lbl.gov/padal-workshop/padal19

[75] https://gitlab.com/bsc-alya/alya/-/wikis/home

[76] https://gitlab.com/rickbp/gempa

[77] https://arxiv.org/abs/2109.03592v2

[78] https://gmsh.info

[79] https://www.excellerat.eu

[80] https://coec-project.eu/
[81] https://cordis.europa.eu/project/id/956104/es
[82] https://prace-ri.eu/news-media/publications/software-strategy-for-european-exascale-systems/
[83] https://www.top500.org/lists/top500/2022/06/
[84] https://github.com/ngnrsaa/qflex/tree/master/config/circuits

List of Acronyms and Abbreviations

aisbl	Association International Sans But Lucratif (legal form of the PRACE-RI)
AMR	Adaptive-mesh refinement
BETI	Boundary element tearing and interconnecting
CoE	Centre of Excellence
CPU	Central Processing Unit
CHASE	Chebyshev Accelerated Subspace iteration eigensolver
CUDA	Compute Unified Device Architecture (NVIDIA)
DCCRG	Distributed Cartesian cell refinable grid
DoA	Description of Action (formerly known as DoW)
EC	European Commission
EuroHPC	European High-Performance Computing Joint Undertaking
FETI	Finite element tearing and interconnecting
FMM	Fast-multipole method
GASPI	Global Address Space Programming Interface
GB	Giga (= 2^{30} ~ 10^9) Bytes (= 8 bits), also GByte
Gb/ s	Giga (= 10^9) bits per second, also Gbit/s
GB/ s	Giga (= 10^9) Bytes (= 8 bits) per second, also GByte/s
GFlop/s	Giga (= 10^9) Floating point operations (usually in 64-bit, i.e. DP) per second, also GF/s
GHz	Giga (= 10^9) Hertz, frequency =10^9 periods or clock cycles per second
GPU	Graphic Processing Unit
HPC	High Performance Computing; Computing at a high performance level at any given time; often used synonym with Supercomputing
HPL	High Performance LINPACK
KB	Kilo (= 2^{10} ~10^3) Bytes (= 8 bits), also KByte
LINPACK	Software library for Linear Algebra
MB	Management Board (highest decision making body of the project)

MB	Mega (= 2^{20} ~ 10^6) Bytes (= 8 bits), also MByte
MB/ s	Mega (= 10^6) Bytes (= 8 bits) per second, also MByte/s
MFlop/s	Mega (= 10^6) Floating point operations (usually in 64-bit, i.e. DP) per second, also MF/s
MoU	Memorandum of Understanding.
MPI	Message Passing Interface
NIH	US National Institutes of Health
PFC	Plasma-facing component
PIC	Particle-in-cell
PM	Person-month
PRACE	Partnership for Advanced Computing in Europe; Project Acronym
QCD	Quantum chromodynamics
RI	Research Infrastructure
SIMD	Single instruction multiple data
SOL	Scrape-off layer
SPMD	Single program multiple data
SSC	Scientific Steering Committee
SVD	Singular value decomposition
TAMPI	Task-aware MPI
TAGASPI	Task-aware GASPI
TB	Tera (= 2^{40} ~ 10^{12}) Bytes (= 8 bits), also TByte
TFlop/s	Tera (= 10^{12}) Floating-point operations (usually in 64-bit, i.e. DP) per second, also TF/s
Tier-0	Denotes the apex of a conceptual pyramid of HPC systems. In this context the Supercomputing Research Infrastructure would host the Tier-0 systems; national or topical HPC centres would constitute Tier-1

List of Project Partner Acronyms

BADW-LRZ	Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Germany (3rd Party to GCS)
BILKENT	Bilkent University, Turkey (3rd Party to UHEM)
BSC	Barcelona Supercomputing Center - Centro Nacional de Supercomputacion, Spain
CaSToRC	The Computation-based Science and Technology Research Center (CaSToRC), The Cyprus Institute, Cyprus
CCSAS	Computing Centre of the Slovak Academy of Sciences, Slovakia
CEA	Commissariat à l'Energie Atomique et aux Energies Alternatives, France (3rd Party to GENCI)
CENAERO	Centre de Recherche en Aéronautique ASBL, Belgium (3rd Party to UANTWERPEN)
CESGA	Fundacion Publica Gallega Centro Tecnológico de Supercomputación de Galicia, Spain, (3rd Party to BSC)
CINECA	CINECA Consorzio Interuniversitario, Italy
CINES	Centre Informatique National de l'Enseignement Supérieur, France (3 rd Party to GENCI)
CNRS	Centre National de la Recherche Scientifique, France (3 rd Party to GENCI)
CSC	CSC Scientific Computing Ltd., Finland
CSIC	Spanish Council for Scientific Research (3rd Party to BSC)
CYFRONET	Academic Computing Centre CYFRONET AGH, Poland (3rd Party to PNSC)
DTU	Technical University of Denmark (3rd Party of UCPH)
EPCC	EPCC at The University of Edinburgh, UK
EUDAT	EUDAT OY
ETH Zurich (CSCS)	Eidgenössische Technische Hochschule Zürich – CSCS, Switzerland
GCS	Gauss Centre for Supercomputing e.V., Germany
GÉANT	GÉANT Vereniging
GENCI	Grand Equipement National de Calcul Intensif, France
GRNET	National Infrastructures for Research and Technology, Greece
ICREA	Catalan Institution for Research and Advanced Studies (3rd Party to BSC)
INRIA	Institut National de Recherche en Informatique et Automatique, France (3rd Party to GENCI)
IST-ID	Instituto Superior Técnico for Research and Development, Portugal (3rd Party to UC-LCA)

IT4I	Vysoka Skola Banska - Technicka Univerzita Ostrava, Czech Republic
IUCC	Machba - Inter University Computation Centre, Israel
JUELICH	Forschungszentrum Juelich GmbH, Germany
KIFÜ (NIIFI)	Governmental Information Technology Development Agency, Hungary
KTH	Royal Institute of Technology, Sweden (3rd Party to SNIC-UU)
KULEUVEN	Katholieke Universiteit Leuven, Belgium (3rd Party to UANTWERPEN)
LiU	Linkoping University, Sweden (3rd Party to SNIC-UU)
MPCDF	Max Planck Gesellschaft zur Förderung der Wissenschaften e.V., Germany (3rd Party to GCS)
NCSA	NATIONAL CENTRE FOR SUPERCOMPUTING APPLICATIONS, Bulgaria
NTNU	The Norwegian University of Science and Technology, Norway (3rd Party to SIGMA2)
NUI-Galway	National University of Ireland Galway, Ireland
PRACE	Partnership for Advanced Computing in Europe aisbl, Belgium
PSNC	Poznan Supercomputing and Networking Center, Poland
SDU	University of Southern Denmark (3rd Party to UCPH)
SIGMA2	UNINETT Sigma2 AS, Norway
SNIC-UU	Uppsala Universitet, Sweden
STFC	Science and Technology Facilities Council, UK (3rd Party to UEDIN)
SURF	SURF is the collaborative organisation for ICT in Dutch education and research
TASK	Politechnika Gdańska (3rd Party to PNSC)
TU Wien	Technische Universität Wien, Austria
UANTWERPEN	Universiteit Antwerpen, Belgium
UC-LCA	Universidade de Coimbra, Labotatório de Computação Avançada, Portugal
UCPH	Københavns Universitet, Denmark
UEDIN	The University of Edinburgh
UHEM	Istanbul Technical University, Ayazaga Campus, Turkey
UIBK	Universität Innsbruck, Austria (3rd Party to TU Wien)
UiO	University of Oslo, Norway (3rd Party to SIGMA2)
UL	UNIVERZA V LJUBLJANI, Slovenia
ULIEGE	Université de Liège; Belgium (3rd Party to UANTWERPEN)
U Luxembourg	University of Luxembourg
UM	Universidade do Minho, Portugal, (3rd Party to UC-LCA)

UmU	Umea University, Sweden (3rd Party to SNIC-UU)
UnivEvora	Universidade de Évora, Portugal (3rd Party to UC-LCA)
UnivPorto	Universidade do Porto, Portugal (3rd Party to UC-LCA)
UPC	Universitat Politècnica de Catalunya, Spain (3rd Party to BSC)
USTUTT-HLRS	Universitaet Stuttgart – HLRS, Germany (3rd Party to GCS)
WCSS	Politechnika Wroclawska, Poland (3rd Party to PNSC)

Executive Summary

From April 2019 to June 2022, Work Package 8 of PRACE-6IP oversaw a number of projects developing forward-looking software solutions. These projects were selected following two competitive calls for proposals; eight started in April 2019 following the first call for proposals, and two additional projects started in January 2020 following the second call for proposals. Eight of the projects were active, albeit with a reduced level of activity, during the WP8 extension period, which ran from November 2021 to June 2022. This deliverable builds upon the previous deliverable D8.3, that provided an overview of the first phase of the project, and deliverable D8.4, that provided an overview of the second phase of the project. A comprehensive overview is provided of the progress of the projects during WP8, including a summary of the compatibility of the WP8 projects on the leading European HPC systems, as well as individual sections about each of the projects with 1) a summary of each projects and the goals 2) benchmarking results on leading HPC systems 3) engagement with stakeholders and outreach activities and 4) an overall assessment of achievements and areas for future developments. A separate annex is included that provides an overview of the performance and benchmarking results obtained by the eight projects that were active during the WP8 extension period. Overall, it is fair to say that the projects have carried out a thorough level of work, and have contributed to the success of Work Package 8 of PRACE-6IP.

1 Introduction

From April 2019 to June 2022, Work Package 8 (WP8) of PRACE-6IP focused on 'Forward-looking Software Solutions', with the overall objective of delivering high quality, transversal software that addresses the challenge posed by the rapidly changing HPC pre-exascale landscape. In addition to the challenges posed by the diversity of hardware and software complexity, the late delivery of the leading pre-exascale EuroHPC systems, due to procurement issues as a result of the pandemic, resulted in limited access to this latest generation of HPC systems. Despite this, the projects achieved good scaling results across a diverse range of HPC systems in Europe, including PRACE Tier-0 and EuroHPC machines, that included a broad range of HPC architectures.

During the course of WP8, ten projects were chosen, as a result of two competitive, peer reviewed calls, as reported on in deliverables D8.1 and D8.2. Eight projects were funded from the beginning of PRACE-6IP in April 2019, and a further two began in January 2020 following the second competitive call. The projects in WP8 covered a wide range of scientific domains, from fundamental topics such as tasking runtimes, halo-exchange libraries, to mathematical libraries including sparse and dense linear algebra, to application domain related software targeted at science and engineering like plasma physics, biophysics, finite elements, and fluid dynamics, or emerging domains such as quantum computing.

Each of the ten WP8 projects worked autonomously, in accordance with the development roadmaps that they included as part of the project proposals that were submitted to the competitive peer-reviewed calls. Two earlier deliverables, D8.3 and D8.4, have helped to contribute to the sustainability and quality of the software. Namely, in deliverable D8.3, a report on a public prototype release of the software was provided, as well as details on the development infrastructure used. This early release helped to ensure that software sustainability was taken into serious consideration, using industry standard tools, issue tracking, continuous integration, validation and verification, documentation, etc. Deliverable D8.4 then went one step further and documented a production-quality software release, with the aim of bringing this software into the hands of users for the European HPC infrastructure, and provided information on the availability and performance of the software, in addition to outreach efforts carried out by the projects.

This deliverable, D8.5, is the final report from WP8. The document presents an overview of the compatibility of the ten projects on the leading European HPC systems, including the EuroHPC systems and the PRACE Tier-0 systems. Following that, it is arranged per project. A brief introduction and summary is provided for each project, followed by benchmark results on leading European HPC systems, an overview of stakeholder engagement and outreach, and then concludes with an assessment of the achievements of each project and directions for future developments. Finally, an annex with an overview of the performance and benchmarking results of the eight projects active during the WP8 extension is included, summarising the work carried out between November 2021 and June 2022.

2 Compatibility of PRACE-6IP WP8 projects on leading European HPC systems

In the following section, an overview is presented of the compatibility of the WP8 projects on the leading European HPC systems. Table 1 gives an overview of the EuroHPC systems, whilst Table 2 gives an overview of the PRACE Tier-0 systems.

		EuroHPC Pre-exascale systems			**EuroHPC Petascale systems**				
		LUMI[1]	Leonardo[2]	MareNostrum5[3]	Deucalion[4]	Vega[5]	Discoverer[6]	Karolina[7]	MeluXina[8]
Node Architecture[a]		CPU:AMD GPU: AMD	CPU:INT GPU: NVD	CPU: TBD GPU: TBD	CPU: FJU GPU: N/A	CPU:AMD GPU:NVD	CPU: AMD GPU: N/A	CPU: AMD GPU:NVD	CPU: AMD GPU: NVD
Project	**Code**			**0**					
PiCKeX	EPOCH	-	-	**0**	+	-	+	-	-
PiCKeX	OOPD1	+	+	**0**	-	+	-	+	++
MoPHa	GENE/tasks	**0**	+	**0**	**0**	+	+	+	+
MoPHa	SIMPIC	**0**	+	**0**	-	+	+	+	+
MoPHa	StruGePiC	++	+	**0**	-	+	+	+	+
MoPHa	SymPiFE-VMax	+	+	**0**	-	+	+	+	+
MoPHa	Vlasiator	++	**0**	**0**	-		+		
NB-LIB		**0**	+	**0**	**0**	+	+	+	+
LoSync		+	+	**0**	+	+	+		+
LyNcs		+	+	**0**	+	+	+	++	+
GHEX		++	+	**0**	+	+	+	+	+
PPLA	DLA-Future	+	+	**0**	**0**	+	+	+	+
PPLA	ChASE	**0**	+	**0**	**0**	+	+		+
FEM/BEM		+	+	**0**	**0**	+	+	++	+
ParSec		+	+	**0**	+	+	+	+	+
QuantEx		**0**	+	**0**	+	+	+	+	+

Table 1: WP8 Project compatibility on EuroHPC systems (++ actual results obtained, + architecture supported, 0 untested architecture, - architecture unsupported)

[a]Node Architecture: AMD - AMD, INT - Intel, NVD - NVIDIA, FJU - Fujitsu A64FXs, TBD - to be decided, N/A - not applicable

[1]LUMI pre-exascale supercomputer, located in Kajaani Finland, peak performance 550 petaflops. Architecture: CPUs - AMD EPYC, GPUS - AMD Instinct [1]

[2]Leonardo pre-exascale supercomputer, located in Bologna Italy, peak performance 250 petaflops. Architecture: CPUs - Intel, GPUs - NVIDIA Tensor Core [2]

[3]MareNostrum5 pre-exascale supercomputer, located in Barcelona Spain, peak performance 11 petaflops. Architecture: TDB.

[4]Deucalion petascale supercomputer, located in Minho Portugal, peak performance 10 petaflops. Architecture: CPUs - Fujitsu A64FXs [3]

[5]Vega petascale supercomputer, located in Maribor Slovenia, peak performance 10 petaflops. Architecture: CPUs - AMD EPYC, GPUs - NVIDIA A100 [4]

[6]Discoverer petascale supercomputer, located in Sofia Bulgaria, peak performance 6 petaflops. Architecture: CPUs - AMD EPYC [5]

[7]Karolina petascale supercomputer, located in Ostrava Czech Republic, peak performance 15 petaflops. CPUs- AMD EPYC, GPUs - NVIDIA A100 [6]

[8]MeluXina petascale supercomputer, located in Bissen Luxembourg, peak performance 10 petaflops. Architecture: CPUs - AMD EPYC, GPUs - NVIDIA A100 [7]

		PRACE Tier-0 systems									
		PizDaint[9]	JUWELS Cluster[10a]	JUWELS Booster[10b]	Marconi 100[11]	Mare Nostrum4[12]	SuperMUC[13]	Joliot-Curie KNL[14a]	Joliot-Curie Rome[14b]	Joliot-Curie SKL[14c]	Hawk[15]
Node Architecture[b]		CPU: INT GPU: NVD	CPU: INT GPU: NVD	CPU: AMD GPU: NVD	CPU: IBP GPU: NVD	CPU: INT GPU: N/A	CPU: INT GPU: N/A	CPU: INT GPU: N/A	CPU: AMD GPU: N/A	CPU: INT GPU: N/A	CPU: AMD GPU: NVD
Project	**Code**										
PiCKeX	EPOCH	-	++	-	-	+	+	+	+	+	+
PiCKeX	OOPD1	+	-	+	++	-	-	-	-	-	+
MoPHa	GENE/tasks	+	+	+	+	+	+	+	+	+	+
MoPHa	SIMPIC	+	+	+	++	+	+	+	+	+	+
MoPHa	StruGePiC	+	+	+	**0**	+	+	+	+	+	+
MoPHa	SymPiFE-VMax	+	+	+	**0**	+	+	+	+	+	+
MoPHa	Vlasiator	+				+	+		+	+	++
NB-LIB		++	+	+	+	+	+	+	+		+
LoSync		+		+		++	+	+			+
LyNcs		+	++	++	+	+	++	+	+	+	++
GHEX		++	+	+	++	+	+	-	+	+	+
PPLA	DLA-Future	++	+	+	++	+	+	**0**	+	+	+
PPLA	ChASE	+	++	++	**0**	+	+	**0**	+	+	+
FEM/BEM		+	++	++	**0**	+	+	+	+	+	+
ParSec		++	++	++	+	++	+	+	+	+	++
QuantEx		+	++	++	++	+	++	+	+	+	+

Table 2: WP8 Project compatibility on PRACE Tier-0 systems (++ actual results obtained, + architecture supported, 0 untested architecture, - architecture unsupported)

[b]Node Architecture: AMD - AMD, INT - Intel, NVD - NVIDIA, N/A - not applicable, IBP - IBM Power9

[9]Piz Daint PRACE Tier-0 supercomputer, located in Lugano Switzerland, peak performance 27 petaflops. Architecture: CPUs - Intel Xeon, GPUs - NVIDIA P100 [8]

[10a]JUWELS Cluster PRACE Tier-0 supercomputer, located in Jülich Germany, peak performance 12 petaflops. Architecture: CPUs - Intel Xeon, GPUs - NVIDIA V100 [9]

[10b]JUWELS Booster PRACE Tier-0 supercomputer, located in Jülich Germany, peak performance 71 petaflops. Architecture: CPUs - AMD EPYC, GPUs - NVIDIA A100 [9]

[11]Marconi100 PRACE Tier-0 supercomputer, located in Bologna Italy, peak performance 32 petaflops. Architecture: CPUs - IBM Power9, GPUs - NVIDIA V100 [10]

[12]MareNostrum4 PRACE Tier-0 supercomputer, located in Barcelona Spain, peak performance 11 petaflops. Architecture: CPUs - Intel Xeon E5 [11]

[13]SuperMUC-NG PRACE Tier-0 supercomputer, located in Garching Germany, peak performance 27 petaflops. Architecture: CPUs - Intel Xeon Skylake [12]

[14a]Joliot-Curie KNL PRACE Tier-0 supercomputer, located in Paris France, peak performance 2 petaflops. Architecture: CPUs - Intel KNL [13]

[14b]Joliot-Curie PRACE Tier-0 supercomputer, located in Paris France, peak performance 12 petaflops. Architecture: CPUs - AMD EPYC, [13]

[14c]Joliot-Curie SKL PRACE Tier-0 supercomputer, located in Paris France, peak performance 6 petaflops. Architecture: CPUs - Intel Skylake [13]

[15]Hawk PRACE Tier-0 supercomputer, located in Stuttgart Germany, peak performance 26 petaflops. Architecture: CPUs - AMD EPYC, GPUs - NVIDIA A100 [14]

Overview of PRACE Tier-0 systems: [15]

3 PiCKeX: Particle Kinetic codes for Exascale plasma simulation

3.1 Introduction and summary

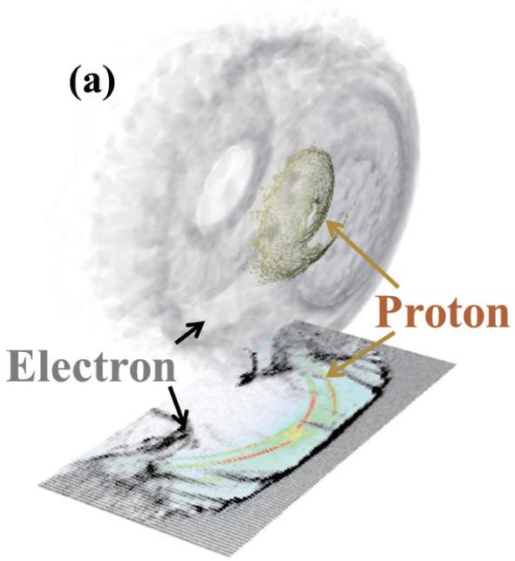


Figure 1: Proton acceleration by circularly polarized, multi-petawatt laser pulse

Particle in Cell (PIC) codes have become one of the main tools for many areas in plasma physics, for example in modelling laser-accelerated particle beams (Figure 1), studies which are instrumental in the development of compact particle GeV-class accelerators, or to understand the detailed dynamics and transport processes near the outer scrape-off layer (SOL) of nuclear fusion vessels containing a magnetically confined plasma. The PicKeX project focuses on two important community codes: EPOCH, a fully relativistic, electromagnetic model and BIT1, a sophisticated PIC/Monte-Carlo model.

For both codes the project has enabled substantial refactoring work to be performed which would have been difficult to realise for a conventional research team utilising the code for scientific investigation. As a result, enhanced versions of both codes are now publicly available for rigorous testing by user groups. In particular, this includes OOPD1, a new GPU version of BIT1. The new version of EPOCH incorporates a significantly faster moving window algorithm - up to 40% on thousands of cores, which is extensively used for an important class of problems based on laser-based particle accelerator schemes. The BIT1 and OOPD1 codes have two main algorithms: particle mover, which updates position and velocities of the simulated “super” particles according to the well-known Newton's laws of motion, and the field solver which calculates the fields inside the simulated spatial region at some grid points. In the new version of the code a fully GPU version of the particle mover was introduced also giving performance improvements of 40%.

3.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

The JUBE benchmarking environment allows the user to perform complete testing workflows - compilation, configuration, execution and verification - from a single script. The data is written in a format such that the desired information can be obtained automatically using either pre- or post-processing scripts. We have created a JUBE script-based framework comprising various test cases which can be used to test EPOCH on different petascale/pre-Exascale systems and evaluate the results [16]. The tests can be performed using any desired simulation domain, compiler environments and allow direct comparison of multiple versions of the code. Additional tests can be added to this framework as needed.

In most laser-based accelerator schemes, the physics of interest lies in the region surrounding the laser or the particle beam moving close to the speed of light. For this widely considered class of applications it is prudent to make use of a "moving window" algorithm, such that the simulation follows the region of interest and does not simulate the entire system. The moving window approach keeps the mesh stationary with respect to the background, creating new particles and fields at the leading edge, shifting the particles and fields to neighbouring mesh points, and discarding any particles and fields in the trailing edge. If *dx* is the grid size and *dt* is the timestep, then the Courant condition for an electromagnetic particle-in-cell code requires that $cdt < dx$, which limits the maximum displacement of the simulation window per timestep to be less than dx. The moving window algorithm shifts the simulation window by integral multiples of the grid size (dx) which requires the algorithm to be invoked after n timesteps such that ndt > dx where n>1.

EPOCH is a parallel, pure-MPI code written using simple cartesian domain decomposition. The entire simulation window is decomposed into small domains where each domain is handled by one MPI rank. After every timestep, each rank communicates with its neighbours to exchange particles and fields. To facilitate this exchange, the domains are surrounded by ghost cells. The number of ghost cells (ng) allocated in each direction depends on the particle shape with a minimum value of 4 for the top-hat shape and a maximum of 6 for the third-order b-spline interpolation. The field boundary conditions exchange the field information in the ghost cells between neighbouring ranks.

The initial version of the code invoked the moving window algorithm whenever the mesh was displaced by dx, each time requiring MPI calls to be made to communicate the field and particle boundary conditions to the neighbouring ranks. Furthermore, the field boundary conditions were invoked in all directions which is unnecessary, as the mesh is shifted only along the direction of propagation. Performance analysis of the code using the Score-P toolkit showed that up to 20% of the simulation time was spent in the moving window routine.

In the new version, we modified the moving window algorithm such that it is called only after the mesh has traversed ng cells and the boundary values of the field quantities are communicated only to the ranks in the propagation direction. These modifications significantly reduce the communication overhead due to the shifting of fields and particles and the number of MPI calls. In addition, we introduced a new data structure to communicate the field boundary values that enable us to send and receive all the three components of the fields using a single MPI_SendRecv call. This data structure reduces the latency effects. These modifications allow the new version of EPOCH to be ~40% faster than the original version.

Benchmarks were performed on the PRACE Tier-0 JUWELS Cluster and Tier-1 JURECA-DC supercomputers at the Jülich Supercomputing Center (JSC). These machines are CPU-based clusters and ideal for EPOCH as it employs pure MPI based parallelism. The hardware configuration of these systems is listed in Table 1. Although not yet explicitly tested, we would

expect similar speedup on the Intel- and AMD-based systems in the table, as well as future ARM-based systems.

	JUWELS	JURECA-DC
Number of nodes	2271	576
Processor	Intel Xeon Platinum 8168 CPU	AMD EPYC 7742 CPU
Node configuration	2 X 24 cores	2 X 64 cores
Clock speed	2.7 GHz	2.25 GHz
Memory	96 (12× 8) GB DDR4, 2666 MHz	512 (16× 32) GB DDR4, 3200 MHz
Network	InfiniBand EDR (Connect-X4)	InfiniBand HDR100 (NVIDIA Mellanox Connect-X6)

Table 3: Hardware configuration of JSC supercomputers benchmarks performed on

To obtain the scaling results, we carry out 2D simulations of a relativistic laser pulse moving through a uniformly distributed electron-proton plasma using the PIC code EPOCH. A simulation window moving with the laser pulse in the x-direction is used. The window is resolved using 25000 cells along the x-direction and 6400 cells along the y-direction with 10 particles per cell. The total time of simulation was 800 fs with a timestep dt = 0.1fs. During the first 100fs of the simulation, the laser was allowed to propagate through the plasma to its centre. After this time, the moving window was started and thus invoked for ~88% of the total run time. Figure 2 shows the strong scaling results on JUWELS (Figure 2a) and JURECA-DC (Figure 2b). The new version of EPOCH is ~30-45% faster than the initial version for this range of core numbers (Figure 3).

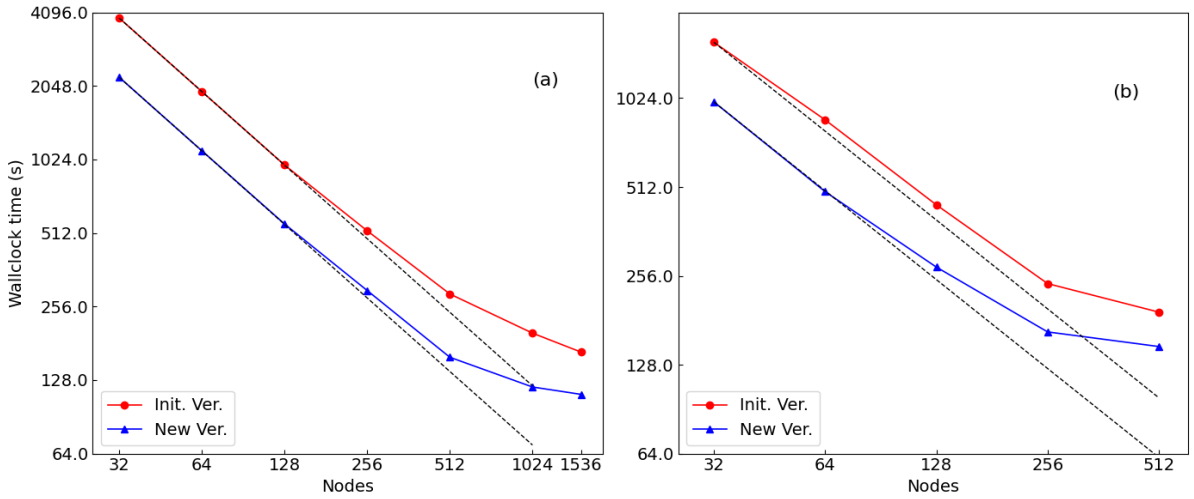


Figure 2: The strong scaling using the moving window algorithm with a simulation window comprising 25000 X 6400 cells with 10 particles per cell for 8000 timesteps. (a) Comparison of the strong scaling results between the initial and new version of EPOCH from 32 to 1536 nodes (1536 to 73728 CPU cores) on JUWELS. (b) Comparison of the strong scaling results between the initial and the new version of EPOCH from 32 to 512 nodes (4096 to 65536 CPU cores) on JURECA-DC.

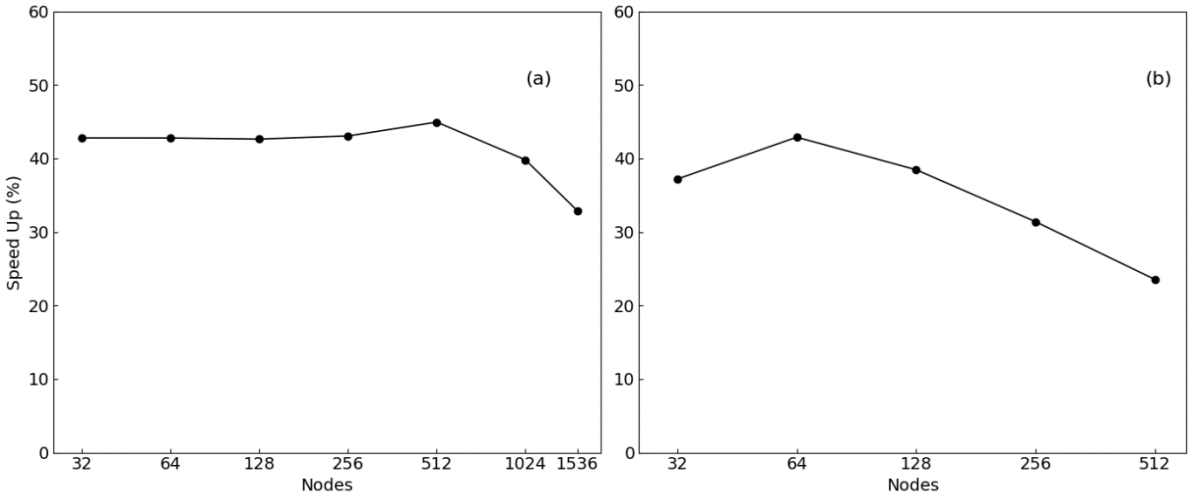


Figure 3: (a) Speed-Up obtained by the new version of EPOCH over the initial version from 32 to 1536 nodes (1536 to 73728 CPU cores) on JUWELS. (b) Speed-Up obtained by the new version of EPOCH over the initial version from 32 to 512 nodes (4096 to 65536 CPU cores) on JURECA-DC.

The OOPD1 (Objected Oriented Plasma Device 1D) code is a PIC code with a Monte Carlo algorithm for calculation of particle interactions and collisions. The code was run on the VIZ supercomputer at the Faculty of Mechanical Engineering in Ljubljana, which has 24 processor cores (2x 12-core Intel Xeon E5-2680V3 processor clocked at 2.5 GHz), 256 GB of DDR4 memory running at 2133 MHz with ECC, 3x NVIDIA Tesla K80 graphics interface and 250 GB of RAM. OpenMPI/3.1.4-GCC-8.3.0 compiler and tk/8.6.8/intel-18.0.2-kzhkvu6 with xgrafix graphical library were used for running interpretation of the results. The GPU optimisation of the fully particle mover was carried out as follows:

1. The code consists of a lot of objects and header files placed in one directory. For better organization first a code refactoring was done. All header files were divided in different subdirectories depending on the calculation functions. The source files written in C++ were in a separate directory. Based on this refactoring it was easier to optimize the code because the functions of the particle mover, field solver, Monte Carlo, particle cross sections and atomic database were organised.
2. A typical particle mover algorithm in the CPU version would be something like (Figure 4a):
    a. Gather field at particle position
    b. Calculate new velocity using field
    c. Calculate new position using new velocity

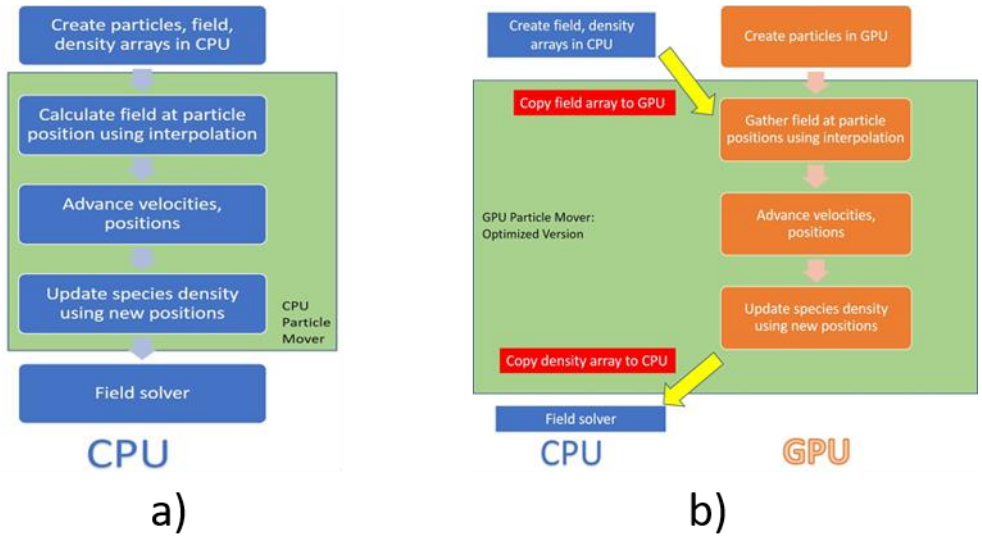


Figure 4: a) CPU version b) GPU version of the particle mover algorithm in OOPD1

For GPU optimisation we generate all particles in the GPU to avoid the memory transfer. Then all calculation functions were rewritten in the GPU version using CUDA programming. When the new position was calculated then all GPU outputs were copied back to the CPU version to start the field solver. For this reason, we called the new version half GPU optimised code.

3. As a test case for benchmarking we used a simple case where only electrons were used with an electrostatic (Poisson's equation) field solver (see Figure 5). The simulation geometry corresponds to a simple one dimensional case, with self and applied electric fields directed along coordinate x. There are no variations in y or z directions. The plane parallel problem consists of single (electron) species. The electrons were simulated between two electrodes. The applied voltage on the left hand side electrode is 25000 V, the right hand side electrode was grounded.

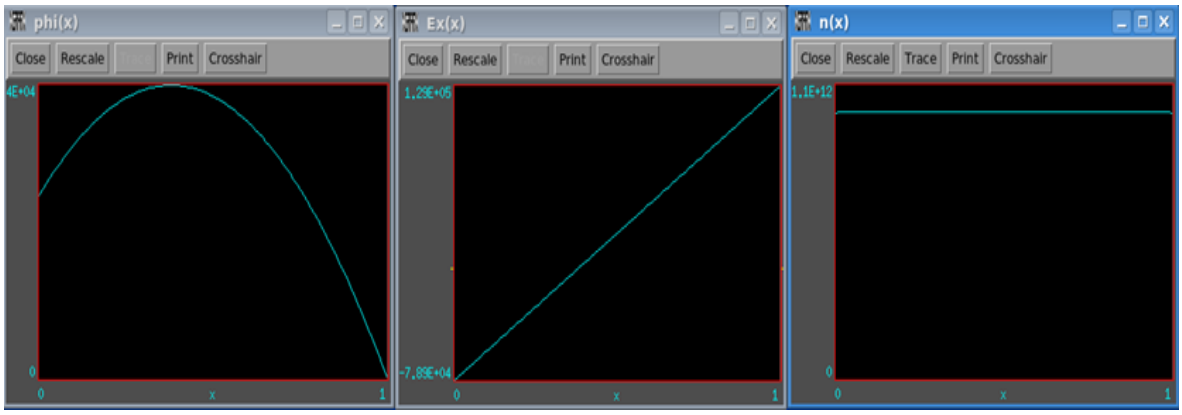


Figure 5: Test case in OOPD1

This case first was run on the CPU and then on the GPU version with fully optimized particle mover. For running this case we used 10000 grid points, 0.001 fractions on time, and we ran the case with 200 timesteps. The results of the benchmarking are performed in Figure 6

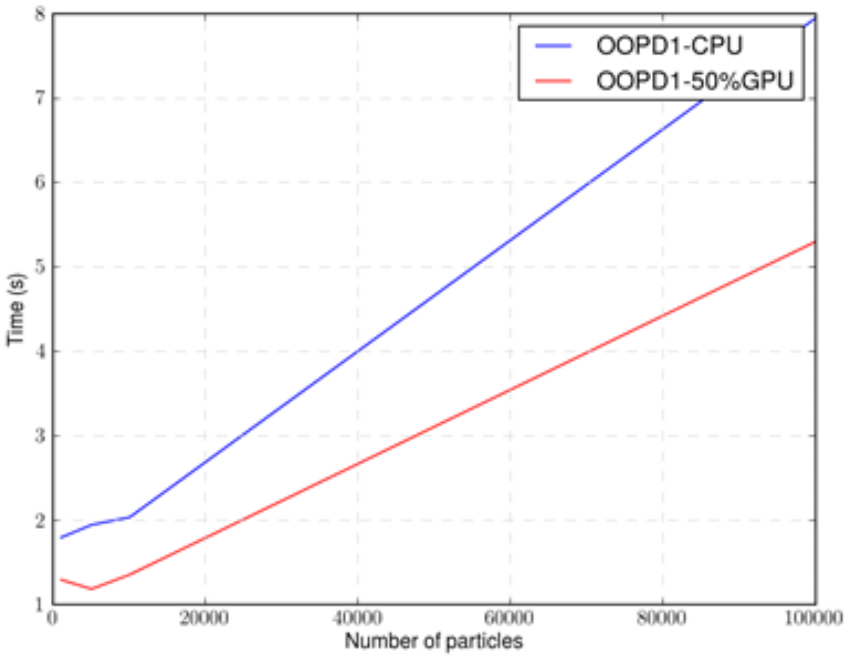


Figure 6: Benchmark of the CPU and full particle mover GPU version of OOPD1

From the results it can be concluded that with only a full particle mover in the GPU we have a nearly 40% reduced computation time compared to the CPU version. Because this case was very simple, only electrons without any collisions in the electrostatic field solver were simulated, and most of the complex functions were not used. In future the field solver will also be transferred to GPU, which will enable runs with more complex cases including ions and neutrals. One of the biggest problems that will occur are the particle cross section calculations and particle reactions. For that reason, heterogeneous computing with StarPU will be used.

The BIT1 code has structures similar to OOPD1. For that reason, the same steps, which were used for optimisation of the OOPD1 will be implemented in BIT1. At this moment only a BIT1 refactoring was done. With this refactoring the main algorithms were separated from the atomic molecular processes and cross sections. At this moment only the first steps of creating particles in the GPU were completed.

3.3 Interactions with stakeholders, users, outreach and publications

Regarding EPOCH, the primary interaction has been with the main developer group based at Warwick University, where the production version of the code is maintained. This repository has been mirrored at JSC to avoid undue interference with the physics module development. The optimisations achieved in the PicKeX project have been pushed to the main branch, where they are pending approval by the Warwick group. Early exchanges with this team made sure that any changes implemented by the project would be compatible with improvements to the main branch, but also that more radical improvements such as domain decomposition and load balancing would entail a fundamental re-engineering of the core code structures, which would have risked a fork

from the community version. A change of repositories at U. Warwick also caused delays in contributing our changes to the official version. At least one current PRACE project using the EPOCH code has been identified and contacted as a potential adopter of the new version. Finally, the JUBE-based benchmarking methodology introduced during the project is also publicly available and should generally benefit the user community.

In terms of dissemination, the Ljubljana group presented two conference publications at MIPRO 2020 [17] and 2021 [18]. The first publication describes how the GPU optimization was done in the prototype PIC code. The second publication describes the fully GPU field solver of OOPD1. Also we have participated in ASHPS (First Austrian-Slovenian HPC meeting) [19] and Autumn PRACE School in 2020 [20]. There we presented the GPU method for optimization on PIC codes. Intermediate results for the optimized EPOCH code were presented at a special session on PIC codes at the annual Hirschegg Meeting on High-Energy Density Physics [21].

3.4 Overall assessment of achievements and future developments

In summary, the PiCKeX project has resulted in two major milestones: first, significant speed-ups of up to 40% for the EPOCH code when applied to frequently studied laser-based particle accelerator schemes; and second, the creation of a new GPU-enabled version of the BIT1 code (OOPD1), which will serve as a basis for future developments of this important tokamak edge physics model.

Given that EPOCH is still a pure MPI code, more fundamental restructuring of its core elements would be necessary in order to implement an efficient hybrid OpenMP-MPI scheme, or to enable GPU capabilities. This was not realistic within the scope of the present project without risking a disconnected fork of the code which would have been difficult to reconcile with the production branch. The extensive analyses of the present version will however guide future consultation with the main developer group in Warwick on whether/how to proceed with further optimisations/restructuring, such as a more robust load-balancing scheme or better domain decomposition.

The OOPD1 GPU code developed within the project is structurally very close to the production BIT1 code, but currently lacking the collisional physics modules. Once the electrostatic field solver is ported to GPU, a possible route for full GPU-capability of BIT1 would be opened up with the help of the StarPU programming model for mixed CPU-GPU operation. The latter approach is likely to become a key design point in future modular exascale systems.

4 MoPHA: Modernisation of Plasma Physics Simulation Codes for Heterogeneous Exascale Architectures

4.1 Introduction and summary

Code modernisation efforts are needed for many scientific simulation codes to fully benefit from the upcoming heterogeneous exascale systems. This is true also for plasma simulation codes, such as ELMFIRE, GENE, and Vlasiator. Task-based parallelism potentially offers better scalability and portability than traditional approaches by abstracting hardware-specific optimisations away from the scientific algorithms. Some frameworks, such as StarPU or AMReX, even offer a relatively easy way to achieve both task-based parallelism and support for GPUs.

In the MoPHA project, we have explored task-based parallelism for plasma simulations and tested ways to add support for GPUs or other accelerators to plasma simulation codes, targeting the three codes ELMFIRE, GENE and Vlasiator. There are three questions to answer:

- How much work is the refactoring of the existing code base to make use of an existing framework like StarPU or AMReX?
- What is the performance of a task-based code compared to the original code?
- How portable is the performance to different upcoming heterogeneous architectures?

The ways to explore these questions have been different for the three codes, but the aim has been to pave the way for the plasma simulation codes to be ready for the upcoming pre-exascale and exascale systems.

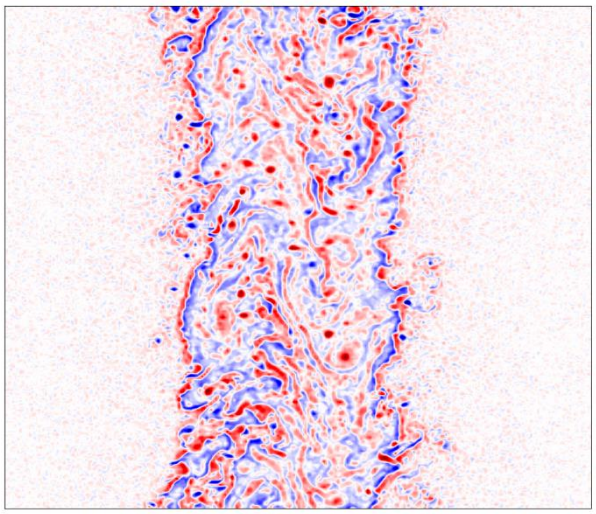


Figure 7: Turbulent flow in a fusion plasma simulation

GENE: Taking the large Fortran codebase and some GPU-ported code paths as a starting point, the introduction of StarPU showed to be a rather complex task. The complexity of StarPU combined with an additional Fortran layer, together with the class hierarchy of GENE made an adaptation a difficult challenge. We decided to focus on a single-node implementation of a single code path. We could show that different iterations of a loop rewritten as tasks can overlap with different physics computations of the right-hand side of the Vlasov equation. To really get an improvement and to answer the above mentioned second and third question, further work is needed.

ELMFIRE: Particle simulation in the field of nuclear fusion is a well-established technique which has spawned dozens of codes around the world through years (e.g. BIT1, VPIC, VSIM, OSIRIS, REMP, EPOCH, SMILEI, FBPIC, WARP, PEPC) with varying degrees of specialization for different physics areas and accessibility. Particle-in-cell (PIC) codes simulate numerous plasma phenomena on HPC systems. Today, flagship supercomputers feature GPUs per compute node to achieve unprecedented computing power at high power efficiency. PIC codes require new algorithm design and implementation for exploiting such accelerated platforms.

Major refactoring of the ELMFIRE code is needed to harness this power, as well as to expand its capabilities to simulate tokamak plasma accurately, including electromagnetic effects, realistic magnetic backgrounds (diverted), and realistic wall geometry. The limited amount of memory per cores available on GPUs also motivates the use of more local and explicit algorithms. To achieve these objectives, we have elected to rely on scalable and versatile frameworks for mesh-based simulations, as well as recent developments in structure-preserving algorithms. Such algorithms enjoy discrete conservation laws that are crucial for stability and accurate simulations of turbulent steady states ($>10^6$ time steps).

We designed three mini-apps that use different existing frameworks (AMReX, MFEM, and StarPU) to explore different aspects of the work. StruGePiC is a mini-app based on the AMReX framework, which includes support for most HPC architectures and has potential for very good scalability. SymPiFE-VMax is a similar mini-app based on the MFEM framework that offers more flexibility and allows for more detailed control of the finite-element mesh, but requires one to write many of the algorithms by hand. Both of these mini-apps explore how to leverage an existing HPC framework for PIC plasma simulations and potentially offer a way to get good performance and highly portable codes aiming for the heterogeneous exascale systems. Additionally, we designed and optimised a simple PIC code called SIMPIC to explore task-based parallelism on GPUs with the StarPU framework. First we provide a fully GPU SIMPIC code and show that the run time is 50 % reduced compared to CPU runs. In future this code will be used as a test example for modifying the other more complex PIC codes in terms of CPU to GPU migration.

Based on the lessons learned, the ELMFIRE community is actively developing the work further with the aim of an accurate, explicit simulation code of plasma tokamaks that is able to fully utilise the upcoming heterogeneous pre-exascale and exascale systems.

Vlasiator: The Vlasiator team has worked to understand how to port various parts of the code to GPUs, participating to GPU Hackathons, profiling the code, and optimizing. The initial approach of porting to OpenACC did not perform as expected. Then we started investigating porting to CUDA which is under development, solving bugs and optimizing. As the code is in advanced C++, we have also identified some bugs with the NVIDIA compiler.

4.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

GENE/tasks: GENE solves the gyrokinetic integro-differential Vlasov-Maxwell system of equations on a 5D phase-space grid with an explicit Runge-Kutta method. It is used by many users to simulate microturbulence in fusion and space plasmas. While usually using a static MPI domain decomposition of the 5D phase space as parallelization paradigm, in this project the usage of a task-based approach for parallelization has been added and evaluated. The well-known StarPU framework has been used to introduce tasks in addition to the original domain decomposition.

The code developed during this project is the first implementation that explores the natural task-based parallelism of GENE. The implementation uses a task-based approach and it is intended to run with StarPU, a runtime system that efficiently maps computation tasks to hardware. This task-based version of GENE (GENE-SPU), is based on the latest CUDA branch of GENE and required a non-trivial extension and refactoring of the original class design. The reuse of the existing objects for the computation of the tasks as much as possible was mandatory as well as to change as little as possible of the original code to preserve readability and code structure. The former is very important since GENE has been highly optimized over the years and we intend to keep the original code structure as an alternative. The latter was also necessary since the amount of code required by StarPU increases considerably with the number of tasks, and the resulting code might bury the original code, making it hard to read, hard to develop new physics or modify it by other users and/or developers.

The current version of GENE-SPU can compute the time-step for linear problems with the local discretization approach using StarPU tasks only. For the profiling shown in Figure 8 and Figure 9, we used an external tracing application FxT for tracing purposes, with the resulting trace visualised using the ViTE visualization program.

Figure 8 shows a Gantt diagram of the tasks performed during the computation of the right-hand side vector (rhs) required in the time-step and how they were distributed on the available hardware.

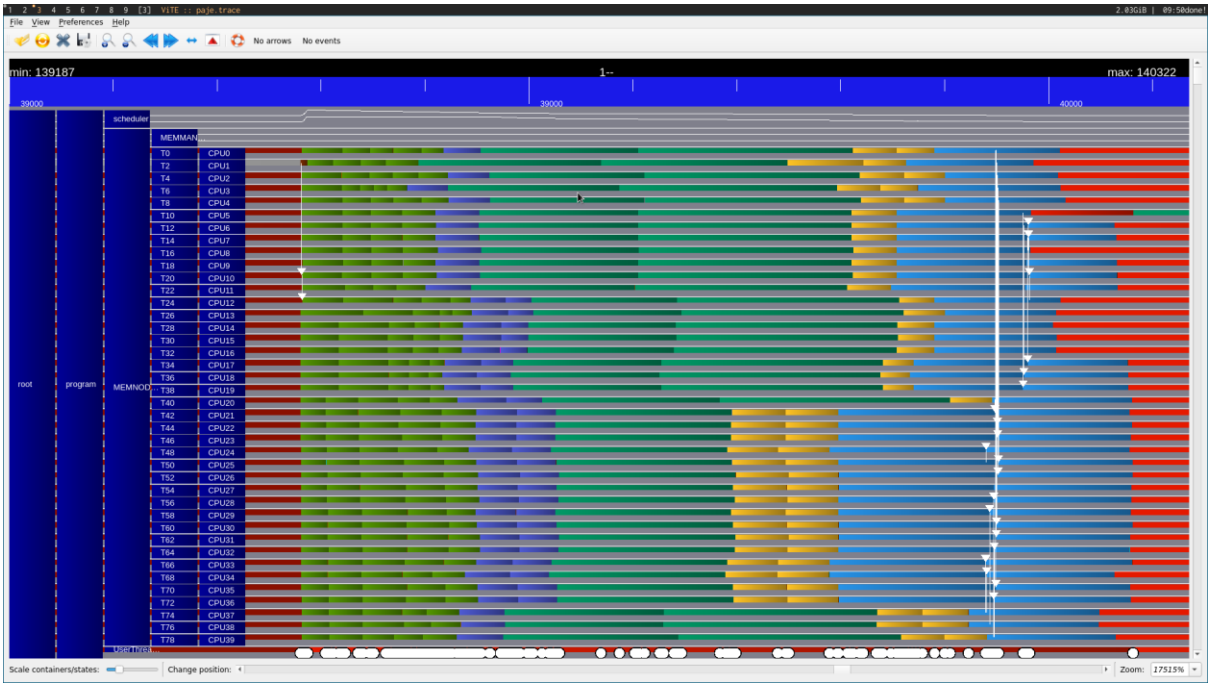


Figure 8: Gantt diagram of the tasks performed during the computation of the right-hand side vector (rhs) required in the time-step and how they were distributed on the available hardware.

In Figure 8, the computational hardware is shown on the left in blue (in this case only CPUs, an entire node), and the tasks are shown as coloured blocks in front of the hardware which executes them. The x-axis is time (in ms) and the length of the block is the duration of the task. Red bars mean that the hardware is idle. This occurs due to synchronization or a bottleneck inherent to the algorithm. At the bottom, white circles denote the different events performed by StarPU, such as

task insertion, synchronization, data partitioning, etc. and the arrows denote memory movements. Blocks with the same colour perform the same operation but on different data, for example, all yellow tasks compute the dfielddxy term. An important addition / feature in GENE-SPU is that not only the tasks required to compute one term (i.e. same colour) can be overlapped but also the computation of the terms. This means that tasks of different colours can be overlapped and be executed in different hardware as can be seen in Figure 9. Note that in contrast to the previous diagram, there is only one task for the computation of each term to fully use the GPU, otherwise tasks will be too small and more memory transfers from CPU to GPU will be required.

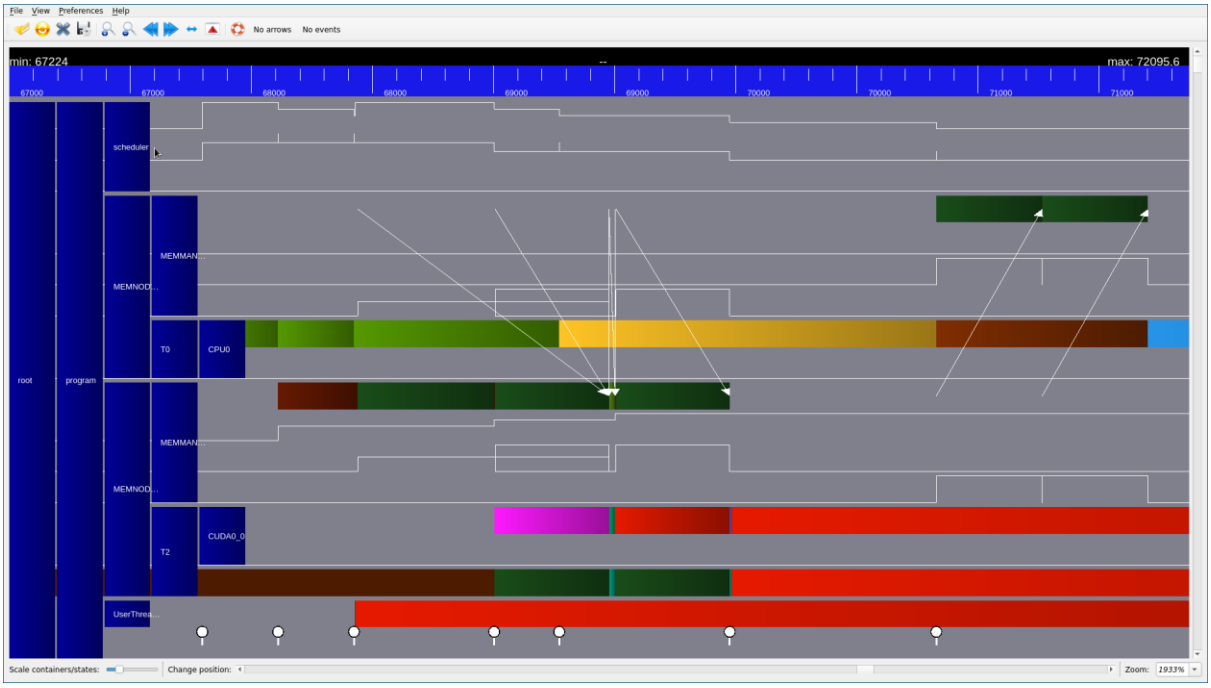


Figure 9: Gantt diagram of the computation of the rhs. The new task-based parallelism allows to overlap the computation of entire terms (which is not possible in the MPI implementation) and now they can be computed concurrently in both the CPU (upper part) and GPU (lower part).

The current GENE-SPU implementation works on a single node using 1 MPI-rank and all available cores and has support for GPUs. Nevertheless, to become exascale ready, running on many accelerated nodes with an underlying MPI parallelization is necessary. StarPU in principle supports this, but it has not yet been included in GENE-SPU. It is then also necessary to schedule the tasks on all available CPUs and GPUs.

Vlasiator: Vlasiator models the near-Earth space plasma by propagating the six-dimensional (3D position, 3D velocity) particle velocity distribution function for ions using the Vlasov equation, under the effect of the Lorentz force caused by the electromagnetic fields [22]. The equation is therefore coupled to the Maxwell equations in the Darwin approximation. The system is closed through the generalised Ohm's law including the Hall and electron pressure gradient terms and hence electrons are approximated as a charge-neutralising fluid. The code [23] is open source and parallelisation is done leveraging MPI, OpenMP, vectorisation as well as hyperthreading when available. Porting effort to GPUs is on-going with initial testing done using OpenACC offloading and current work focusing on restructuring the code as needed and using CUDA directly.

The test case chosen corresponds to a 6D magnetospheric run with four levels of adaptive spatial mesh refinement identical to a production run performed on Hawk at HLRS in 2021, run from t = 0 for 20 steps. To make this test more realistic the temperature of the plasma was made inhomogeneous spatially by a factor of 4, increasing the imbalance in computational load as compared to a uniform case. This is more characteristic of the pronounced spatial disparity in computational load experienced in production conditions.

Mahti is the largest Finnish national supercomputer with a theoretical peak performance of 9.5 petaflops. It is a Bull Sequena XH2000 system with an architecture similar to the Vega EuroHPC petascale system. Mahti has a CPU partition of 1404 nodes with two 64-core AMD Rome 7H12 CPUs and 256 GB of memory on each compute node, and 200 Gbps HDR link for the network interconnection, and a GPU partition of 24 nodes with four NVIDIA A100 GPUs. Benchmarking results shown below were obtained on the CPU partition of the Mahti supercomputer. Similar scalability was also seen on the Hawk supercomputer at HLRS.

On Mahti, the test case for Vlasiator scales well up to 160 compute nodes with a speedup of 12 times compared to 20 nodes as can be seen in Table 4 and Figure 10.

Nodes	Cores	Total time (s)	Propagation time (s)
20	2560	2767	1855
40	5120	1693	1086
80	10240	646	388
160	20480	172	147
200	25600	213	111

Table 4: Results for Vlasiator scalability on Mahti. The propagation time does not include initialization, IO, and load balancing.

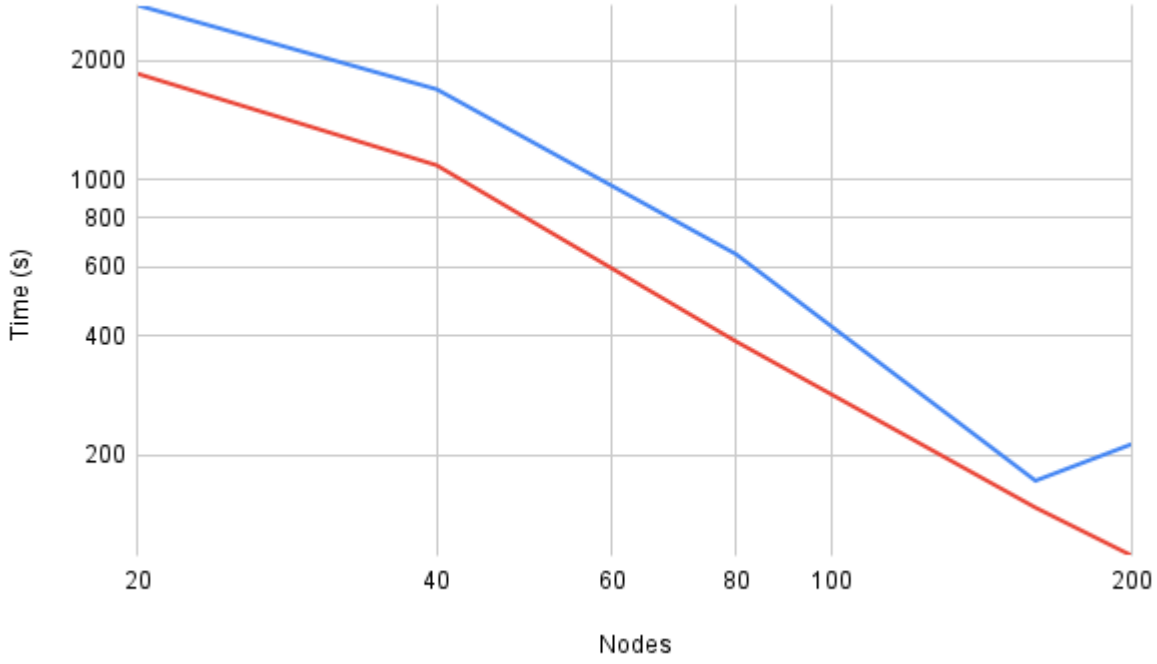


Figure 10: Scalability study of Vlasiator on the Mahti supercomputer with total time (blue) and propagation time (red) in seconds.

The figures reported here include both the total time with the not very well threaded initialisation, one IO call, two load balance calls, as well as the pure propagation time that excludes initialisation, IO and load balance. IO was performed using optimised MPI IO calls using the VLSV library (github.com/fmihpc/vlsv). Runtime snapshots have a typical size of 20 GB each and are output at a cadence of a few per wall time hour. Restarting checkpoint files are typically stored every 12 or 24 hours and can be up to several TB. The standard procedure is to keep the minimum number of restart files on disk to secure the continuation of the run in case of a file corruption or similar issue arises.

StruGePiC / SymPiFE-VMax: StruGePiC (Structure-preserving Geometric PiC) and SymPiFE-VMax (Symplectic Particle-in-Finite-Element Vlasov-Maxwell) are both codes developed to simulate the same system, the Vlasov-Maxwell equations for full-orbit (6D) charged particles. The numerical methods employed, explicit symplectic time integration of Lagrangian markers and discretisation of fields on de-Rham preserving finite differences/finite-elements are essentially identical. The two software are distinguished by the use of different frameworks as building block: the adaptive mesh refinement framework AMReX in the case of StruGePiC, and high-performance finite element framework MFEM in the case of SymPiFE-VMax. Both frameworks are written in C++, as are the softwares. The frameworks offer extensive parallel infrastructure for MPI, as well as for threading on the CPU or the GPU, enabled by architecture-agnostic macros. This permits very flexible porting of the software, once the conversion to a threaded version has been performed. AMReX and MFEM support a variety of backends, which include CUDA and HIP. AMReX readily supports PiC methods, which made the implementation of our particular scheme of choice straightforward. Conversely, MFEM is designed for pure finite-element applications, which poses

some challenges to implement efficiently certain methods of MPI-parallelisation common in large scale PiC simulations (i.e. halo regions and domain cloning).

The test cases used for the scaling tests are a non-linear wave conversion case, and a simple plasma oscillations case. The first consists of a slab simulation domain separated between a region occupied by plasma, and a region of vacuum. A constant magnetic background is used, and an electromagnetic wave (X-mode) is traveling in the vacuum region towards the plasma, which nonlinearly excites oscillations with different frequency and polarisation (Bernstein wave). The second test case consists of a slab simulation domain filled with plasma of uniform density and temperature. In this case the charges move freely, and generate electromagnetic Langmuir waves at the so-called plasma frequency.

Porting of StruGePiC was made on national machines Puhti (architecture similar to SuperMUC-NG), Puhti-AI (similar to JUWELS Cluster) and Mahti (similar to Vega and MeluXina). Mahti is also equipped with NVIDIA A100 GPUs, however scaling tests were only conducted on Puhti and Puhti AI. As can be seen in Figure 11, StruGePiC showed excellent scaling on one GPU node, and good scaling up to four GPU NVIDIA V100 nodes. Further optimisation work is ongoing to improve the scaling on more nodes in realistic cases. Porting to AMD GPU machines has not been performed, however StruGePiC relies on AMReX's architecture-agnostic macros for handling GPU threads, hence it can readily be ported to the architectures that AMReX supports i.e. for GPUs, NVIDIA (CUDA), AMD (HIP) and Intel (DPC++).

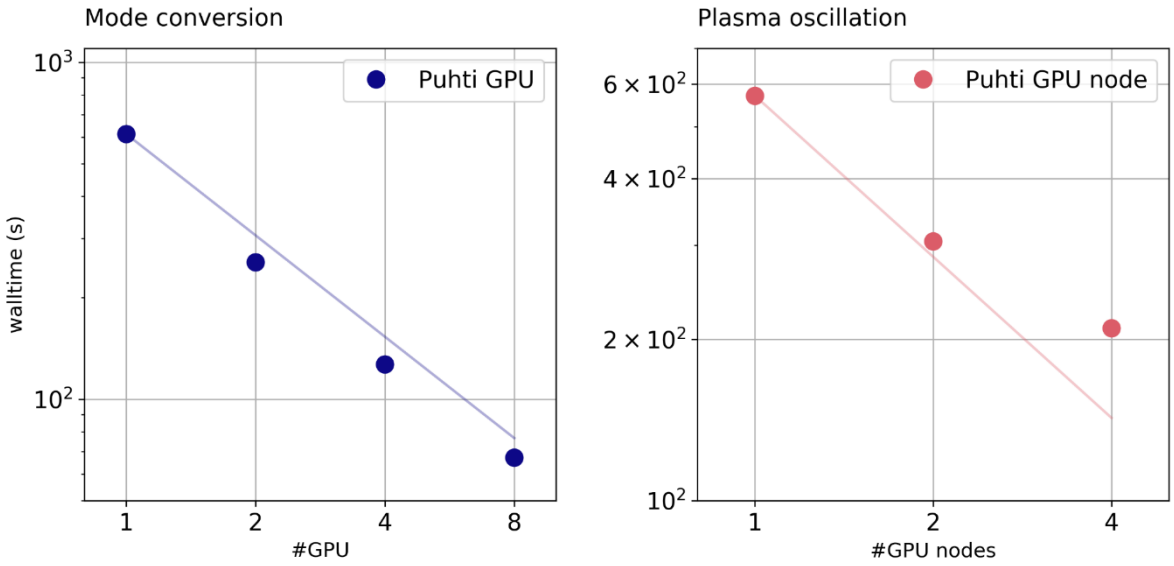


Figure 11: Scaling of StruGePiC on Puhti

SIMPIC: SIMPIC (Simple PIC) is a simplified Particle in Cell code. The SIMPIC code was developed under certain hypotheses which make the simulation significantly easier. There is assumed to be no collisions between particles, no magnetic field and only free electron particles (no ions). As per these assumptions, the complicated Maxwell's equations boil down to solving only a Poisson equation for the potential. This is easily done using the well-known finite difference method. Now, the field can be calculated simply by taking the gradient of the potential. In a PIC code, the whole plasma region is divided into sub regions called cells. Inside each of these cells, there are some particles (ions/electrons). We give an initial random distribution of particles inside

the plasma device. Then, we apply an external electromagnetic field to these particles usually in the form of a voltage source. After this initialization, the PIC code follow a common algorithm as seen in Figure 12 for SIMPIC.

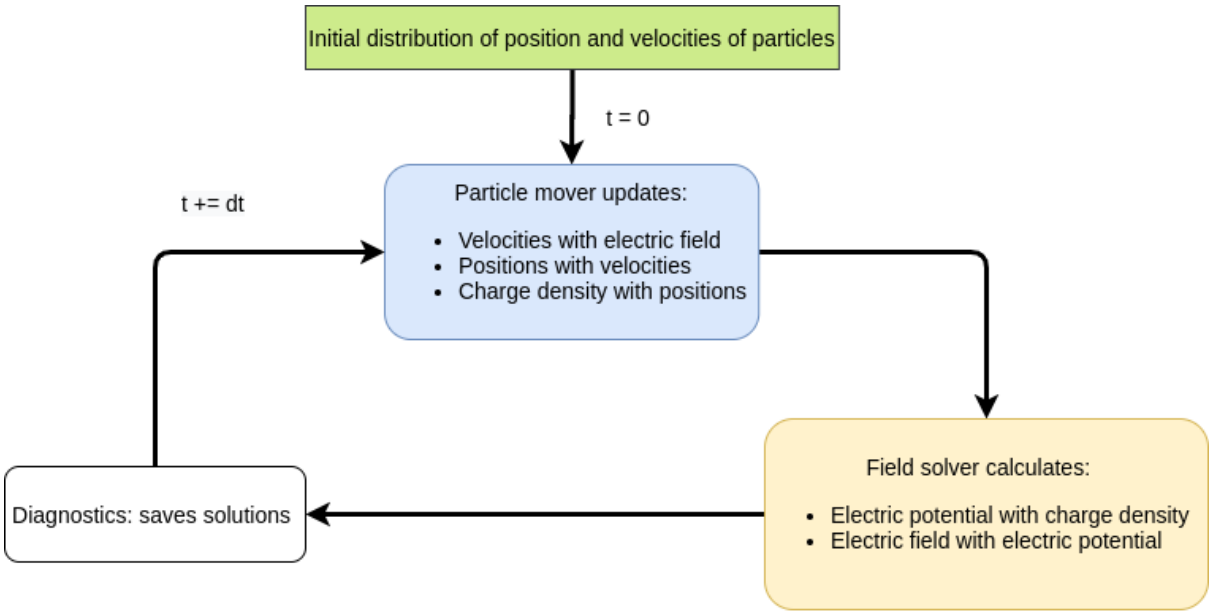


Figure 12: SIMPIC workflow diagram. It shows the general algorithm flow of the PIC codes which consists of two algorithms: Particle mover and field solver.

The code was run on the VIZ supercomputer at the Faculty of Mechanical Engineering in Ljubljana, which has two 12-core CPUs (Intel Xeon E5-2680V3, 2.5 GHz), 256 GB of DDR4 memory running at 2133 MHz with ECC, three NVIDIA Tesla K80 GPUs and 250 GB of memory per node, and on Marconi100 at CINECA, which has two 16-core CPUs (IBM POWER9 AC922, 3.1 GHz), four NVIDIA Volta V100 GPUs with Nvlink 2.0 and 16 GB RAM memory of the GPU, and 256 GB of memory per node. We used OpenMPI 3.1.4 and CUDA 10.1 with the GCC compiler on VIZ and OpenMPI 4.0.3 and CUDA 10.1 with the GCC compiler on Marconi100.

The GPU optimization of the fully particle mover was done in the following steps:

1. Optimisation of the Particle mover

   Before we can move the particles, we need to know what forces are acting on each particle. This force is derived from the surrounding electric field. However, this is not so trivial. It is important to understand that cells in the plasma region form a grid and the potential and electric fields are calculated only at these “grid points”. A typical particle mover algorithm would be something like:

   a. Gather field at particle position;

   b. Calculate new velocity using field;

   c. Calculate new position using new velocity.

   One should also note that the GPU has its own memory space. Hence, we decided to create the particles in the GPU alone to avoid this memory transfer. We also implemented an optimised algorithm for particles that go beyond the plasma region using a Boolean array to flag particles alive/dead. This aids in vectorised processing on GPUs.

2. Optimization of the Field solver

   To calculate the electric potential, the code has to compute the solution of a tridiagonal matrix system which comes from the Poisson's equation, and this is a very sequential calculation. There are many algorithms designed to do this calculation, but CUDA comes with a library called cuSPARSE for algebraic calculations on GPU. This library contains a function which calculates the solution for this tridiagonal system. The parallelization process for the rest of computations of the field solver part is similar to the particle mover, but in this case, each thread is assigned to a grid point.

   In summary, the parallelization of the field solver follows the next steps:

   a. Solves the tridiagonal matrix using an external library;

   b. Corrects electric potential with the boundary values;

   c. Calculates the electric field with the electric potential.

3. Create tasks

   Now, we can make even better use of the computing resources if we can use both the CPU and GPU for computation. This is done by creating tasks or 'codelets' using StarPU, which is a software tool which can schedule tasks to run on heterogeneous architectures (CPU+GPU). All memory transfers and allocations in the application is done by StarPU itself, which saves some amount of code. We have to note that the two last steps are independent and, consequently, efficient for GPU calculations. Using the same case, we produced benchmarks on VIZ HPC and Marconi100 with CPU, GPU and StarPU version of the code.

Our benchmarks of the SIMPIC versions show that the GPU versions have much better performance than the CPU version. The GPU particle mover shows a speedup of greater than 5x, which is to be expected as the particle calculations are well parallelised and the CPU-GPU memory transfers have been optimised. However, this speedup seems to saturate as we increase the number of particles above 10^5 particles. Hence, this calculation limits the performance of our code. With regard to this, it can also be observed that the number of particles per cell (PPC) affects the speedup. This could mean that the performance would be better if we have more cells for a given number of particles. On the other hand, the CPU-GPU memory transfer of the bigger density arrays associated with a larger number of grid points also requires more time. Hence, we observed that there is an optimal number of particles per cell which would give us the best speedup for the particle mover. On the other hand, for the field solver, we see that the GPU version is slower than the CPU one for low numbers of grid points, and it is faster for a high number of grid points. This time consumption mainly comes from the tridiagonal solver which is not efficient in GPU for low numbers of grid points but its calculation time remains more or less constant with the number of grid points. It should be noted that the most time expensive part of this code can be the generation of the diagnostic logs that should be turned off for performance. As can be seen on the plot on the right in Figure 13, there is a constant difference between the GPU fields + particles (red curve) and GPU fields + particles w/o diags (dotted curve), which indicates the amount of time consumed by the diagnostic operations.

Our StarPU version of SIMPIC shows good speedup when compared to the CPU version. However, this is not as much as our CUDA-only GPU version. The particle mover runtimes for the StarPU

version are slightly faster than the GPU version. This is because the StarPU data management is more efficient at data transfers.

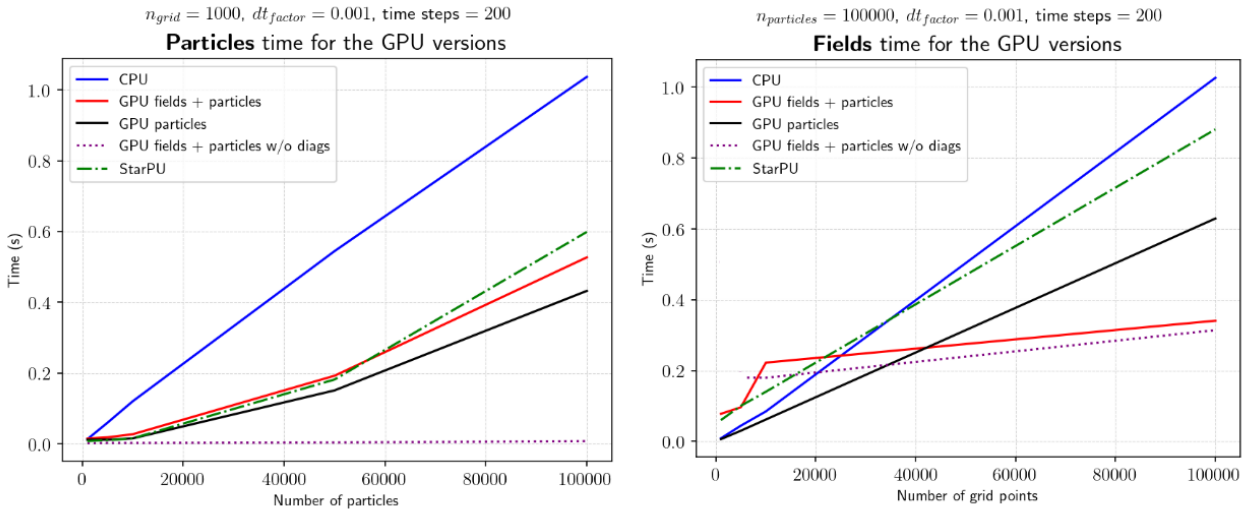


Figure 13: Comparing performance of various accelerated versions of SIMPIC on the VIZ cluster. Left: Runtime Plot of the Particle Mover against number of particles. Right: Runtime plot of field solver against number of cells.

For the SIMPIC GPU version profiling we used the NVIDIA Visual Profiler to visualize the profiling done on the full CUDA version of SIMPIC. The NVIDIA visual profiler comes along with the CUDA toolkit and hence does not need to be installed separately. In Figure 14 the timeline for the fully GPU version of SIMPIC is outlined.

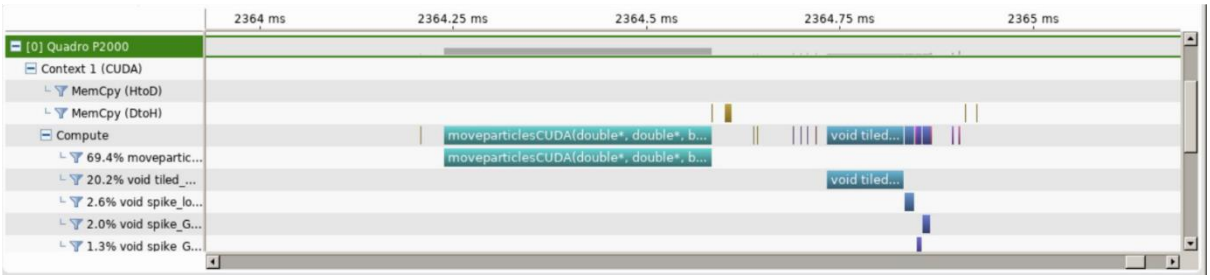


Figure 14: Timeline of One Time Step of the full SIMPIC GPU version.

From Figure 14 one can see very minimal data transfers between the host and the device when using our optimized data transfer algorithm. The particle mover accounts for close to 70% of the compute time whereas the remaining compute time is taken by the cuSparse matrix solver for fields. This shows that we have been able to incorporate the two main computationally intensive parts of a Particle-in-Cell code in GPU.

For the profiling of the StarPU version we again made use of the external tracing application FxT for tracing purposes, with the resulting trace visualised using the ViTE visualization program. The profiling timeline for the application is presented in Figure 15.

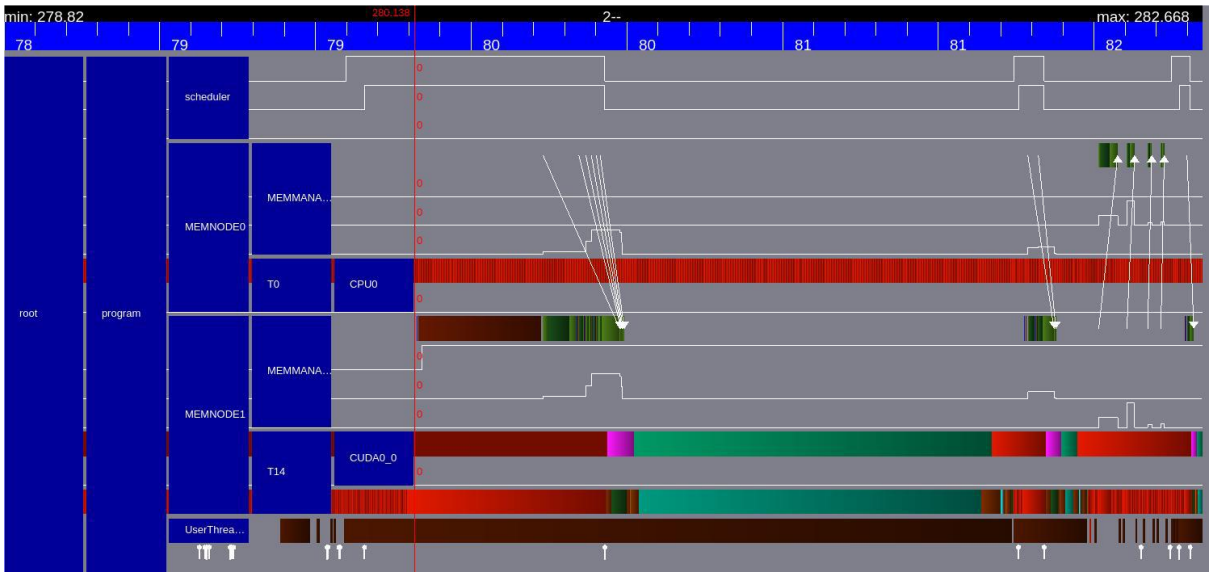


Figure 15: Timeline for StarPU particle mover task.

From Figure 15 we can see explicitly the memory transfers from the CPU memory to the GPU memory as white arrows. The green bars indicate the StarPU tasks. The first task is the GPU particle mover task and the smaller green bar later is the field solver GPU task. The red parts indicate that the PU is idle. Note that there are no explicit memory transfer calls in our code. All the required memory transfers are taken care of by the StarPU memory management and scheduler automatically. The timeline looks very similar to the previous GPU version, which is to be expected as it is essentially the same code but with the kernels executed as StarPU tasks.

4.3 Interactions with stakeholders, users, outreach and publications

GENE/task: GENE is a well-established plasma micro turbulence code that is widely used by the plasma physics community. Since we are working directly with the GENE code, its modernization and any performance improvement achieved by the task-based parallelisation with StarPU and use of heterogeneous hardware will directly be a benefit for all developers and users of the code. All code modifications have been in discussion with GENE developers and are therefore easily adapted and integrated in the production version.

VLASIATOR: A presentation is scheduled at the upcoming PRACE Autumn School 2021: Harnessing the EuroHPC Flagship Supercomputers with the title ‘Program acceleration with GPU using CUDA’ - Dr. Talgat Manglayev (October 13th 2021).

SYMPIFE-VMAX/STRUGEPIC: Peer-reviewed articles of the numerical methods developed with the mini-apps SymPiFE-VMax and StruGePiC are in preparation. Both SymPiFE-VMax and StruGePiC are being used by the ELMFIRE team at Aalto University. Further developments and use for physics research are ongoing within the EUROfusion Theory, Simulation, Validation and Verification program (TSVV-4). This activity, in direct collaboration with the Max-Planck Institute for Plasma Physics (Garching, Germany) aims at demonstrating the use of these simulations in regions near the tokamak edge, where the validity of gyrokinetic theory is in question (motivating simulations with full-orbit particles), installing them among reference tools for the European community.

The mini-apps are also central to two projects pending review from the Academy of Finland, in collaboration with leading experts of plasma-gas PIC simulations at the Institute of Plasma Physics (Prague, Czech Republic), in order to access strongly multiphase regimes (so-called detached plasma).

SIMPIC: We produced two conference publications on MIPRO 2020 [17] and 2021 [18]. The first publication describes how the GPU optimization was done in the prototype PIC code. The second publication describes the fully GPU field solver of OOPD1. We also participated in ASHPS (First Austrian-Slovenian HPC meeting) [19] and Autumn PRACE School in 2020 [20]. There we presented the GPU method for optimization on PIC codes.

4.4 Overall assessment of achievements and future developments

Overall, the project has been a success with many of the main objectives achieved, but due to the complexity of the problems tackled not everything could be finished within the project timeline. Further work is planned and it is expected that all of the original objectives will be met or exceeded in the near future.

Mini-apps

In-line with the project goals, three plasma physics mini-apps were developed and/or ported to GPUs during the project. These mini-apps explored various approaches and ease the adoption of the lessons learned by the community.

SIMPIC: The methods for PIC codes optimization in GPU and explored task-based parallelism using StarPU were developed. These methods have been implemented also in more complex PIC codes such as BIT1 and OOPD1. Future work is planned to produce a workable SIMPIC on multiple GPUs and carry over the GPU implementation to even more complex PIC codes.

SymPiFE-VMax: A new mini-app for particle-in-finite-elements Vlasov-Maxwell systems with multiple species was developed. It demonstrates the use of the MFEM framework for creating PIC plasma simulations on both CPUs and GPUs. Together with the StruGePiC mini-app, it serves as a basis for the refactoring of the ELMFIRE code (see below).

StruGePiC: A new mini-app was developed for structure preserving PIC simulations using AMReX to implement an explicit structure-preserving algorithm [24]. It demonstrates the use of the scalable framework AMReX for creating PIC plasma simulations on both CPUs and GPUs. Together with the SymPiFE-VMax mini-app, it serves as a basis for the refactoring of the ELMFIRE code (see below).

GENE/tasks

The main objective was to implement and to explore task-based parallelism in GENE. We clearly underestimated the amount of work that it requires for such an implementation. One difficulty comes from the fact that GENE is written in modern Fortran, but the Fortran layer of StarPU is not as mature as the C layer. Hence, several contacts to the StarPU developers with feature requests and bug fixes were necessary to make the full functionality of StarPU available from Fortran.

The resulting code can now compute a time step in the Runge-Kutta scheme using a local approach with StarPU for a single test scenario. However, since the implementation required a modification and refactoring of interfaces present in the code path of the test problem, the implementation for other paths that use the same interfaces require minimal changes to complete its taskification. This

is true even for the global approach. The design of a class hierarchy that fits into the existing object structure of GENE needed several attempts and a lot of implementation work.

Due to the time constraints, not enough research was done regarding the optimal task size to be used in the critical GENE components, the different schedulers, the performance models and the portability of the code. We plan to work on single node optimization now that the tasks are available for CPU and GPU and only the necessary synchronization points have been kept. These tests will improve our understanding of the effects of each component in the task-based parallelization. Finally, we have plans to support MPI in GENE-SPU. Either GENE is launched with just one MPI rank per node and StarPU pauses during MPI communication and resumes to use MPI as usual, or we use the MPI library provided by StarPU.

Vlasiator

The main objectives for Vlasiator were to look into porting some of the main algorithms to GPUs and to explore the possibilities of task-based approaches. Initial GPU porting efforts were put into investigating how well OpenACC could be utilised to offload main computational routines to GPUs. As part of this, the team participated in a NVIDIA mentored GPU Hackathon and worked on the code with the guidance of an expert. Even though initial impressions were positive, achieving good performance was difficult e.g. due to overheads from data movements that could not be mitigated without major restructuring of the code and its data structures. In the end, the decision was to proceed with the GPU porting by restructuring the code as needed, but to use CUDA instead of OpenACC to gain more direct control of the detailed implementation. Moreover, the CUDA code can then be readily ported to HIP and thus one will be able to support all the EuroHPC systems. The new 3D capability of the Vlasiator simulation is based on using a suboptimal numerical grid for computations, as the simplest first step was to implement an adaptive mesh refinement (AMR), which is static in time. The work is on-going.

ELMFIRE

Two mini-apps (SymPiFE-VMax and StruGePiC) have been developed which demonstrate the use of structure-preserving numerical methods for plasma turbulence simulations aimed at accelerated architectures. Relying on existing frameworks has made it possible to produce such parallel softwares in a relatively short time, although the features absent from MFEM have induced significant challenges in the development of SymPiFE-Vmax. We expect however that this will be compensated by the very high versatility of MFEM, which will facilitate simulations in realistic geometry using curvilinear meshes. In addition, one mini-app (SIMPIC) was used to explore task-based parallelism using the StarPU framework.

Further developments are planned both to improve performance and scaling, and to add features expanding the domain of applicability of the simulation (such as collisions or relevant boundary conditions).

5 LoSync: Synchronisation reducing programming techniques and runtime support

5.1 Introduction and summary

The LoSync project aims to improve the scalability of applications by removing unnecessary synchronisation and serialisation, and by fully exploiting the potential for overlapping computations and communications. To do this, we make use of modern features of well-standardised APIs, to ensure portability and relevance.

Efficiently implementing a pure task-based programming on distributed memory systems is very challenging. Instead, we propose a hybrid model which uses task-based programming inside a node and traditional message-passing between nodes. To minimise synchronisation and expose as much parallelism as possible, our experience has shown that communications as well as computations should be expressed as tasks. However, standard communication libraries are difficult to use like this without encountering the risk of deadlocks, for example where all threads are executing tasks containing blocking communication calls. In LoSync we are developing and evaluating task-aware versions of MPI and GASPI libraries (called TAMPI and TAGASPI) which are integrated with OmpSs-2 and OpenMP task-based runtimes. In TAMPI and TAGASPI, tasks blocked on communication calls are paused, freeing their executing threads to process other tasks, until the communications complete and the paused tasks can be resumed.

Figure 16 illustrates the software stack for our solution. The application sits on top of the TAMPI and TAGASPI libraries (normally an application would choose one or the other, but to call both libraries from the user level is possible). The user interface for these is mostly the same as for the underlying MPI and GASPI libraries, with a small number of extensions. The TAMPI and TAGASPI libraries in turn interface with the underlying communication libraries and the tasking runtime system (OmpSs-2 or OpenMP).

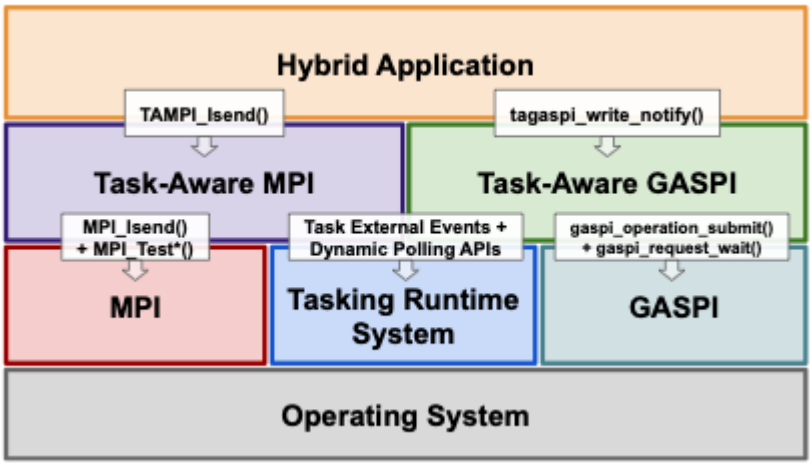


Figure 16: Software architecture for hybrid applications using TAMPI and TAGASPI libraries

Figure 17 shows the state transition diagram for tasks in this model. Whenever a running task is blocked in MPI (for example), its status is changed to paused, and the executing thread is also paused, but the CPU that was running that thread is able to execute other tasks (either computation

or communication). When the blocking MPI call completes, the task becomes ready again, and can be rescheduled for execution when resources are available. TAMPI also supports another mode which allows the task to complete but only release its dependencies when the MPI call completes: this mode is easier to integrate with existing OpenMP implementations, but requires extensions to the MPI interface. A similar model is used in the TAGASPI implementation.

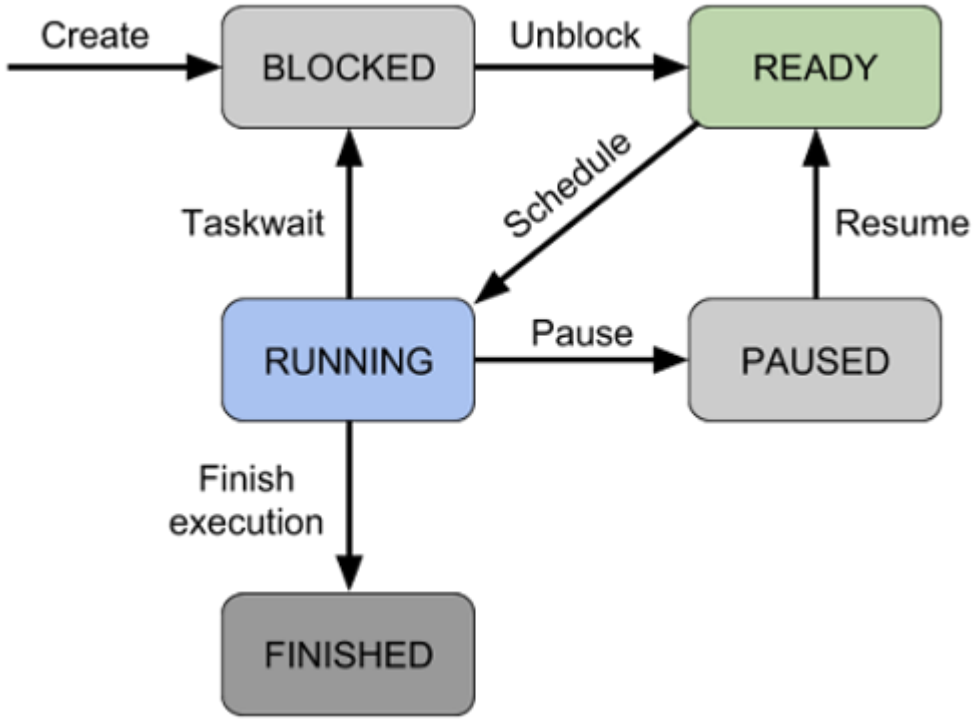


Figure 17: State transition diagram for tasks (blocking mode)

5.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

We have evaluated the performance and programmability of the Task-Aware MPI (TAMPI) and Task-Aware GASPI (TAGASPI) libraries on a number of mini-applications. In this section, we focus on just two of these - Gauss-Seidel and miniAMR. Further results can be found in the publications cited in Section 4.3. For both codes, we evaluate (1) an optimized two-sided MPI-only approach, (2) a two-sided hybrid MPI+OmpSs-2 variant that leverages TAMPI, and (3) a one-sided hybrid GASPI+OmpSs-2 variant that leverages the TAGASPI library.

We run our experiments in the Marenostrum4 supercomputer with up to 256 nodes (12288 cores). Each node has two sockets Intel Xeon Platinum 8160 (2.10GHz) with 24 cores each (48 total cores), 96 GB of memory, and an Intel Omni-Path HFI Silicon 100 Series network. We also use 16 nodes (1024 cores) of the CTE-AMD cluster. Each node has a single AMD EPYC 7742 (2.250GHz) with 64 cores (SMT is disabled), 1TB of memory, and a Mellanox InfiniBand HDR100 network. We use the Intel 2017.4 compilers and Intel MPI 2017.4 on Marenostrum4, while Intel 2018.4 and OpenMPI 4.0.5 on CTE-AMD.

A. Gauss-Seidel

We first use the iterative Gauss–Seidel method that solves the Heat equation, a parabolic partial differential equation describing the heat distribution in a region over time. This benchmark uses a 2–D matrix logically divided into blocks. The matrix is distributed across ranks assigning a consecutive set of rows to each one, so processes exchange the boundary rows with the upper and lower neighbors only.

We evaluate an optimized MPI-only version that uses non-blocking MPI primitives. We also evaluate two hybrid MPI+OmpSs-2 variants that taskify both computations and communications and leverage the TAMPI non-blocking support (TAMPI_Iwait) and TAGASPI one-sided support respectively.

We evaluate this benchmark in Marenostrum4; the MPI-only spawns 48 ranks/node, and the hybrid variants use one rank/socket (24 cores/rank) to avoid NUMA effects. Figure 18 shows the strong scaling experiment using the optimal block size of each variant. The upper figure shows the speedup, and the lower presents the parallel efficiency.

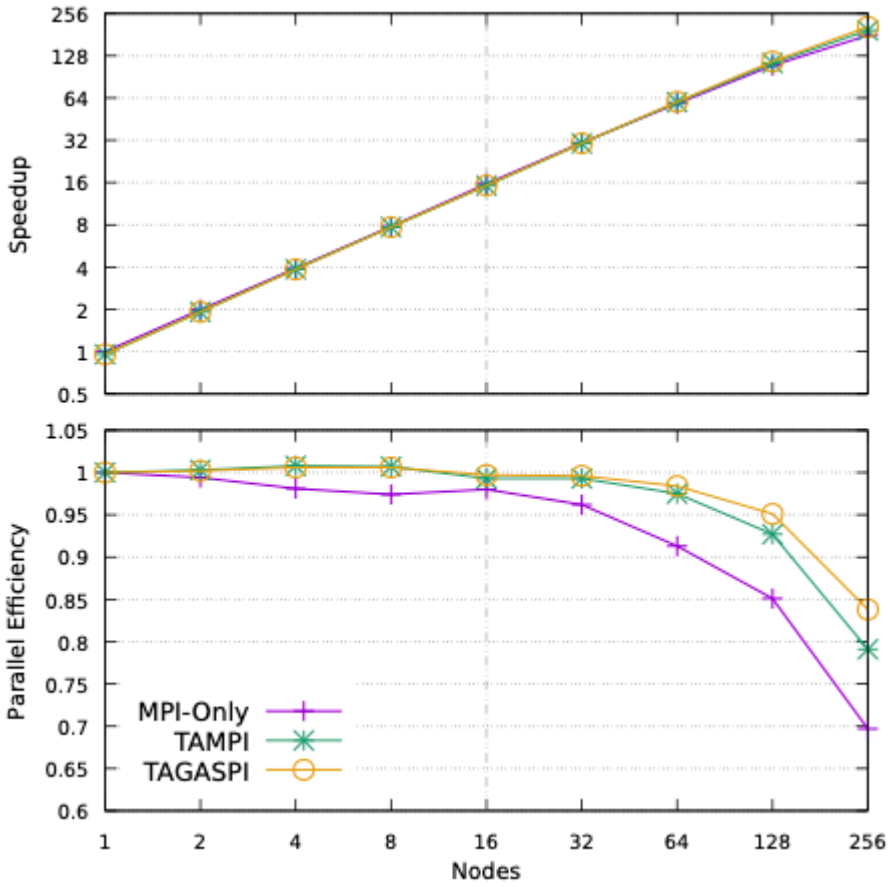


Figure 18: Gauss-Seidel strong scaling with a 256Kx128K matrix and 1000 timesteps in MN4 from 1 to 256 nodes. Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input (64Kx32K matrix and 1000 timesteps) for the experiments from 1 to 8 nodes.

The MPI-only version is only competitive on a single node, its performance starts to degrade with two nodes and it ends performing worse at 128 and 256 nodes. The TAMPI version improves that

performance in this latter scenario, but TAGASPI is the version that scales best. At 256 nodes, TAGASPI outperforms MPI-only and TAMPI by 1.15x and 1.06x, respectively.

We run another experiment to observe how the variants behave when putting more pressure on communication. Figure 19 shows their throughput at 128 nodes, halving the previous input size and modifying the block size. Notice that changing the block size implies changing the computation and communication granularities, as well as the task granularity for the two hybrid versions. TAGASPI outperforms the rest in all cases, especially for configurations with a small block size, where the communication cost has a larger impact. The lower performance of TAMPI for small block sizes can be explained by threading contention inside the MPI library. TAGASPI using small blocks of 128x128 still works at an acceptable 60% of the peak throughput, while MPI-only is at 41% and TAMPI at 30%.

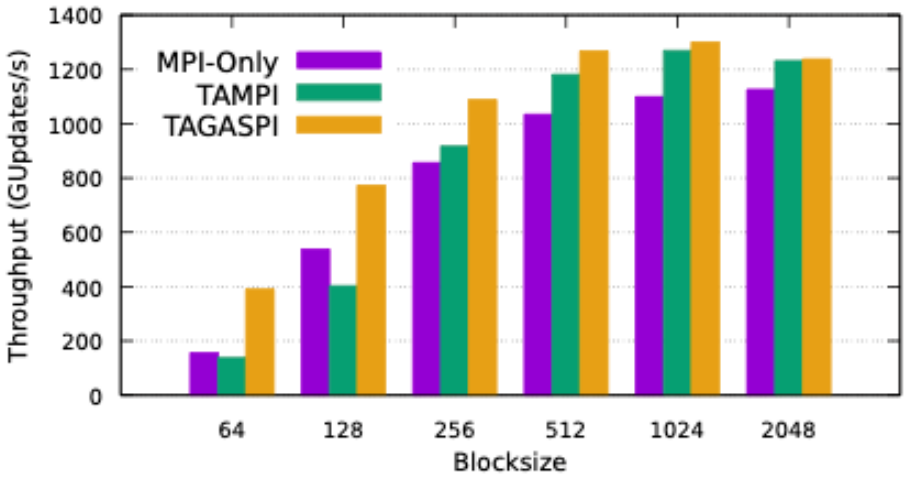


Figure 19: Gauss-Seidel throughput varying the block size with a 128Kx128K matrix and 500 timesteps in Marenostrum4 with 128 nodes.

B. miniAMR

The second application is the miniAMR, which mimics the communication, refinement, and load-balancing of larger adaptive mesh refinement applications. MiniAMR simulates the physics conditions of a 3–D domain when objects move across it. These objects create turbulent conditions in the regions they are present in and miniAMR increases the simulation accuracy in those parts. The domain is initially divided into 3–D blocks and distributed among processes, but due to the dynamism of the simulation, turbulent blocks are refined into smaller blocks and redistributed periodically. MiniAMR features multiple phases of computation and communication interleaved, and then a refinement and load-balancing phase periodically.

We run the following experiments in Marenostrum4; the MPI-only uses 48 ranks/node, and hybrids use 4 ranks/node and 12 cores/rank. That is the optimal configuration for hybrid approaches in miniAMR, given that the refinement phase is not fully taskified. In the TAGASPI variant, we also use TAMPI during the load-balancing stage to demonstrate that both libraries can work together. The load-balancing stage represents a small portion of the total time and does not present improvement opportunities, so this stage is still implemented with tasks that call two-sided TAMPI services.

Firstly, we perform a strong scaling experiment using 20 variables at each mesh point. The number of variables per grid point can be varied via an input parameter – larger numbers of variables increase the computation to communication ratio. The hybrid variants send/receive/write each boundary block face from a different task (separate messages). This is not the optimal configuration but provides competitive performance and puts more pressure on the communication phases. We show the speedup of a strong scaling experiment on the upper part of Figure 20, and the parallel efficiency on the lower part. We compute the speedup with respect to the throughput of the MPI-only variant in one node. We calculate the efficiency with respect to each variant's throughput in one node. Again, since the input is very large, we use a 16x smaller input from 1 to 8 nodes.

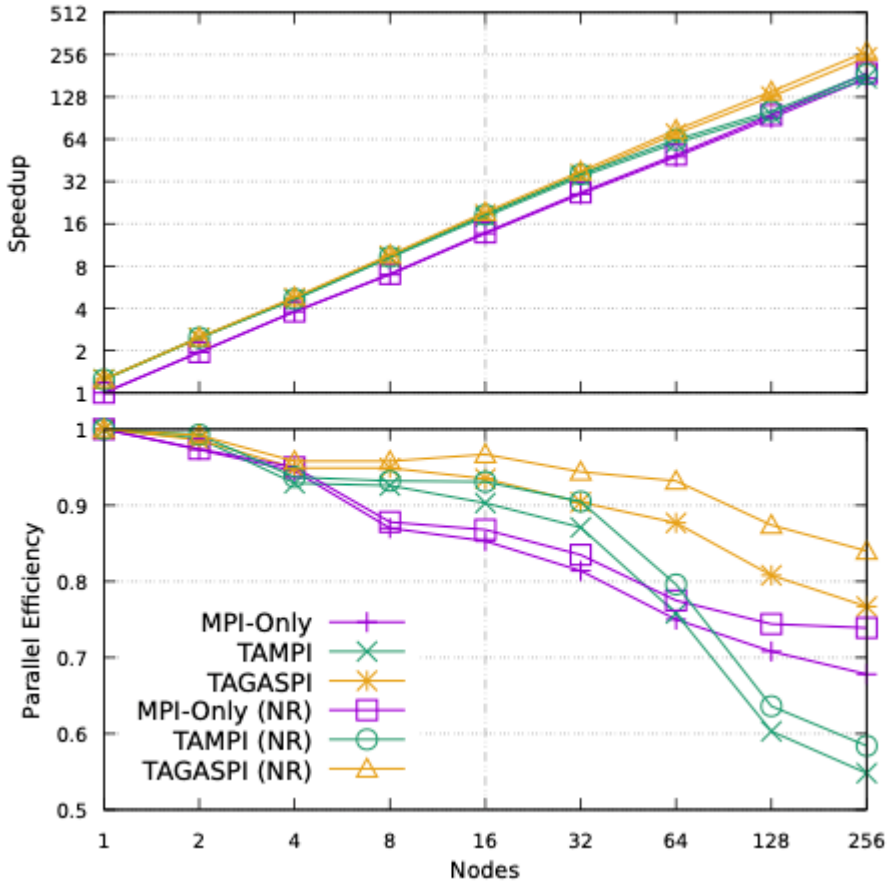


Figure 20: miniAMR strong scaling in Marenostrum4 from 1 to 256 nodes. The lower shows the efficiency for both the total time and assuming a negligible refinement time (NR). Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input for the experiments from 1 to 8 nodes.

In this case, TAGASPI achieves the best scalability and efficiency; it improves both MPI-only and TAMPI by 1.41x at 256 nodes. The efficiency of TAGASPI is significantly better since it ends with an efficiency of 0.84, while MPI-only is at 0.73 and TAMPI at 0.58 (non-refinement). Notice that TAMPI scales well up to 32 nodes, but it starts decreasing the efficiency from 64 nodes due to the high pressure on the communication. In those cases, TAMPI would need to increase the communication granularity to mitigate that effect.

We perform another experiment with 128 nodes using the previous input but varying the computed variables from 10 to 40 to see the impact on each variant. Figure 21 shows the throughput of each variant in this experiment. Again, TAGASPI performs better in all configurations with significant

differences. MPI-only is barely affected by the number of variables but has lower performance. The hybrid versions computing 10 variables show low throughput because the small granularity of computation tasks brings up the tasking runtime's overheads. TAMPI improves as we increase the computed variables and reduce the pressure on the communication side. The largest differences are with 20 variables, where TAGASPI outperforms MPI-only and TAMPI by 1.46x and 1.40x (non-refinement), respectively.

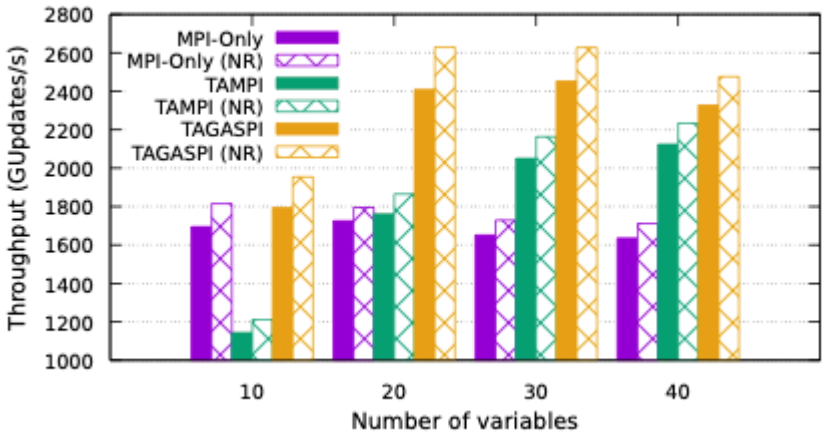


Figure 21: miniAMR throughput varying the number of computed variables in Marenostrum4 with 128 nodes. The figure shows the throughput for both the total time and assuming a negligible refinement time (NR). Notice that the vertical axis (throughput) starts at 1000 GUpdates/s.

During an analysis of the execution traces of the hybrid variants with 20 variables, we observed that the tasks using TAGASPI communications are much faster than with TAMPI. For instance, the sender and receiver tasks using TAGASPI are around 5x and 100x faster than the TAMPI ones, respectively. This difference is mainly explained by the high contention inside the MPI library when calling MPI_Isend and MPI_Irecv concurrently from several tasks. In contrast, the GASPI model allows GPI-2 to implement communications with lower threading synchronization.

5.3 Interactions with stakeholders, users, outreach and publications

Outreach and stakeholder engagement activities were unfortunately much reduced from what we had anticipated due to the pandemic, especially with regard to hands-on training or hackathon-style workshops. This project has produced three main publications. The first publication [25], describes how the TAMPI library can be leveraged to improve the performance of the miniAMR mini-app that has a complex and dynamic communication pattern. The second publication [26] describes the implementation of the Lulesh and HPCCG mini-apps using TAMPI. It shows the importance of task granularity control and how worksharing tasks can reduce overheads. The third publication [27] presents the new TAGASPI library that eases the integration of tasking models such as OpenMP and OmpSs-2 with the one-sided primitives of GASPI.

We have participated in the minisymposium "Bringing Task-Based Programming to the Mainstream" collocated with the PASC 2021 conference. We presented the work done on the LoSync project to improve hybrid programming with the TAMPI and TAGASPI libraries.

We have had successful collaborations with other EU-funded projects. The TAMPI and TAGASPI libraries have been used in the DEEP-EST project to evaluate the Modular Supercomputing

Architecture (MSA). Specifically, TAMPI has been used to implement the communications required to split applications across different modules. The TAGASPI library is also being used in the EPEEC project.

5.4 Overall assessment of achievements and future developments

The project has developed the new TAGASPI library, and ported a wide range of benchmarks and mini-apps from pure MPI or conventional MPI + OpenMP to hybrid models leveraging the TAMPI and TAGASPI libraries. These include miniMD, Gauss-Seidel, miniAMR, HPCCG, N-Body, Lulesh, Matmul, and Co-MD. In many cases we have demonstrated performance gains from the task-aware communication library approach, especially in strong scaling scenarios where communication overheads start to dominate at high core counts. We have also put effort into improving the support for these models in the Extrae tracing facility, so that the behaviour and performance characteristics of these highly asynchronous codes can be more readily visualised.

This project has also allowed us to understand much more clearly the limitations of our approach and to focus on what needs to be done to improve it. Topics for further investigation include the following:

Software engineering issues

We did not succeed in implementing a large-scale application code using our programming model(s), as we underestimated the porting and refactoring effort involved in translating pure MPI or conventional MPI + OpenMP codes to fully exploit TAMPI or TAGASPI. In assessing some larger codes for porting feasibility, we noted that otherwise beneficial software engineering practices, such as encapsulating MPI communications, or allowing use of different intra-node parallel APIs (e.g. OpenMP and CUDA) from the same source base, significantly add to the complexity of porting. This is often a result of deep-seated assumptions being made that intra-node parallel computations and inter-node communications do not overlap with each other. The design of data structures can also be a poor match for the tasks with dependencies approach. For a successful implementation of a large scale application using TAMPI or TAGASPI, it would be far preferable to start with a task-based implementation in mind early in the design process, but this was not feasible with the effort available in this project. A topic for future work would be to design applications “ground-up” with the programming model specifically in mind.

Task granularity and locality

As with all tasking approaches, the granularity of tasks can strongly affect performance. With too many small tasks, the overheads of task creation and scheduling may dominate. On the other hand, too few large tasks risks leaving cores idle due to load imbalance or lack of available parallelism at certain points in the computation. Optimising the granularity typically requires parameterising the size of tasks, by grouping together multiples of the application’s “natural” tasks to form larger tasks.

In OmpSs-2 and OpenMP this often has to be done quite manually, adding to code complexity, and making reasoning about task dependencies more difficult. Including communication in tasks can compound the problem, due to internal synchronisation and contention for shared resources inside the communication library. This is especially a problem in most MPI implementations, and the main reason why TAGASPI can outperform TAMPI is thanks to the better thread-safety design in GASPI. We found that the fine-grained communication tasks naturally found in classical molecular dynamics codes such as miniMD and Co-MD were particularly problematic. The code paths in the

MPI library most exercised by TAMPI are not the same as the ones usually exercised by conventional MPI applications, and therefore may have not been optimised for multi-threaded execution. Future work will focus on optimizing the TAMPI library to reduce thread contention inside MPI.

If there are many different kinds of tasks in an application, all of which have parameterised granularity, we face a challenging optimisation problem to find the best set of task size parameters. Coupling parameters together, as we did for Lulesh, for example, in [26] can simplify the problem but may lead to a suboptimal solution. Exposing these parameters to an auto-tuning framework could be a possible way to handle this, but it would be even better if the "natural" tasks could be automatically coalesced by the runtime, though this is not trivial in the presence of complex dependency patterns.

Tasking is well known to be most beneficial for irregular computations. However, the use of TAMPI or TAGASPI has the potential to benefit regular applications by providing more effective overlapping of computation and communication than is possible with conventional MPI, for example. For such regular applications, the use of tasks can introduce unwanted overheads of task packaging and scheduling, as well as degrading locality (and therefore cache utilisation) due to the unpredictable patterns with which tasks are scheduled to cores. The use of worksharing tasks [26] goes some way towards solving this problem, but further research in this direction is required.

Library coverage

At present, the TAMPI library does not implement the whole of MPI, and in particular it would be useful to extend the coverage to include MPI 4.0 features such as MPI_info hints, and persistent collectives.

Tracing and debugging tools

When programming with TAMPI or TAGASPI, the out-of-order execution of both computation and communication tasks can make reasoning about both correctness and performance very hard for the application developer. Developing robust tool support is highly effort intensive, and in this project we have worked hard on improving support in the Extrae tracing package for TAMPI and TAGASPI, though more remains to be done.

6 FEM/BEM based domain decomposition solvers

6.1 Introduction and summary

ESPRESO [28] is a massively parallel framework based on the finite element method focusing on engineering applications. The main objective of its development team is to create a robust open source package applicable for a wide range of complex engineering simulations in areas such as mechanical engineering, civil engineering, biomechanics, and energy industry. It features scalable I/O tools, tools for mesh processing and morphing, finite element library, and massively parallel solver (see Figure 22 below). The solver parallelized in the distributed memory is based on the non-overlapping FETI (finite element tearing and interconnecting) domain decomposition method. In the past it has already shown an excellent scalability when applied to problems such as linear elasticity or heat transfer on machines like Titan at Oak Ridge National Laboratory or Piz Daint at CSCS [29]. The main goal of the project was to extend the framework's functionality to support massively parallel solution of harmonic analysis and acoustic problems, thus providing new capabilities in these engineering areas.

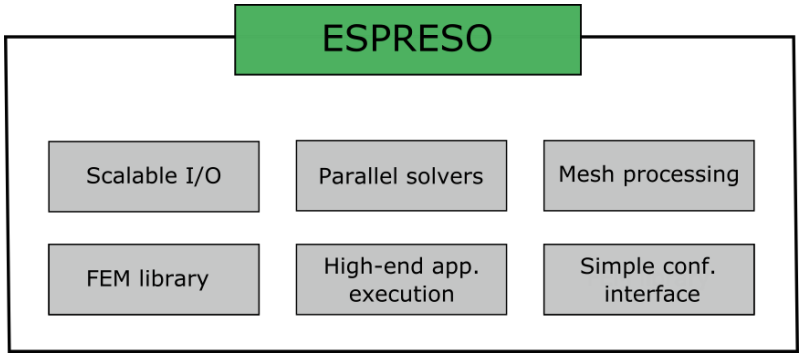


Figure 22: Capabilities of the ESPRESO library.

Within this project, several tasks had to be tackled, these included:

- The refactoring and optimization of the ESPRESO library;
- Development of the harmonic analysis and acoustic module;
- Acceleration of computationally intensive code using GPUs;
- Inclusion of the solver into the Solver as a Service online platform at IT4Innovations.

In Figure 23 we employ the developed software to compute the frequency response of the electric motor case discretized into 15 million degrees of freedom.

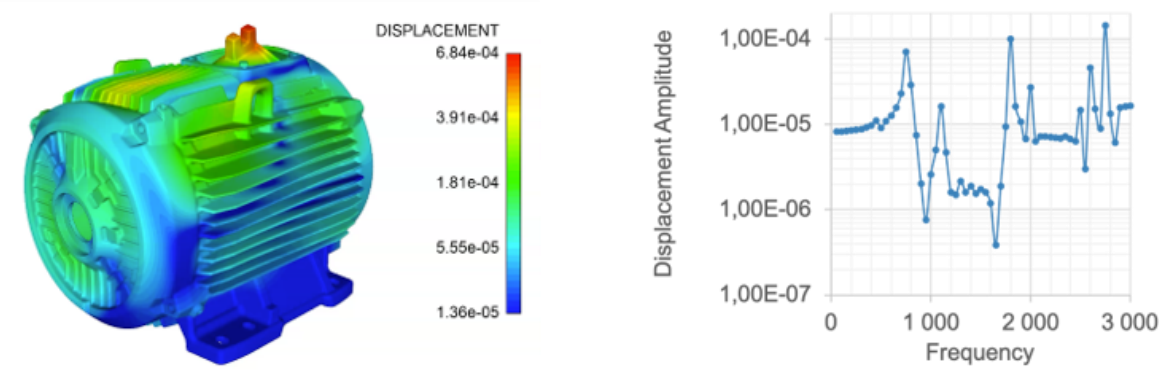


Figure 23: Frequency response of the electric motor case computed using 450 nodes of the Salomon cluster at IT4Innovations in 714 s (15 million DOFs, 60 frequency samples).

6.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

The code has been tested on several European systems, including Salomon and Karolina at IT4Innovations National Supercomputing Center in Czech Republic or JUWELS Cluster and JUWELS Booster modules at Jülich Supercomputing Center in Germany. The Salomon cluster consists of 1009 nodes, each equipped with two 24-core Intel Xeon E5-2680v3 processors and 128 GB of RAM. The Karolina cluster consists of 829 compute nodes, totalling 106752 cores, giving over 15.7 PFLOP/s theoretical peak performance. The JUWELS Cluster module is equipped with 2271 standard nodes with two 24-core Intel Xeon 8168 CPUs and 96 GB of RAM. The JUWELS Booster module consists of 936 nodes with two AMD Epyc Rome 7402 CPUs, 512 GB of RAM and four NVIDIA A100 GPUs.

Parallel performance of the harmonic analysis module

The harmonic analysis module allows combined parallelization both in spatial and frequency domains. This significantly improves scalability on large machines. On the other hand, load balance may be affected since iterative solution of respective linear systems on nodes handling frequencies close to eigenfrequency requires a larger amount of iterations. The combined spatial-frequency decomposition was benchmarked on the Salomon cluster using cubical domain with 43 million degrees of freedom, testing frequency response in 36 sample frequencies ranging from 0 to 15000 Hz. We used six groups of 72 compute nodes (10368 MPI processes in total). Within each group, the whole spatial domain was decomposed using the FETI domain decomposition method and 6 frequencies were resolved. Duration of iterative solvers for individual frequencies is depicted in Table 5. While the load balance is far from optimal, it still enables us to solve significantly larger problems than using only spatial or frequency decomposition separately.

nodes 0-72	nodes 73-144	nodes 144 -216	nodes 145 -288	nodes 289 - 360	nodes 361 - 432
24.965	27.992	31.528	27.784	30.174	32.645
24.988	27.844	27.55	27.708	27.378	32.089
24.73	27.915	29.295	28.404	24.379	30.626
25.069	28.003	28.397	28.226	27.799	38.301
24.643	31.793	30	28.797	33.298	29.451
25.064	30.782	27.351	30.394	32.792	28.443
Total time					
149.459	174.329	174.121	171.313	175.82	191.555

Table 5: Computation of the harmonic analysis with combined spatial-frequency domain parallelization. Solution of the system using iterative solver in [s].

Scalability of the GPU accelerated code

Since the heterogenous architectures are currently prevailing in HPC, the code has to be accelerated using GPUs in order to leverage the full power of modern supercomputers. We employ the local Schur complement approach within our FETI solver. This enables us to work with dense Schur complement matrices instead of sparse system matrices which are common in FEM. Therefore, we are able to replace direct solution of large sparse systems by iterative solution with small dense matrices which can be efficiently performed on GPUs.

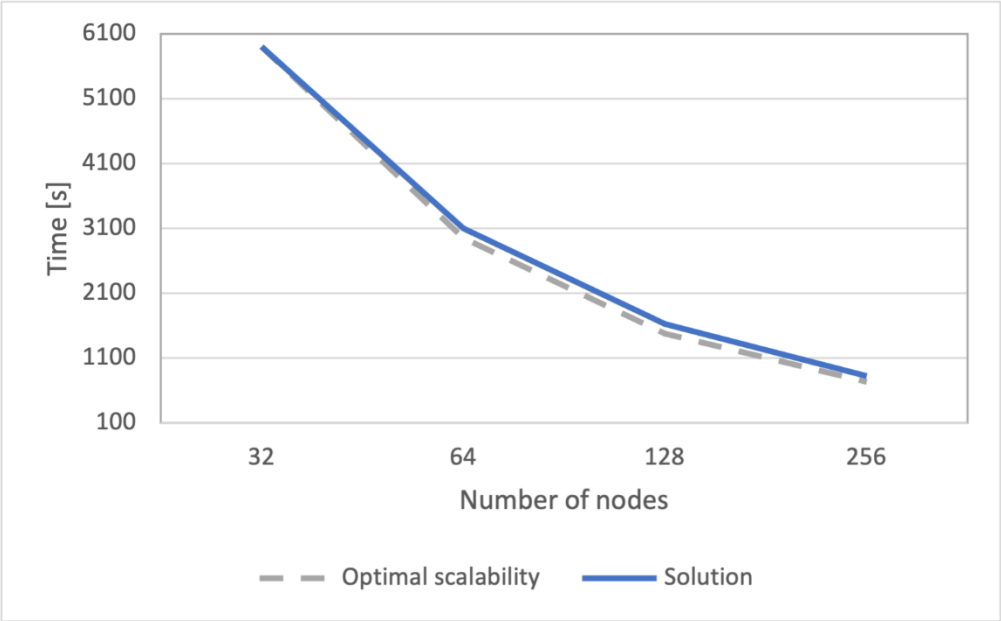


Figure 24: Strong parallel scalability of the harmonic analysis solver on JUWELS Booster module.

Strong parallel scalability of the accelerated code tested on JUWELS Booster module in Jülich Supercomputing Center is depicted in Figure 24. For this test we used a spatial domain decomposed into 6 million degrees of freedom. The parallel efficiency reaches approximately 90% on 256 nodes.

Performance of optimized I/O

The ESPRESO package is primarily an engineering software. Since the engineering meshes are usually stored in unstructured sequential database files often containing hundreds of millions nodes, we provide an interface for their efficient parallel loading and manipulation. This interface has been optimized within the project and extended to support additional file formats. Scalability of the individual phases of mesh manipulation on the JUWELS Cluster module is depicted in Figure 25.

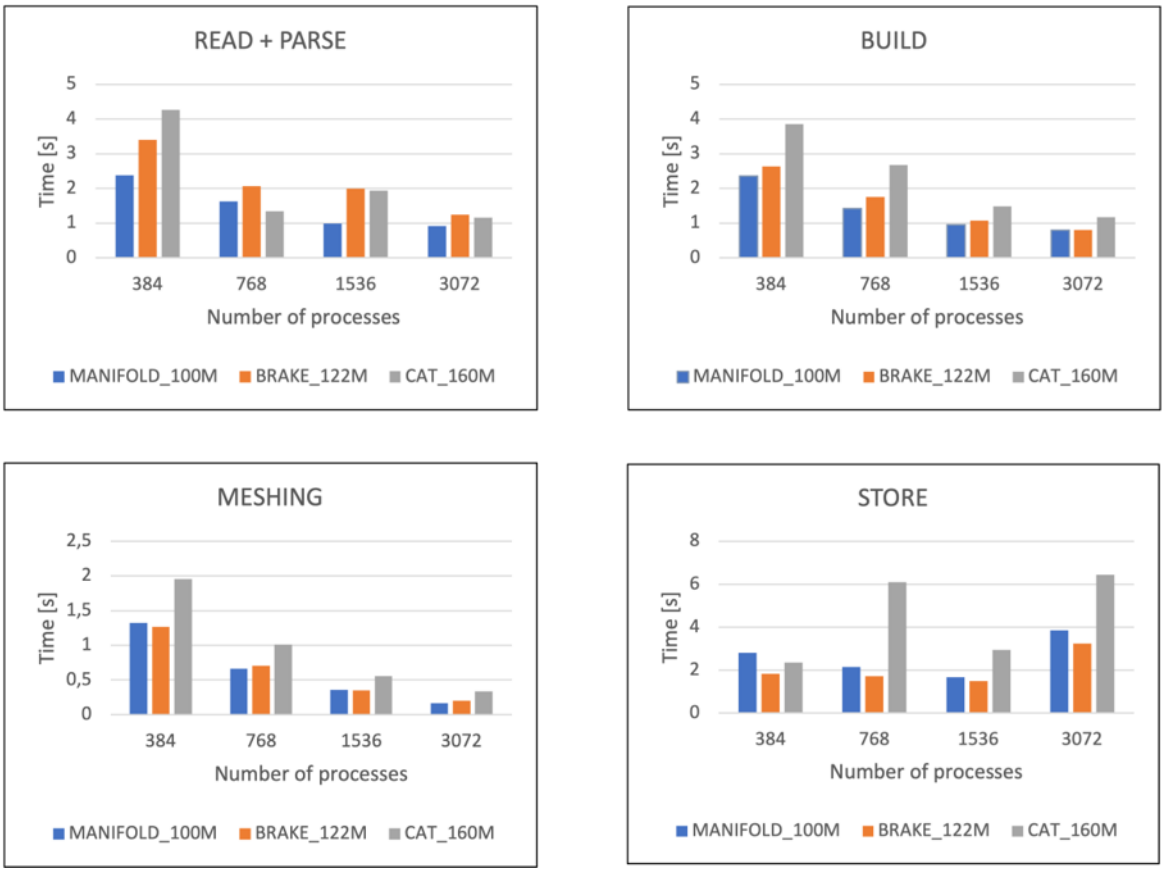


Figure 25: Scalability of the individual phase of mesh manipulation on the JUWELS cluster module

Reading the mesh from the drive initially scales well, however soon reaches the limit given by the underlying hardware. During the BUILD phase the mesh is assembled in parallel from the loaded data and the MESHING phase prepares the mesh for usage in the parallel solver. These phases scale well until approximately 3000 processes when the parallel overhead starts to dominate. Finally, the STORE phase is again limited by the capabilities of the underlying hardware.

Similar results were obtained on the Karolina cluster at IT4Innovations. In Figure 26 we only present the behaviour of the READ + PARSE phase and MESHING phase. While the first phase

reaches the hardware bandwidth limit at approximately 1024 - 2048 MPI processes, the MESHING phase scales well up to 4096 MPI processes.

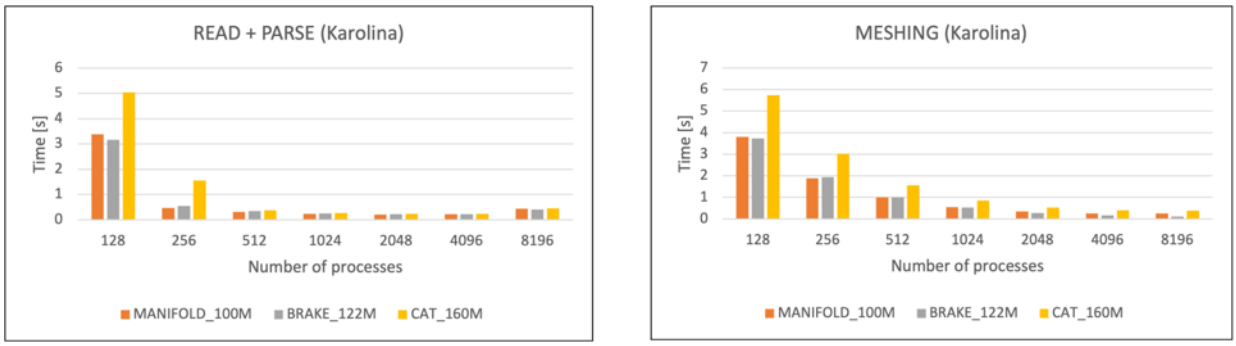


Figure 26: Scalability of the READ + PARSE and MESHING phases on the Karolina system

6.3 Interactions with stakeholders, users, outreach and publications

Within the EXPERTISE project [30], IT4I has established cooperation in the application of parallel solution of harmonic problems and its application especially to nonlinear harmonic balance method. EXPERTISE is a European Training Network (ETN) that will contribute to train the next generation of mechanical and computer science engineers with a common basic knowledge on the challenges, the paradigms, the technologies and the methodologies in the field of nonlinear structural dynamics of turbomachinery and High Performance Computing. The results achieved within the PRACE project were presented at regular project meetings and the findings were consulted with individual members of WP3 - Structural dynamics of turbine and its components.

IT4Innovations national supercomputing centre has an ongoing cooperation with SIEMENS Czech Republic. Siemens Electric Machines s.r.o., Frenštát is one of the leading producers of low-voltage asynchronous electric motors. Their primary customers are producers of pumps, compressors, and air-conditioning equipment. One of the priorities of the Siemens company includes the production of electric motors with highly efficient cooling systems, allowing smooth operation of these machines even under extreme conditions. In cooperation with experts from the Siemens company, we are pursuing development of a digital twin of an electric motor in order to improve the efficiency of asynchronous electric motors. Within the national project focused on the development of a digital twin of the electric motor, the results of the PRACE project are applied to calculations of the structural harmonic response from forces generated by electromagnetism. The project is funded by the Ministry of Industry and Trade of the Czech Republic. We would like to apply the ESPRESO solver also to solve acoustics generated by electromagnetism.

Effective utilization of HPC systems has a significant impact on the validity of spending financial resources into HPC infrastructures from a global point of view.

In the context of the digital twins' concept, the use of the ESPRESO framework will allow a combination of numerical simulations of complex physical problems and a deep learning approach. This approach will consist of using the ESPRESO framework to create a large number of data sets containing the results of complex simulations for a wide range of initial and boundary conditions. These data sets can then be used to train neural networks that can then be used for simulation and prediction of the behaviour of a given product represented by a digital twin in real time.

Within the activities of the National Competence Centre of the EuroCC project under the European Union's Horizon 2020 [31], Digital Innovation Hub Ostrava [32], and IT4Innovations provides knowledge, technology transfer and services based on PRACE project results to IT4Innovations clients and as such contributes to the improvement of their processes, products, and services and thus provides added value for their customers and society at large, and creates new business opportunities.

ESPRESO is also newly involved in the European Pilot for eXascale (EUPEX) project. The objective is to enable the ESPRESO FEM framework on the EUPEX modular architecture and to optimize its performance for the SiPearl Rhea chips in order to leverage both SVE vectorization and HBM memory, as the ESPRESO solver is mostly memory bound.

Due to the Covid-19 related travelling restrictions, most of the planned conferences have been cancelled. However, the ESPRESO package was presented, e.g., at the Supercomputing 2019 conference in Colorado, Denver.

6.4 Overall assessment of achievements and future developments

Within the project, several goals specified in the proposal have been tackled:

Refactoring and optimization of the original code:

We have optimized the parallel input workflow and the global matrix operations, created new interfaces to external solvers and mesh partitioners (such as Pardiso, SuperLU, Watson Sparse Matrix Package, HYPRE, PT-Scotch), and redesigned the ESPRESO configuration file. We have also optimized the system matrix assembler by replacing the BLAS routines by manually tuned code for small matrices. We have also extended the functionality of the code to support mortar type gluing conditions enabling computation with non-matching meshes.

Development of the harmonic analysis module:

The ESPRESO module for harmonic analysis has been implemented. It supports hybrid MPI and OpenMP parallelization as well as acceleration using GPUs. Within its development, the main obstacle was development and implementation of the regularization of the system matrices and preconditioner by the artificial coarse problem. We have implemented three approaches for assembly of the artificial coarse problem that can be used as a preconditioner for various types of problems. Their development and debugging slowed down the overall progress of the project.

GPU acceleration of the code:

Some of the computationally intensive parts of the code have been accelerated using GPUs. We have accelerated the solver part using the local Schur complement approach that enables us to replace the large sparse system matrix by smaller dense matrices more suitable for GPU processing. The scalability of the accelerated code has been tested on clusters equipped with NVIDIA A100 GPU.

Development of the acoustic module:

Due to the issues with the regularization and coarse problem assembly for the harmonic analysis module, the development of the acoustic module was delayed and has not yet been finished. Currently, the acoustic problems are treated using direct a sparse solver which limits the maximum problem size and parallel scalability. We plan to continue working on the module after the end of the PRACE project.

Incorporation into the Solver as a Service platform at IT4Innovations:

Although the original plan was to include the harmonic analysis solver into the Solver as a Service (SaaS) platform at IT4Innovations and enable users not familiar with HPC to use the software on a supercomputer, due to a developer leaving the team, the implementation of the SaaS platform is not yet finished. Its development will continue in future.

Comparison with the boundary-element-based solvers

This part has been cancelled in order to fully focus on the development of the harmonic analysis module.

7 Performance portable linear algebra

7.1 Introduction and summary

Scientific applications, especially Material Science ones, rely heavily on Linear Algebra to tackle complex tasks. A key issue, both in terms of importance and of performance, is the solution of distributed dense eigenvalue problems. For example, modern electronic structures methods, like Density Functional Theory (DFT), manipulate many-body Schrödinger equations to obtain (dense or sparse) eigenvalue problems or linear systems.

ScaLAPACK [33] is considered the de-facto standard library for distributed linear algebra. However, its performance is strongly limited by modern supercomputer architectures: multi-socket nodes and multi-core CPUs (replacing single-core nodes) and the availability of GPU accelerators make the fork-join mechanism unscalable. Along with communication minimization strategies, task-based libraries improve efficiency by executing in parallel independent tasks, on different cores.

The goal of this project is to provide a modern and efficient distributed task-based linear algebra package, called DLA-Future (DLAF), based on HPX [34], a tasking library. The main goal of DLAF is the implementation of a generalized eigenvalue solver for Hermitian matrices.

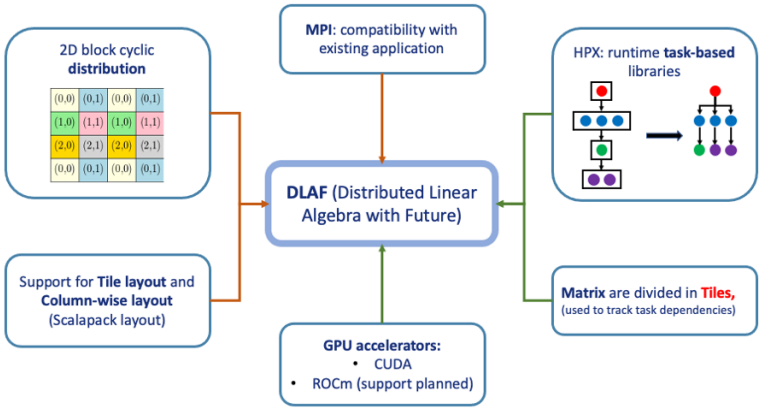


Figure 27: Overview of DLA-Future

HPX futures [35] allow tasks of different routines/algorithms to run concurrently, creating a single dependency graph for the full application. The choice of MPI as the communication library allows

DLAF to be compatible with existing applications. GPU acceleration is exploited by relying on CUDA and cuBLAS: a custom standard-conforming asynchronous API has been specifically written and will be upstreamed to HPX. Matrices are divided into submatrices, called tiles, distributed according to a 2D block cyclic distribution scheme and with column-wise layout (both for tiles and their elements). Algorithms have been rearranged to take advantage of this tile layout, further enhancing performance.

An alternative strategy in the development of a distributed eigensolver for dense eigenproblems is to leverage well-known and well-established iterative algorithms such as subspace iteration, e.g., the Chebyshev Accelerated Subspace iteration Eigensolver (ChASE) library [36]**.** When tackling sequences of Hermitian eigenproblems, as they often appear in electronics structure codes, ChASE takes advantage of the distinctive features connecting adjacent problems in a sequence. ChASE is able to scale well on large-scale distributed supercomputers because of its algorithmic design. The most important kernel in ChASE is the Hermitian Matrix-Matrix Multiplication (HEMMs). As a typical BLAS-3 operation, the performance of an efficient implementation of distributed HEMM is able to approach the peak performance of any system.

The objective of this project regarding ChASE is to port it onto modern distributed multi-GPUs supercomputers with the support of multiple data distribution geometries: custom-block, block-cyclic, and element-wise cyclic (Elemental). ChASE is expected to feature a parallel MPI-CUDA hybrid execution on distributed many-core clusters with multiple GPUs per node. Another important objective is to redesign and develop a more complete and user-friendly documentation for ChASE, which includes more examples to use this library as a standalone solver as well as integrating it in application software, especially in the domain of Condensed Matter Physics.

7.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

DLA-Future:

We executed a strong scaling and a weak scaling analysis of the DLA-Future Cholesky implementation on the following systems and we compared the performance achieved with the state-of-the-art libraries available. Figure 27 presents the results on Piz Daint MC, a Cray XC40 system whose nodes are equipped with two 18 cores Intel Xeon E5-2695 v4 running at 2.10GHz and 64/128 GB of DDR4 memory. We compare the results with optimized implementations of ScaLAPACK, SLATE and DPLASMA.

We benchmarked each library using different values for the blocksize of the 2D block-cyclic distribution (for each series of data the blocksize is indicated in the legend as the number inside the parenthesis). DLA-Future performance is better than ScaLAPACK and SLATE and very similar to DPLASMA. However, DPLASMA weak scaling results show a sudden drop in performance with 256 and 384 block size.

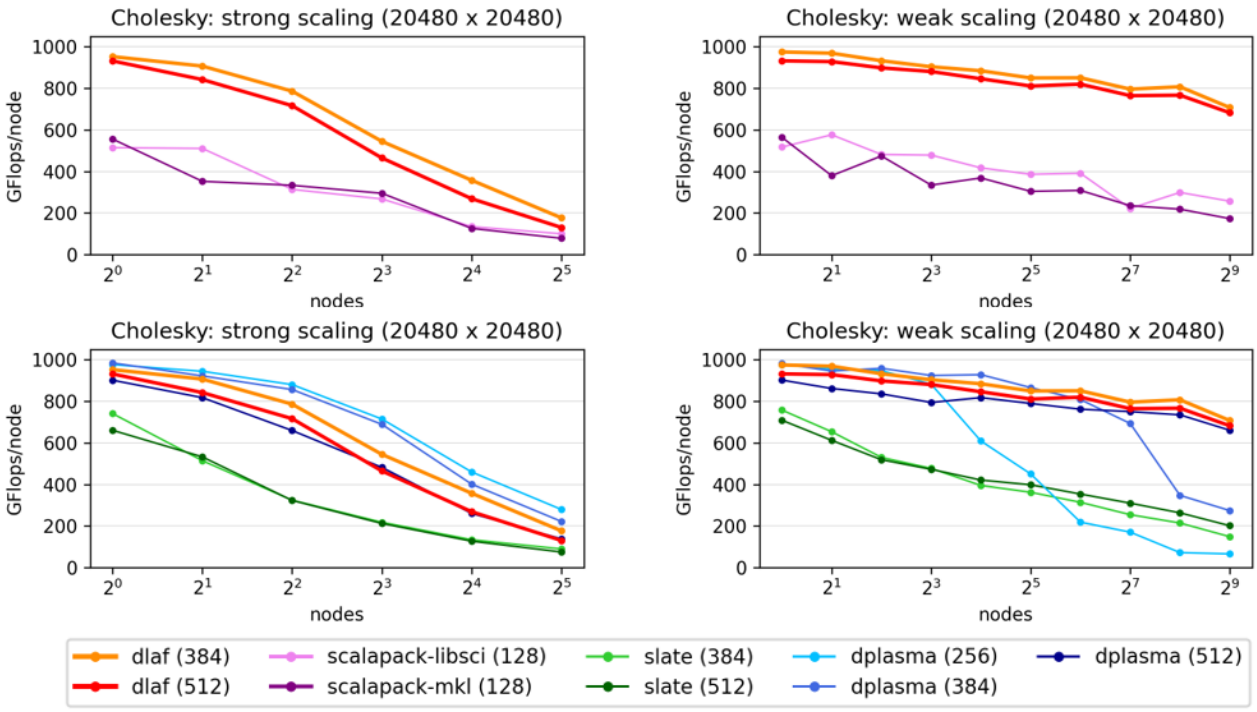


Figure 28: Cholesky factorization on Daint MC. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node.

Figure 28 presents the results of the same problems on Daint GPU, a Cray XC50 system whose nodes are equipped with a 12 cores Intel Xeon E5-2690 v3 running at 2.60GHz, 64 GB of DDR4 memory and a NVIDIA P100 GPU (16 GB HBM2 memory). Similarly, we present the strong and weak scaling with a matrix of size 20k.

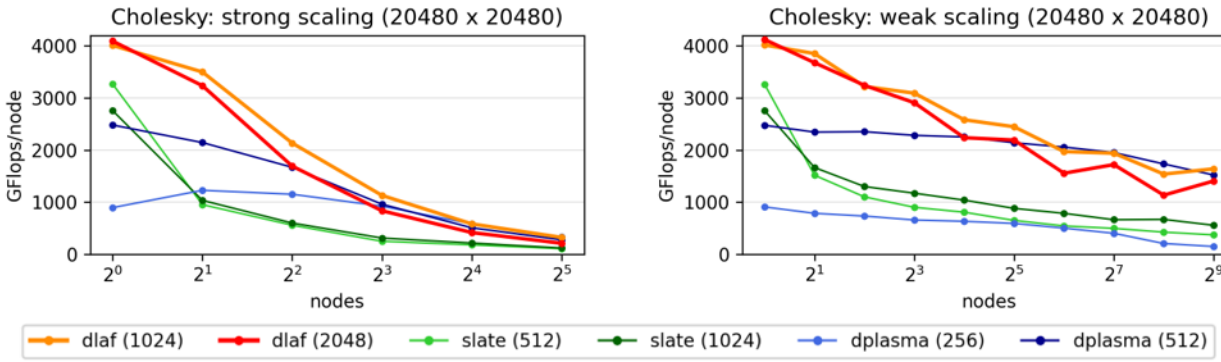


Figure 29: Cholesky factorization on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node).

DLA-Future clearly performs better than the competitors. It can be noted that we presented results for DPLASMA only with small block sizes. Unfortunately, any attempt to run it with block size of 1024 or 2048 resulted in a failure.

Figure 29 presents the results of Cholesky factorization on Marconi100 system, an IBM POWER9 system whose nodes are equipped with two 16 cores IBM POWER9 AC922 running at 3.10GHz, 256 GB of memory and 4 NVIDIA Volta V100 GPUs (16 GB memory).

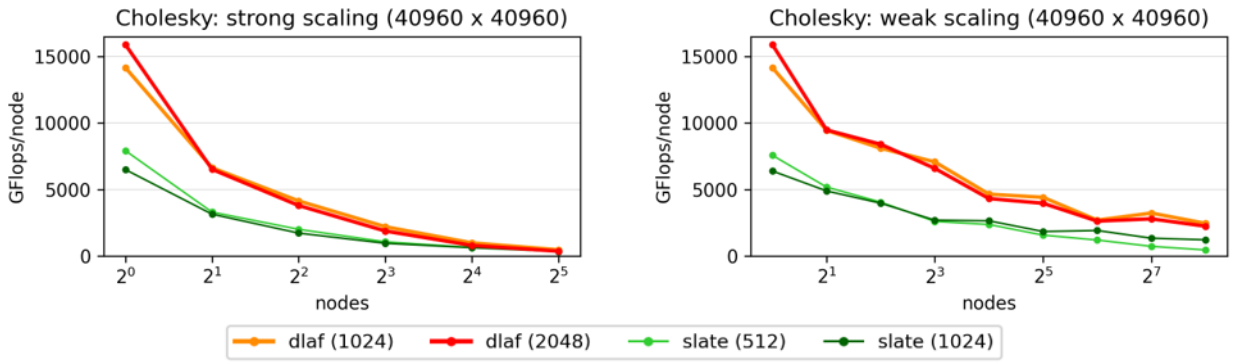


Figure 30: Cholesky factorization on Marconi 100. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node).

The results show a clear performance advantage of DLA-Future compared to SLATE. We attempted to compare the results against DPLASMA as well, but all the combinations of nodes and block size we tried resulted in a failure of this library. The results of the other algorithms we present have been produced on Piz Daint GPU. Figure 30 presents the scaling of the triangular solver.

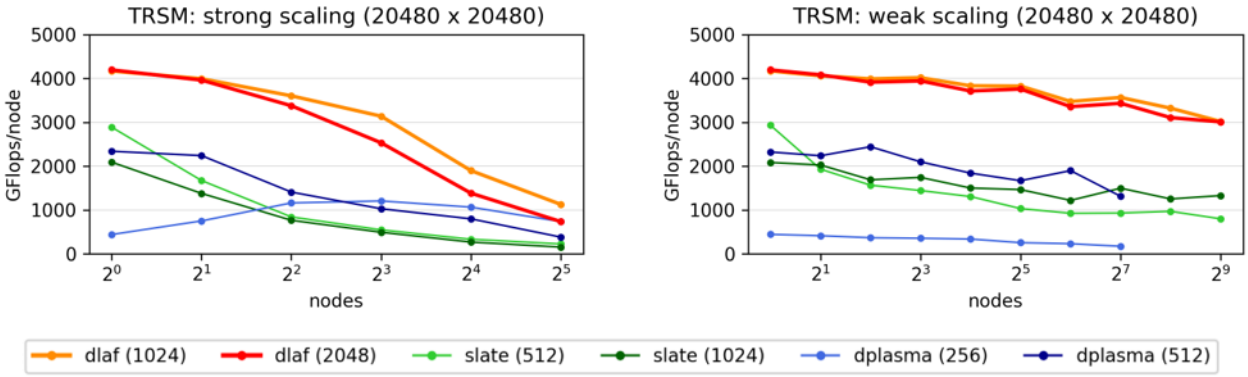


Figure 31: Triangular solver on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node).

DLA-Future performs better than the two competitors. In certain cases, DLA-Future is twice as fast as DPLASMA and SLATE. Moreover, DPLASMA runs with larger block-sizes (1024 and 2048) failed as weak scaling runs with 256 and 512 nodes.

Similarly Figure 31 shows the result of the transformation from generalized to standard eigenproblem. The results of DLA-Future are very good and better than SLATE. (note: this algorithm is not implemented in DPLASMA). As the performance of SLATE is poor (we didn't go beyond 32 nodes in the weak scaling because the performance is too low) and in the best cases equal or generally lower than the performance of the CPU available in the system, one may think that the GPU is not used. A trace of the SLATE algorithm shows that some operations are executed

on the GPU, however it shows a lot of host/device copies as well, which might be the reason for the poor performance.

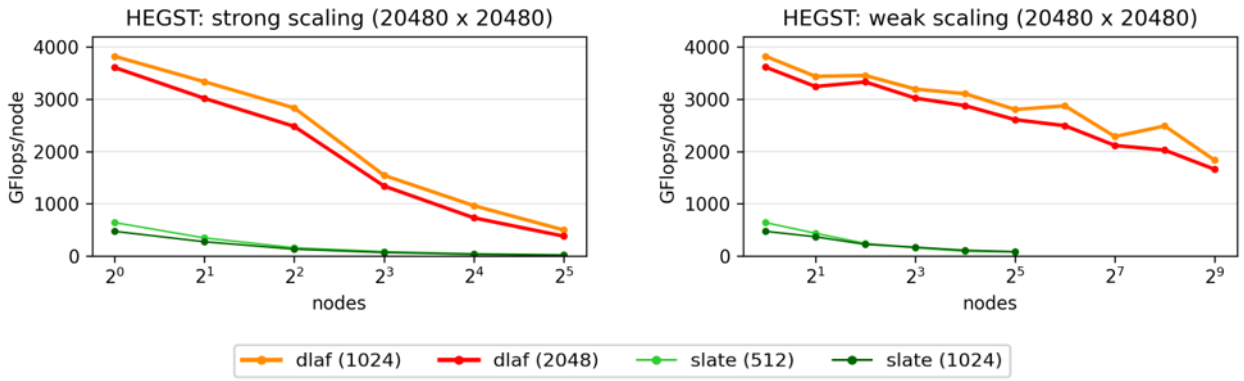


Figure 32: Transformation from generalized to standard eigenproblem on Daint GPU. Left: we present the strong scaling for a matrix of size 20k. Right: we present the weak scaling for 400M elements per node (20k x 20k matrix for the run on a single node).

Figure 32 demonstrates the advantage of task based libraries. The traces show the execution of 4 independent Cholesky factorizations of matrix size 10k on 4 nodes of Piz Daint MC. To simplify the figure, the trace presented represents only the threads and the tasks executed on a single rank. The figures for the remaining ranks are similar.

The upper trace illustrates the case in which a synchronization is put after each factorization (simulating standard libraries), while the trace below shows how a full task based application can improve the execution time. For 4 independent Cholesky factorizations of size 10k, the execution time drops from 1.57 seconds to 0.98 seconds.

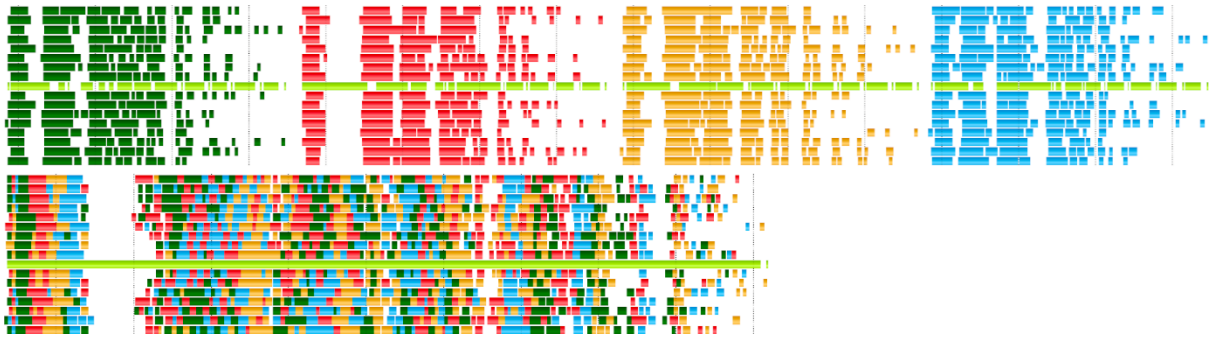


Figure 33: Trace of 1 rank (over a total of 4) for the execution of 4 independent Cholesky decompositions. The tasks of each factorization are depicted with a different color. (Due to a limitation of the trace utility, MPI communications cannot be identified, therefore they are all colored in light-green). The trace above shows the case in which after each factorization a synchronization point is added, the trace below shows the case in which the factorizations are allowed to overlap.

ChASE:

ChASE has been comprehensively tested on all major clusters (JUWELS Cluster and Booster, JURECA-DC and JUSUF) at the Juelich Supercomputing Centers (JSC). Thanks to the variety of the JSC clusters, ChASE is ported and ready to work on both Intel-based and AMD-based CPUs together with NVIDIA multi-GPUs.

For the purpose of this report, we benchmarked ChASE's behaviour in the strong and weak scaling regime. For all the tests, we selected 4 MPI ranks per node, with 1 GPU and 32 threads assigned to each rank. We present here results of the benchmark of ChASE executed on the JURECA-DC supercomputer. Each node of JURECA-DC is equipped with two 64 cores AMD EPYC 7742 CPUs @ 2.25 GHz (16x32GB DDR4 Memory) and four NVIDIA Tesla A100 GPUs (4x40GB high-bandwidth memory). In these benchmarks, ChASE is compiled with GCC 9.3.0, OpenMPI 4.1.0 (UCX 1.9.0), CUDA 11.0 and Intel MKL 2020.4.304.

Strong scaling

Figure 33 below illustrates the results of the strong scaling experiment of ChASE using an artificial matrix of size N = 130000. The subspace dimension of ChASE is fixed as 1300, which is 10% of the matrix size. The counts of compute nodes are selected to be square numbers 1, 4, 9, ..., 64. This figure reports the runtime of ChASE as a vertical stacked bar plot, which also includes the fractions of runtime of numerical modules, such as Filter, Lanczos-based DoS (Lanczos), QR factorization (QR), Rayleigh-Ritz (RR) projection and Residual computation (Resid). As it is visible from the leftmost bar in the plot, the computation is heavily dominated by the Filter. Consequently, achieving a good scaling for this module is paramount. Within the Filter the most important kernel is the execution of repeated calls to HEMM within a 3-terms recurrence calculation. The blue bar in the plot shows that our customized distributed multi-GPUs HEMM, achieves very good strong scaling performance.

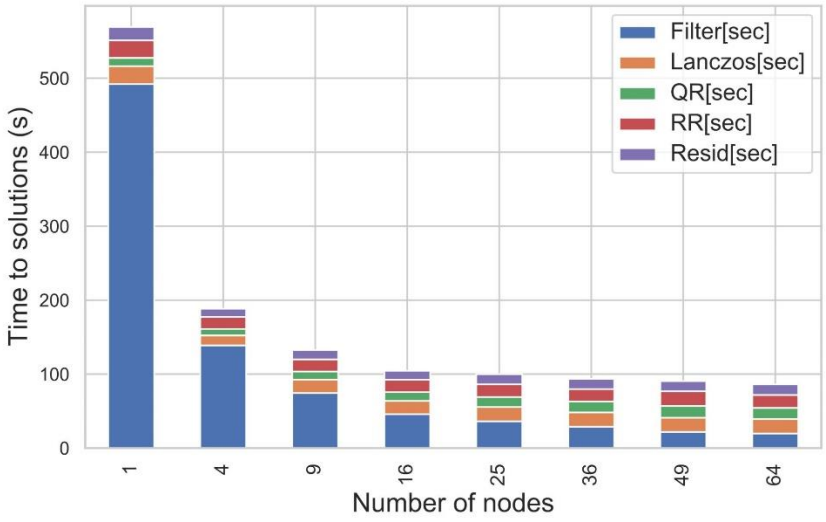


Figure 34: Runtime of ChASE as a vertical stacked bar plot, including includes the fractions of runtime of numerical modules

For the Filter, ChASE with 64 nodes achieves 8x speedup over the test with 1 node. As the impact of Filter on the overall computational time decreases with the increase of the number of nodes, other BLAS/LAPACK operations in ChASE (QR and part of RR), which were computed redundantly, become the new bottlenecks. For other modules such as Lanczos and Resid, which also employ the multi-GPUs HEMM, only 1.2x speedup has been attained. These are not as scalable as the Filter due to their limited operational intensity. As shown in Figure 34 below, the distributed multi-GPUs version of ChASE with 1 compute node has the maximal speedup over the CPU version, which is 8.1. With the increase of count of compute nodes, the speedup finally keeps constantly ~5.

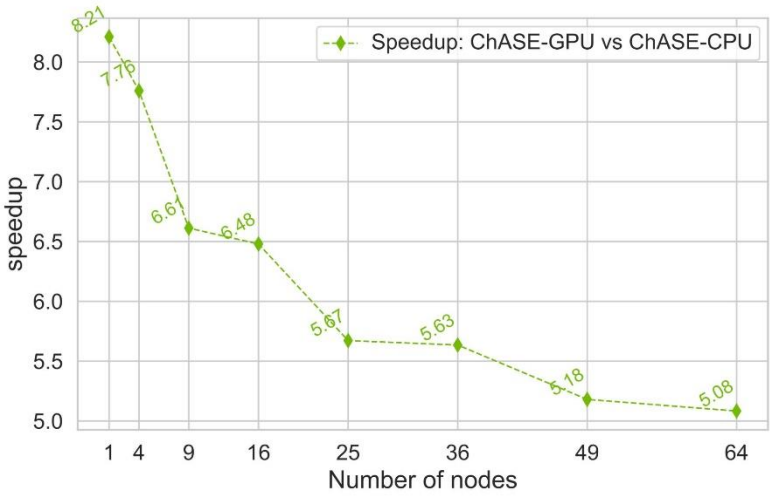


Figure 35: Speedup of the ChASE GPU implementation compared with the ChASE CPU implementation

Overall, ChASE achieves good strong scaling performance at the beginning. However, with the increase of the number of compute nodes, the decrease of total runtime of ChASE becomes negligible.

Weak scaling

Figure 35 below shows the results of the weak scaling experiments. The weak scaling tests have been set up so that only one iteration has been performed without full convergence. This ensures a constant workload for the Filter. The test matrix sizes are 30k, 60k, 90k, …, 360k, with the numbers of compute nodes 1, 4, 9, …, 144, respectively. The maximal size of active subspace is fixed as 3000. Overall, the runtime of ChASE increases with the augmentation of problem size and compute node count. ChASE time-to-solution increases 4.6x, when the node count increases from 1 to 144. Despite the overall trend, the weak scaling performance of the customized distributed multi-GPUs HEMM is good, which confirms the efficiency of its implementation. With the increase of problem size, QR and RR, which are executed redundantly on each node, become more and more dominant. In order to improve on these results, we plan to redesign the algorithmic realization of less scaling modules and explore the possibility of a task-based approach.

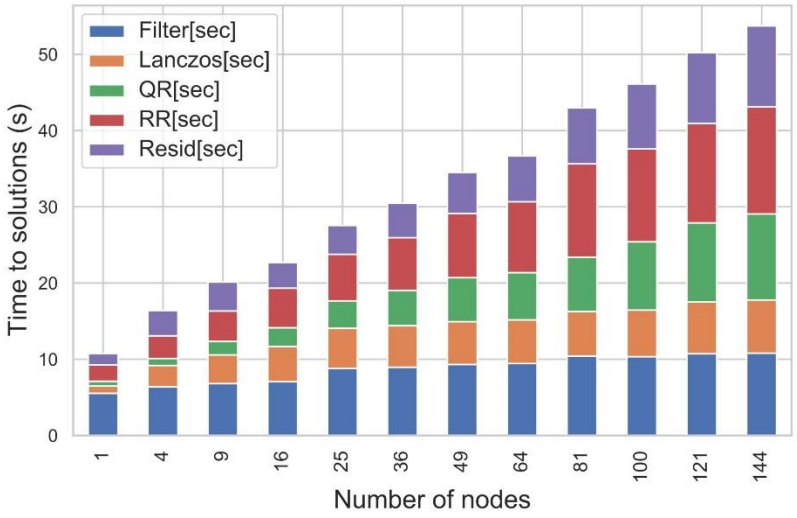


Figure 36: ChASE weak scaling results across increasing number of nodes

7.3 Interactions with stakeholders, users, outreach and publications

DLA-Future is a newly developed library which wants to provide a task-based implementation of the eigenvalue solver. As only some of the algorithms are available, it is too soon to consider integrating the library in real applications.

A poster regarding the development of the library has been submitted and accepted at the PASC20 conference. Unfortunately, due to the Covid 19 pandemic the conference has been cancelled and the poster has been presented at PASC21.

ChASE is a relatively new library and as such it is only recently being viewed as a replacement for more seasoned and well-tested ones. Despite its young age, ChASE has been already integrated in two active codes in the community of Condensed Matter Physics. In a recent publication [37], we showcased the improvements in performance when ChASE is used to solve the Bethe-Salpeter Equation (BSE) in a code developed at the University of Illinois Urbana-Champaign. Moreover, such integration revealed that ChASE enables the BSE code to increase the size of the physical system and explore new physical phenomena that were previously unreachable due to the limited scaling of the previously used eigensolver.

Similarly, ChASE has recently been integrated in the FLEUR code, a work horse in the community of Density Functional Theory which is also part of the MaX Center of Excellence. The further integration of the new multi-GPUs porting of ChASE is currently underway. In addition, we are actively discussing with the developers of the Yambo and Quantum Espresso code on how to integrate and test ChASE on these two codes.

A very active role in this project as an external partner is played by the Ruđer Bošković Institute (RBI) in Croatia. Its main PI, Davor Davidovic, has supported and substantially contributed to the multi-GPU HEMM implementation used in the Filter. In collaboration with the RBI, we carried on an extensive benchmarking of ChASE. The results of this study, some of which are included in this report, were submitted to the Proceedings of the SIAM conference on Parallel Processing to be held in Seattle in March 2022.

7.4 Overall assessment of achievements and future developments

DLA-Future:

The objectives of the original proposal have been only partially achieved. The full pipeline for the generalized eigensolver is not available.

We completed the work (distributed multicore and GPU implementation) on the following algorithms: Cholesky factorization, transformation from generalized to standard eigenproblem, triangular solver. Moreover, a local implementation is available for the two stage tridiagonalization and relative back-transformations. The distributed versions of these algorithms are still in development. Finally, the development of the tridiagonal eigensolver just started. Similarly, the support for AMD GPUs is not yet available, but it is planned to be added in the next months.

The reasons for the delay in the development are multiple. On one side the missing functionalities of the tools available made the development more difficult, for example:

- The use of a C++ heavily templated library (HPX) increases the complexity of the error messages generated by compilers. A small syntax error may end up in multiple pages of error messages, in which the useful information is hidden.
- Debugging tools specific for task based programs are not available. Standard debugging methods are not suitable with task based programming. E.g. the stack, for non-task based applications, helps to identify the position in the code, the parameters used, etc., while for task-based applications it just contains the isolated information related to the current task which is executed.
- Profiling and tracing tools are still in development. In particular APEX, the tracing tool shipped with HPX, is not mature enough to give all the information needed to identify performance bottlenecks.

On the other hand, some tasks required more effort. E.g. the move from MPI synchronous collective communication to asynchronous collective communication introduced performance problems which haven't been identified in the tests done with the prototype.

Finally, we are using some HPX APIs which are not yet standardized in C++. This introduces the risk that these APIs are modified before the standardization occurs. This happened with the promise/future concepts, which have been superseded by the senders/receivers concepts [38]. We had, therefore, to conduct some experiments with the new APIs in DLA-Future to ensure that the functionalities needed are still available and the performance is not affected. Future work on the library will concentrate on the completion of the eigensolver.

ChASE:

The objectives of the original proposal have been achieved. We developed a customized distributed multi-GPUs HEMM for ChASE, and offloaded selected BLAS/LAPACK operations onto GPUs. Moreover, the block-cyclic data distribution scheme has been also included into ChASE, which makes it easy to be integrated into other applications and libraries which mostly employ this ScaLAPACK-style data distribution. The experiments show a good parallel performance of this implementation of distributed multi-GPUs HEMM. The version of ChASE ported to distributed multi-GPUs supercomputers achieves large speedup compared to the CPU version. However, few weaknesses have also been identified, which we were not aware of at the time the original proposal

was written. This was in part possible because ChASE was ported to the newest massive parallel heterogeneous architectures.

After offloading HEMM to GPUs with excellent parallel efficiency, the QR factorization and the Rayleigh-Ritz projection, which make use of BLAS/LAPACK multi-threaded routines executed redundantly on the node, become new bottlenecks. This is quite important when dealing with matrices of size larger than 100k or requiring the computation of a relatively large fraction of eigenpairs. In future, we plan to provide customized partially distributed QR and Rayleigh-Ritz modules for ChASE. The reasons behind the customization have to do with the particular type of QR factorization ChASE does: it is not enough to execute a standard update on the Q matrix because of the deflation and locking mechanism change the locked eigenvectors. Moreover, distributing over the entire MPI grid would enormously increase the communication and erase the benefits of the distributed parallelization. We plan, instead, a hybrid approach maintaining a careful balance between redundant computation and a partial distribution over an optimal subset of the MPI grid.

Another promising development in our future plans is to integrate a well-designed rational filter [39] into ChASE which is able to compute the eigenpairs within any given interval, rather than the extremal ones. When an eigenproblem can be split into multiple small intervals and solved in parallel, the bottlenecks of QR factorization and Rayleigh-Ritz can be removed.

The distributed multi-GPUs version of ChASE supports a flexible configuration of binding multiple GPUs to each MPI rank, however, current benchmarks show that the configuration which binds 1 GPUs to each MPI ranks outperforms the others. This is caused by the overheads of communications between GPUs within each node. In the future, we will explore GPU-aware MPI for the direct communications between GPUs for the implementation of distributed multi-GPUs HEMM of ChASE.

8 LyNcs: Linear Algebra, Krylov methods, and multi-grid API and library support for the discovery of new physics

8.1 Introduction and summary

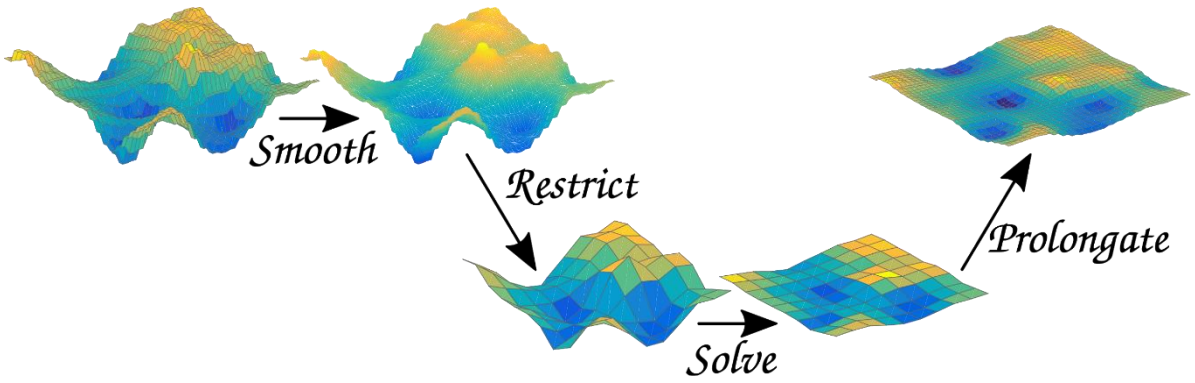


Figure 37: Important algorithmic steps in the Krylov accelerated multigrid solver developed in the LyNcs project

The project, Linear Algebra, Krylov-subspace methods, and multigrid solvers for the discovery of New Physics (LyNcs), is addressing challenges encountered with parallel iterative solvers for large sparse matrices which arise in computational physics on modern and upcoming architectures due to massive parallelization. LyNcs is targeting efficient solutions for linear systems for large sparse matrices by pooling together software development efforts across Europe. LyNcs intends to provide the European communities with the next generation of parallel libraries for solving sparse linear systems at the Exascale. The project is led by the Computation-based Science and Technology Research Centre (CaSToRC) of The Cyprus Institute, which joins forces with partners from the French Institute for Research in Computer Science and Automation (Inria) and the Leibniz Supercomputing Centre (LRZ).

Along the duration of the project, LyNcs has targeted the design of new software, the development of existing libraries and the necessary research on new algorithms for the solution of large sparse matrices. Part of LyNcs is the development of an API that is targeting massively parallel machines to perform advanced task management with shared and distributed memory among huge parallel partitions. This LyNcs API together with implementing cutting-edge sparse linear solver algorithms, the development of novel block Krylov solvers and optimization of existing parallel codes will enable community software to efficiently utilize the up-coming pre-exa and exascale machines. The software improvements target all levels of the scientific application software stack, from the basic Sparse BLAS library up to fully-fledged simulation codes. Namely, LyNCs is targeting the Fast-Accurate Block Linear Krylov Solver (Fabulous), the Lattice QCD community solver library DDalphaAMG and at the lowest level the efficient sparse matrix support software LIBRSB. In the following table we summarise the major achievements of the project in software development.

Software	Before the project	After the project
Lyncs-API [40]	• Not existing • Many libraries available • Written in C/C++ • Manual cross-checks	• New community software • Interfaced to LQCD libraries and common framework • First Python package • Automatic cross-checks between implementations • High development standards
Fabulous [41]	• Version 1.0.1 • Algorithms IB-BGMRES-DR/ IB-GCR/BCG	• Version 1.1.2 • new algorithm IB-BGCRO-DR (including new stopping criteria and computational and numerical search space expansion) • Improved distribution & documentation (debian / ubuntu / MacOSX / spack / brew / guix)
DDalphaAMG [42]	• One right-hand side • SSE intrinsics • No deflation on coarse grid	• Multiple right-hand sides • Automatic vectorization and portability • Deflation on coarse grid • Interface to Fabulous • Block solvers
LIBRSB [43]	• Version 1.2.0-rc7 • Code/Fun. Coverage:37%/41% • SpMM via SpMV kernels • No C++-specific interface	• Versions 1.2.0.10 and 1.3 • Code/Fun. Coverage: 92%/99.9% • Native SpMM kernels in C++ • Header-only modern C++ interface • New tests, e.g. using GoogleTest • Bug-fixes • Now also on Spack, EasyBuild, guix
PyRSB [44]	• Pre-alpha implementation	• Stable release • Compatible with *scipy.sparse*
GNU Octave “sparsersb” [45]	• Version 1.0.6	• Version 1.0.9 • Bug-fixes • New tests • Improved documentation

Table 6: Overview of the major achievements of the project in software development

8.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

Overview: Within this section we summarize key features and improvements of the software packages developed within LyNcs in order to address the impact on the community. Within the project we focused on the implementation of the three “P”s of HPC, namely Performance, Portability and Productivity. Integration between software packages of the different project partners were established to separate concerns and extend flexibility and ensure performance and portability on novel hardware architectures of the software packages. Also modern tools have been used and novel interfaces have been developed to increase the productivity in our software. In the following we describe the main achievements in these three categories:

- **Performance**: Within the Lyncs-API we have been developing for the first time a fully-fledged Python framework for lattice QCD aiming at high performance interfacing to optimized libraries. Within LIBRSB we have developed new kernels using C++ templates and focusing on improving the performance of multi-vector operations. A similar focus took place in DDalphaAMG, where most of the library has been rewritten for enabling

multiple right-hand side calculations and the usage of block solvers linking to Fabulous. Within Fabulous we have developed a new algorithm for faster convergence and interfaced to LIBRSB for improved performance.

- **Portability**: Within the Lyncs-API we have been targeting both CPU and GPU architectures linking to various libraries and using existing Python tools optimized for this purpose. New interfaces have been added to LIBRSB and the code has been validated for various CPU architectures (e.g. such as AMD Epyc or ARM SVE). In particular, interfacing of LIBRSB within Maphys [46] allows Fabulous (the top of the stack) to profit from this efficient Sparse BLAS. Within DDalphaAMG we have used compiler flags and pragmas for achieving automatic vectorization of the computational kernels ensuring portability to various architecture and replacing the old SSE intrinsics.
- **Productivity**: Finally, improving productivity has been the main focus of the Lyncs-API and the choice of using Python for this purpose. We have been bringing under a common framework the main tools needed for lattice QCD calculations and we have employed modern approaches and tools in the developments like CI/CD using GitHub actions, software distribution on pip and online documentation on readthedocs. Also, we have developed interfaces to LIBRSB for high-level languages like PyRSB for Python and improved the GNU Octave interface. Also, linking DDalphaAMG to Fabulous allowed us to test many block-solver algorithms without needing to implement them. And as last, we have improved the deployment, distribution and documentation of Fabulous.

In the remainder of the section we will provide a technical description of the improvements including benchmarking results on petascale and Tier-0 systems. Namely we will focus on:

- **Lyncs-API**: Demonstration and scalability runs on JUWELS-Booster with QUDA;
- **Librsb**: Performance evaluation on BEAST and SuperMUC-NG at LRZ;
- **DDalphaAMG and Fabulous**: Scalability runs on SuperMUC-NG and JUWELS.

Lyncs-API:

The Lyncs-API offers Python interfaces to various lattice QCD libraries, which have been optimised for different architectures, and/or to implement different tools. Such libraries are for example DDalphaAMG (interfaced in lyncs-DDalphaAMG) that offers multigrid solvers on CPUs, and QUDA (interfaced in lyncs-QUDA) that offers operators and solvers, including multigrid, on NVIDIA GPUs and port to HIP and AMD GPUs is in progress. The main goal of the Lyncs-API is to bring as many libraries as possible under a common framework aiming for performance and portability, relying on a pool of libraries optimised for various architectures seeking high-coverage on the tools, and to productivity, choosing Python as programming language and providing a high-level user-friendly framework.

Therefore, when talking about performance and benchmarks, there are various aspects to analyse and hereafter we will comment on some of them:

- **Overheads introduced by the usage of Python**: Python is well known to be a significantly low-performing programming language (as compared to compiled ones). For this reason, in the Lyncs-API, it is never used in calculations but always as a driver. All kernels that process data are implemented in C/C++, compiled upon time and delivered either by the libraries or other Python modules (e.g., Numpy on CPUs and Cupy on GPUs). Interfaced libraries are compiled in a shared mode and linked with automatic bindings thanks to cppyy,

which offers high flexibility and high-performance Python-C++ bindings [47]. With such an approach we maintain all overheads introduced by Python several orders of magnitudes smaller than the calculations themselves, whose time can vary from a few milliseconds to many seconds.

- **Parallelism and parallel implementation**: As it is common in Lattice QCD, all libraries we consider offer data-parallelism via domain decomposition, distributing equally-sized subblocks of a 4-dimensional grid (the lattice) between processes. All of them implement parallelization via MPI with topological communicators, which are well supported in Python thanks to mpi4py. Therefore, data-parallelism is guaranteed by the libraries, while none of them implement task-parallelism nor modular computing. These are additional features that the Lyncs-API aims to offer. We do this in the module lyncs-mpi where we have implemented an interface that facilitates the usage of Dask with MPI. Dask is a Python package for easy and seamless task parallelism with a client-server approach, i.e., a client submits the tasks to be executed, a scheduler manages and optimizes their execution, and all other processes are idle workers waiting for instructions. In the lyncs-mpi we have divided workers into groups that internally communicate via MPI and execute library functions. In such a way we have combined data- and task-parallelism with a user-friendly approach (Note: this is still an experimental feature that is partially supported in the various interfaces).
- **Multigrid benchmark results on GPUs**: Finally, we discuss performance results measured via benchmark kernels on the Tier-0 system JUWELS Booster equipped with NVIDIA A100 GPUs. We will focus on the performance of the multigrid solver implemented in QUDA, which is nowadays one of the most critical components in our lattice QCD calculations where about 75% of the execution time is spent. For studying the scaling behaviour of the multigrid, we focus on the performance of its main components: the fine-grid Dirac operator (D) in double and single precision, as well as the intermediate-grid (Dc) and coarsest-grid (Dcc) Dirac operators in half precision. As depicted in the left-hand side of Figure 37, their parallel scalability strongly differs and is affected by the amount of parallelism that can be exposed on each grid whose size is reduced drastically at each level by the coarsening procedure. This can be improved, as in the right-hand panel of the figure, by solving for multiple right-hand sides together. The latter increases the scaling region but also the memory requirements of the solver, increasing the minimum number of nodes the solver can run on.

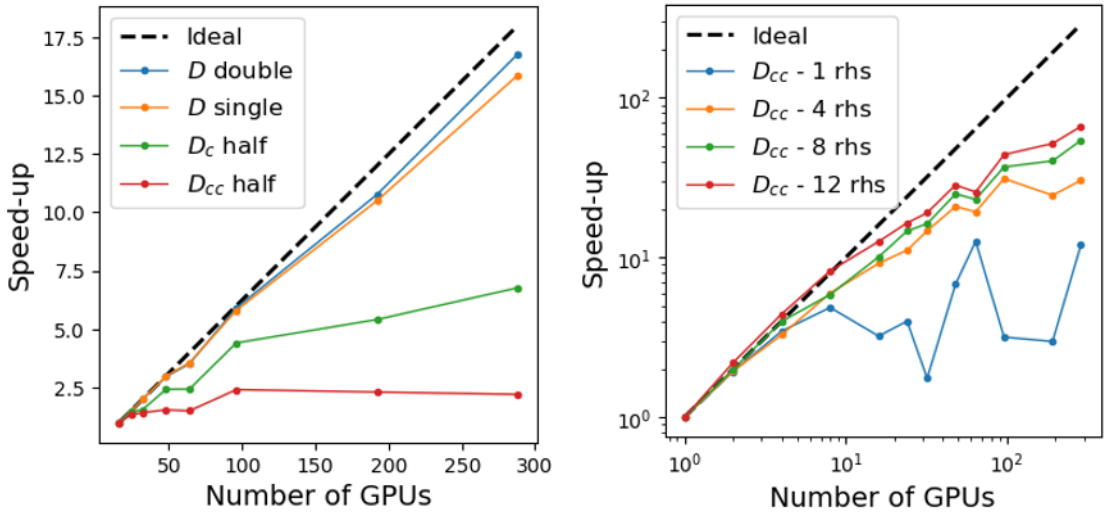


Figure 38: (Left) Strong scaling study of the QUDA Dirac operators on the fine (D), intermediate (Dc) and coarsest (Dcc) grids for a lattice of size 963 x 128. (Right) Strong scaling of the coarsest operator Dcc varying the number of right-hand sides (rhs) inverted at the same time.

LIBRSB and PyRSB:

The project has started with two branches: a stable one (namely LIBRSB-1.2) and a development one (namely LIBRSB-1.3). Fixes to old bugs making it into LIBRSB-1.3 have been applied in bugfix releases of LIBRSB-1.2. The most relevant improvement in LIBRSB-1.3 took place in the native Sparse Matrix-Matrix multiplication ("SpMM", of which Sparse Matrix-Vector multiplication, or SpMV, is a subcase); accessible via LIBRSB function rsb_spmm(). Modern C++ code has been written for this purpose, and is selected internally, at kernels' dispatch time. This code is templated with the number of right-hand sides as compile-time parameters; indeed, efficient SpMM is needed to make adoption of block Krylov methods convenient. The API of LIBRSB-1.3 is backward-compatible with LIBRSB-1.2's: this eases comparisons. Purpose of comparing branch 1.3 against 1.2 is to quantify performance improvement of the new SpMM kernels over old ones: 1.2 had SpMM emulated via repeated block-level SpMV. The performance experiment we report here involved SuperMUC-NG (smng) and four other experimental CPUs available on the BEAST (Bavarian Energy Architecture and Software Testbed) cluster: Marvell ThunderX2 (thx), Fujitsu A64FX (a64fx), Intel Cooper Lake (coop), AMD Rome (rome). The code has been compiled with 'icc -ipo -O3 -no-prec-div -fp-model fast=2 -xHost' on smng and rome; 'gcc -Ofast -march=native -mtune=native' on the others.

On each machine, eight batches have been run, all with 24 threads (more not needed, given the bandwidth-bound kernels) and OMP_NUM_THREADS=spread. Each batch has OMP_PLACES to sockets or cores; SpMM operands layout by-rows (C order) or by-columns (Fortran order); LIBRSB-1.2 or LIBRSB-1.3. Each batch ran with 44 matrices from different application fields; these are symmetric or asymmetric (general); the matrices are considered in each of the four BLAS numerical types, and 1, 2, 4 right-hand sides (denoted NRHS); for a total of 528 records per batch. Each pair of batches is elementally comparable with another one. We consider SpMM results after autotuning (rsb_tune_spmm()). Having different matrices is important: performance within a batch can differ by orders of magnitude, solely because of the sparsity pattern. Lacking a well-defined "average performance" concept in the context of vastly differently parametrized operations within one batch (say, SpMV of matrix bone010 as double vs SpMM-2 with matrix rajat31 as double

complex), we chose median speedups to get a sound overview. We performed a pretty extensive experiments campaign, but due to space constraints, omit a detailed results presentation and discussion, reporting only on the essential findings [48]:

- For NRHS>1, by-rows layout is now the recommended SpMM layout in LIBRSB-1.3 (in LIBRSB-1.2 it was the by-columns layout). With NRHS=2 it improves 0.6-16% over by-columns; with NRHS=4, 9-38%, depending on the architecture.
- With one exception (SpMV on thx), LIBRSB-1.3 to LIBRSB-1.2 median ratios indicate an improvement across all architectures. Specifically, the SpMV speedup median improvement amounts usually to a few percentage points. If using the recommended operands' layout, SpMM with NRHS=2 improves by 8-59%, SpMM with NRHS=4 by 25-73%, depending on the architecture. Notice that again, these are median values: individual ratios vary much more.
- LIBRSB-1.3's SpMM performed (normalized by NRHS) always better than corresponding SpMV (contrast with certain cases on LIBRSB-1.2 on a64fx, where we observed SpMM performing less than SpMV). This is important: convenience of block methods relies on this.
- Invoking autotuning brings relevant SpMM performance improvement (median on each architecture between 15.6% on coop and 38.0% on thx) in most cases. Largely recommended for say, > 100 expected SpMMs (a hundred operations' time being a reasonable estimate for the amortization cost).

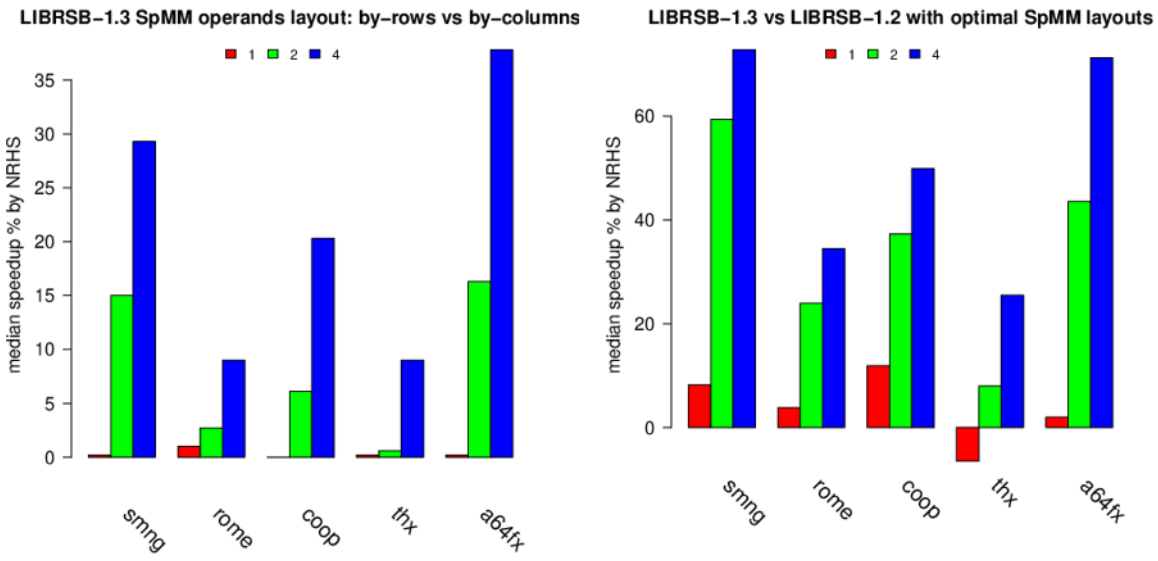


Figure 39: (Left) Median of speedup ratio between SpMM measurements with by-rows operands layout and by-columns, on different machines and for different right hand sides count. The by-rows layout is recommended in LIBRSB-1.3 because of its better locality in the lower level loops improves performance. Notice how for NRHS=1, that is SpMV, the layout is the same, and so the performance difference vanishes. (Right) Median of speedup when comparing LIBRSB-1.3 samples to LIBRSB-1.2 ones. Notice how with one exception, each machine/NRHS combination has been (overall) improved over LIBRSB-1.2.

DDalphaAMG and Fabulous: The community multigrid library DDalphaAMG has been extended to allow simultaneous solves of multiple right-hand sides (rhs) with an optional usage of advanced Block Krylov solver methods enabled in a linkage of Inria's solver library Fabulous. A major change within the low level kernels was implemented by changing the vector ordering from column to row ordering. This adds flexibility capabilities by having vectorization during compilation without explicitly using vector instructions. This guarantees portability without major performance loss to different CPU architectures, like ARM, Intel or AMD CPUs.

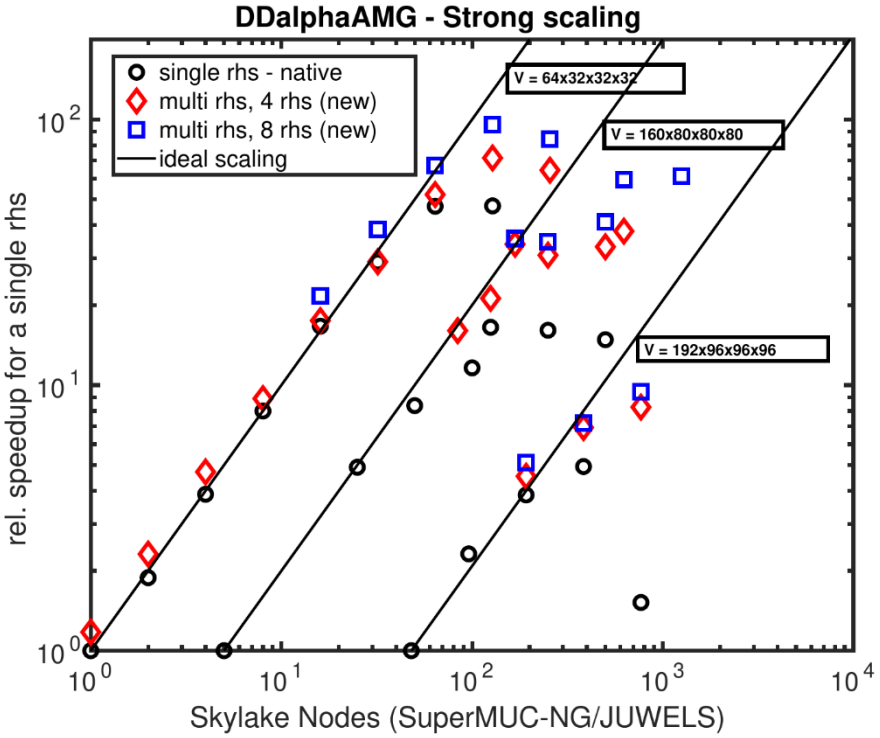


Figure 40: Strong scaling of DDalphaAMG with several lattice sizes and number of right hand sides.

The strong scaling tests of the new version of DDalphaAMG with multiple rhs were performed at various lattice sizes, namely V=64*32*32*32, V=160*80*80*80 and V=192*96*96*96, using twisted clover fermion where the runs with 4 and 8 rhs are compared to the original version with 1 rhs. For the smaller lattice size, a two level approach is used, while for the larger sizes a three level approach is used. In all cases the strong scaling region with multiple rhs is extended by a factor 2-5. Moreover, running with 8 rhs is outperforming the original version by up to a factor 1.5 in the scaling region utilizing 1 rhs.

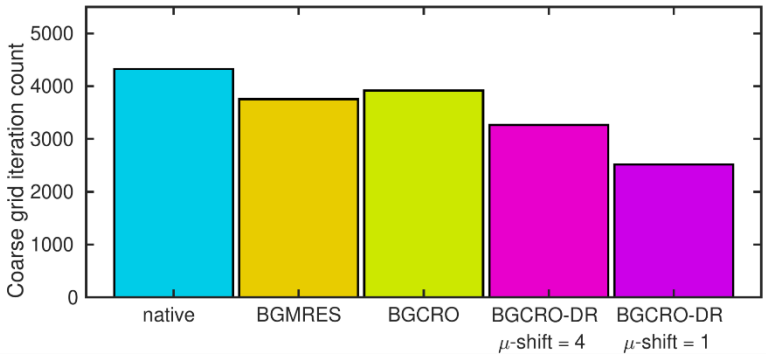


Figure 41: Comparison between the total iteration count of different Block-Krylov methods at the coarsest level of an 3 level multi-grid approach using a lattice with volume 64*32*32*32 employing twisted mass fermions.

In the second part of LyNcs, DDalphaAMG was linked to Fabulous, now enabling the use of the Fabulous library for block Krylov solver methods at all levels of the algebraic multigrid. We focused on the performance of Fabulous block Krylov solvers on the coarsest level. We found that although the required coarsest grid solve accuracy is low, namely already effective with residuals around 0.1, Block Krylov solvers are reducing the overall iteration count. It turns out that the (IB-)BGCRO-DR was the most effective algorithm, reducing the iteration count by 40% to the native approach. Note that this approach is working on the fly without additional preconditioning (mu-shift = 1) or exact deflation as done in QUDA. However, the linkage with Fabulous comes with additional overheads, namely with an additional 25% from reordering row to column and 75% communication overhead due to orthogonalization. This increases the overall time to solution by roughly 40% and needs to be addressed in the future to profit from numerically advanced Block Krylov solver methods.

8.3 Interactions with stakeholders, users, outreach and publications

Overview: For the dissemination and outreach of LyNcs activities we utilize all our channels to reach our community including researchers which expressed support via letters. This enabled us to embed software developed under LyNcs within the community and guarantee future support. For example, the newly developed software package LyNcs-API is selected by the Extended Twisted Mass collaboration for their future simulation driver. Moreover, within the project we could focus on enabling kernels, needed by the European community, such as twisted mass fermion and non-degenerated twisted mass fermions with the community solver libraries DDalphaAMG and QUDA.

Over the duration of the project linkage between different partner software was made, now enabling new options to the package Maphys via LIBRSB and DDalphaAMG via fabulous. The dissemination of Lyncs activities and results started with a contribution to the PRACE Inter-WP workshop in October 2020, where Lyncs was one of the two selected WP8 projects to represent WP8. Most of the other dissemination activities focused on the specific software efforts and they are presented in the following.

Lyncs-API: The Lyncs-API implements middle- and high-level tools distributed via various Python modules using a modular structure (see https://github.com/Lyncs-API). Being in the middle of the development chain, the interaction with other developers/users goes in two opposite

directions: on one side, towards improving the low-level libraries and Python packages used by the API, and, on the other side, by interacting with users from various communities that want to employ the API in their applications. In the first direction, during the duration of this project, we have been active in adding features and identifying/solving issues to various packages such as the lattice QCD libraries Quda, DDalphaAMG and tmLQCD, as well as to the Python packages cppyy, h5py, sh and pylint. In the other direction, instead, we have been working towards i) facilitating the usage of Python in HPC, developing generic purposes tools like lyncs-io that offers parallel IO for various file-formats, ii) offering for the first time Python interfaces to various libraries, that are welcomed by the developers of the libraries themselves as well as users of the libraries that seek for a Python experience, and iii) developing a high-level framework that aims to be portable and easy-to-use towards exascale supercomputing. All the developed software is open source and welcomes contributions from users. It also implements high-quality features of community software like documentation, issues tracker and CI/CD (short for Continuous Integration / Continuous Delivery) via GitHub actions.

Over the course of this project, the Lyncs-API has been object of dissemination at:

- Extended Twisted Mass Collaboration Meeting within February and November 2021;
- Seminar at the Cyprus NCC with a presentation (by J. Finkenrath and S. Bacchio);
- Presentation at the Lattice Conference 2021 with a presentation (by S. Bacchio);
- Publication in the Proceedings of the lattice conference (due by end of October 2021).

LIBRSB: Over the course of this project, LIBRSB has been object of dissemination at:

- PRACE Days (March, 2021): Poster by Martone + Video by Bacchio;
- Slideshow with feedback for the the BEAST (Bavarian Energy Architecture and Software Testbed) user community (March, 2021), by Martone;
- ISC'21 (June, 2021): Poster by Martone, Bacchio, Finkenrath, Giraud and Simonin;
- Scipy'21 (July, 2021): Oral presentation + Proceedings paper by Martone and Bacchio;
- Invited seminar for the Max Planck Institute for Dynamics of Complex Technical Systems (July, 2021) by Martone.

Further dissemination activities are ongoing. A renewed communication round is ongoing with past, present, and prospective LIBRSB users and collaborators. Beside performance, points of interest pertain to new kernels (e.g. the ATA kernel), specific matrices, new code generation techniques, bindings to other languages, and most novel hardware.

Fabulous: During the project dissemination activities of Fabulous are given as follows:

- A new release has been made available and publicized to our close academic and industrial collaborators such as Airbus, BSC or Cerfacs to name a few that have expressed the wish to have such linear solvers.
- The revised version of a paper on the mathematical part of the new Krylov solver IB-BGCRO-DR has been submitted to SIAM with acknowledgements to the Lyncs project that enables us to further validate the implementation of IB-BGCRO-DR in Fabulous.

DDalphaAMG: During the project results of the multiple right hand side extension of DDalphaAMG were disseminated on various events, such as:

- At the major online conference APAC lattice conference 2020 (by S. Yamamoto);
- Invited Riken Kobe seminar contribution, Sept. 2020 (by S. Yamamoto);
- Contributed talk at Lattice Conference 2021 (by S. Yamamoto);

- Publication in the Proceeding of lattice conference (due to end of October 2021).

8.4 Overall assessment of achievements and future developments

Within the project LyNcs HPC experts from different European organizations joined forces to address heterogeneous challenges arising in future European supercomputing. By utilising the wide range of expertise various tasks could be addressed on all different levels of sparse linear solver software packages needed in applications of High Energy Physics, such as lattice QCD, Computational Electrodynamics and Computational Chemistry. This enabled support to prepare European software for the upcoming European Supercomputers thanks to new synergies among different teams across Europe. This was possible thanks to the provided funding and infrastructure by PRACE, which not only connected computational scientists of different fields but also was timely and crucial to perform the necessary first steps to enable the community to utilize future computing resources in Europe. Needless to say software efforts such as PRACE-6IP WP8 are playing a key role to provide support for the diverse HPC community which are not represented by European Centres of Excellence, such as the High Energy Physics community.

In detail, within LyNcs we achieved readiness of software solutions for the next-generation of European supercomputing. Among these achievements is LIBRSB-1.3's very high coverage statistics, going in pair with newly established unit tests and a rich CI/CD pipeline. Test code amounts now to circa 10kLOC (10% of total lines). Nearly every bugfix has originated a specific test. Further tests have originated during refactoring or documentation consolidation and general fortification. Many of the new tests use the Google Test framework for unit testing. This LIBRSB fortification activity was mostly concentrated in the first period, and allowed stable ground for development of new SpMM kernels and new features later on. Once obtained an improved performance in SpMM, the last project phase went into performance evaluations, dissemination, and community involvement (especially thanks to the different access layers to LIBRSB). With Lyncs-API a new portable, user-friendly python interface was developed, which comes with task-parallelism and linkage to various lattice QCD solver packages such as QUDA and DDalphaAMG and legacy lattice QCD simulation packages, such as tmLQCD. With the linkage of Fabulous to DDalphaAMG new combinations of algorithms could be studied and iteration counts could be improved.

A major challenge in LyNcs was the design and development of the new API. Originally proposed as a linear solver API on the intermediate level between simulation codes and solver library, the API became the driver to enable task parallelism and modular supercomputing capabilities. One reason for this adjustment, is that the lower level kernels are latency bound, making task parallelism on lower level ineffective. Finding a trade-off between numerical and parallel efficiency remains a challenge. In the Fabulous package, advanced numerical search space expansion policies have been introduced to attempt reducing the number of iterations. They are numerically effective under the constraints of building a basis with a ``good enough" orthonormality of the residual space. In the current implementation, this is ensured by a modified Gram-Schmidt procedure which unfortunately is revealed to lead to some latency bottleneck, difficult to overcome without significant changes in the numerical method.

Future plans for each software package are the following:

- **Lyncs-API**: The main focus within the project has been the design and creation of the first Python API for lattice QCD applications. An important step has been interfacing to lattice QCD libraries as well as the creation of supporting tools like parallel IO. Beside continuing

supporting and improving the software developed, in the future we plan to develop mapping software that will allow for a seamless interchange of functions implemented by different libraries. Indeed, each library may use a different data layout and ordering. Furthermore, we will complete the functionalities needed for performing HMC simulations using the Lyncs-API and we plan experiments for enabling machine learning in lattice QCD.

- **LIBRSB**: Refining SpMM strategies for large NRHS counts (e.g. >8) can be an object of further investigation. Performance gains are expected by introducing new block-level formats that may fit best certain specific matrices (e.g. LQCD matrices may benefit from BCSR). Adding new kernels for existing formats, but with specific structural characteristics (e.g. low or high nonzeroes density) may improve overall performance. The autotuning procedure can be optimized and made faster; its strategy can be improved and lead to faster SpMM. Matrix assembly routines can be improved in speed and memory usage. Storing autotuned blocking information in a reusable form may speed-up matrix assembly and spare repeated autotuning. Supporting mixed arithmetics has not been addressed in this project, but has potential. GPU integration strategies exploiting the hybrid format nature of RSB (e.g. coarse-grained parallelism) may be explored.

- **Fabulous**: in the future we plan to study minimum residual norm block algorithms based on Householder reflections, that have a higher floating-point arithmetic complexity but possibly enabling the introduction of blocking algorithms in the orthogonalization phase, possibly reducing the latency effect. Having such algorithms in Fabulous will provide higher flexibility and adaptability to enhance the capability to best explore the trade-off between numerical and computational efficiency.

- **DDalphaAMG**: possible future steps will be to further optimize the multiple rhs version of DDalphaAMG towards large scale applications on Fugaku and future European exascale machines with European chips. The linkage to Fabulous and with it the usage of advance Block Krylov solver methods such as IB-BGCRO-DR solver can be further intensified, namely by minimizing communication overheads in order to profit from the numerical advanced methods. Moreover, support for additional variants of Wilson Dirac operators including support of different boundary conditions can be added.

9 QuantEx: Efficient Quantum Circuit Simulation on Exascale Systems

9.1 Introduction and summary

The QuantEx project aims to provide a quantum circuit simulation framework for Exascale systems which is scalable and extensible. The ability to simulate quantum circuits is essential for the design and development of quantum computing hardware and algorithms. With the emergence of Noisy Intermediate Scale Quantum (NISQ) devices, it has become intractable to simulate devices of this size using the traditional method of direct evolution of a quantum wave-function, even on the largest supercomputers [49]. QuantEx uses a representation of quantum states known as tensor networks which enable the output probability amplitudes to be calculated by contracting a network of tensors [50] [51]. This approach has achieved state of the art performance when simulating Random Quantum Circuits (RQC) as part of the recent quantum advantage experiments [49]. Despite these impressive results, it is not competitive for simulating all circuit types. In particular, for very deep/highly entangled circuits, the tensor network representation can require the same amount of memory as full wave-function methods. In these cases, full wave-function approaches with simpler memory management and fewer overheads are generally more efficient. For circuits targeting NISQ devices, with moderate depth/entanglement and where approximate results suffice, tensor network approaches can offer significant advantages. Figure 41 below illustrates this area of application:

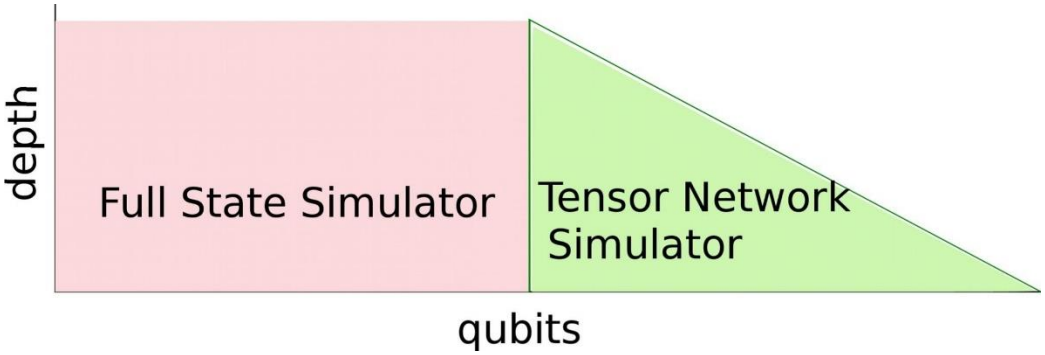


Figure 42: Expected area of applicability of simulation methods

The framework developed by the QuantEx project consists of several special purpose software packages aiming to address different issues that arise in tensor network simulations. These are QXTools, QXTns, QXGraphDecompositions and QXContexts, each of which we describe below and are available on github under the JuliaQX organisation [52]. The packages are also registered as Julia packages making them easily accessible. Julia [53] is used as the primary language, because of its flexible type system, the ability to wrap components in other languages while also providing native performance and native support for GPGPU programming. A domain specific language (DSL) is used to represent the simulation as a set of tensor network operations. This separates the high level index accounting and contraction planning from the low level implementation of the tensor network operations and makes it easier to support new hardware and network architectures.

QXTools is the main QuantEx package for orchestrating a tensor network simulation of a quantum circuit. It can be used to create a tensor network for a quantum circuit, identify an efficient contraction scheme for the network and generate simulations files, including tensor data files and DSL files, that describe how the simulation should be executed on a cluster. It provides a quantum circuit simulation workflow which consists of the following steps:

1. Circuits are built and represented as QXZoo circuits.
2. The QXZoo circuit is converted to a QXTns tensor network.
3. This network is converted to a graph data structure provided by QXGraphDecompositions and a suitable tree decomposition and set of edges to slice are identified.
4. Using the tree decomposition and set of edges to slice a DSL representation of the computation is generated. This is then used as input to QXContexts to perform the computation using the context and settings that make the best use of the available resources.

QXZoo Provides data structure and functions for representing and generating quantum circuits.

QXTns is a Julia package with data structures and utilities for manipulating tensor networks. As well as a generic tensor network data structure, it also contains specific data structures for handling tensor networks derived from quantum circuits. It includes the ability to automatically identify and track hyper-indices of tensors which can lead to significant performance improvements.

QXGraphDecompositions is a package for analysing and manipulating graph structures describing tensor networks. It provides data structures and functions for analysing and manipulating graph representations of tensor networks. In particular, it provides functions for finding efficient tree decompositions and for identifying sets of indices which when sliced can reduce the treewidth of the selected tree decomposition. This makes it possible to distribute computations across multiple processes/nodes.

QXContexts is designed to parse the simulation files created by QXTools and perform the tensor contractions that constitute the circuit simulation making use of distributed compute resources via MPI as well as hardware accelerators. It also provides implementations of sampling algorithms which can be used to generate random bit-strings which are distributed according to the output state of the simulated quantum circuit.

9.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

The following strategy has been developed for assessing the performance of JuliaQX on multiple HPC platforms:

- The software has been tested on a number of platforms, including both production machines available at the sites of the project partners and test systems;
- On multiple platforms, performance diagnostics (in particular, those related to the memory and cache hierarchy) have been collected by using the command-line, performance tool suite LIKWID [54];
- A novel trend in the management of HPC workloads is the usage of containers to easily deploy a whole software environment on any given architecture. Motivated by the increasing interest of the community in this solution, we benchmarked JuliaQX with and without use of containers on multiple architectures;

- Finally, GPU support has been added and tested.

To evaluate the performance of the JuliaQX software, we use as a test case the problem of computing probability amplitudes for a list of possible bitstring outputs of a quantum circuit. The quantum circuits we use in these test cases are instances of random quantum circuits (RQC) defined in [55] and used in Google's quantum advantage experiments [56]. These circuits consist of a 2 dimensional array of qubits with several layers of quantum gates acting on all qubits. For our initial scaling calculations, simulation files were generated for a RQC with a 5 by 5 grid of qubits and 24 layers of gates.

Initial scaling results were computed on ICHEC's Kay cluster of 336 nodes where each node has 2x 20-core 2.4 GHz Intel Xeon Gold 6148 (Skylake) processors, 192 GB of RAM, a 400 GB local SSD for scratch space and a 100Gbit OmniPath network adaptor. In Figure 42 below, we take the case of computing 2048 amplitudes for the 5x5x24 RQC, with a single sliced bond, on 4 nodes with an increasing number of processes.

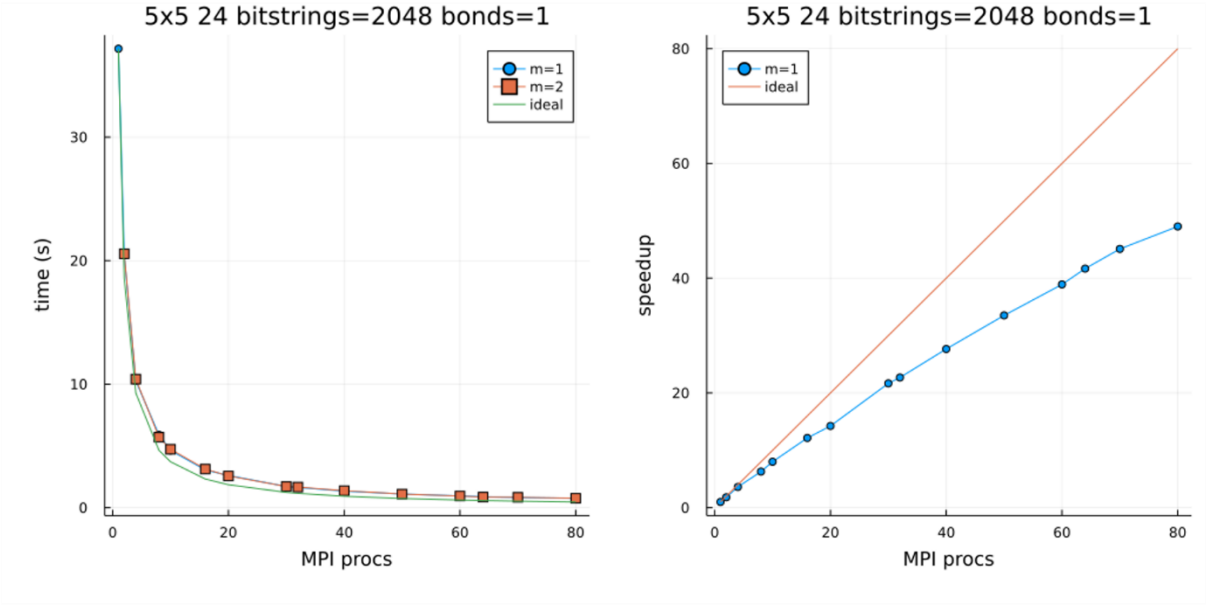


Figure 43: Computation of 2048 amplitudes for the 5x5x24 RQC, with a single sliced bond, on 4 nodes with an increasing number of processes

Additional tests of other HPC architectures have been performed on the BEAST system at LRZ. The Bavarian Energy Architecture and Software Testbed (BEAST) is a collection of systems for the research and evaluation of new hardware technologies. Currently BEAST consists of three different CPU architectures: AMD X86, and Arm Fujitsu A64fx. An additional system segment is equipped with Arm ThunderX2, but the LIKWID tool was not fully functional on this architecture at the time of testing and therefore we leave it for future investigation.

The AMD systems consists of two node Rome GPU 2U servers, with two AMD EPYC 7742 with 64 cores along with 512GB of DDR4-3200, two 1.9 Terabyte SSD and two AMD Radeon MI-50 GPUs with 32 Gigabytes of high bandwidth memory (HBM). The interconnections between the nodes are Mellanox InfiniBand: HDR 200Gb/s.

Finally, The Fujitsu A64fx system is an eight node HPE system consisting of Arm Fujitsu A64fx CPUs with 64 cores and two 512-bit vector units and 32 gigabytes HBM2 memory that is connected with a Mellanox InfiniBand EDR interconnect.

Table 7 below lists the key features of the different architectures evaluated on BEAST. Extensive benchmarking and optimisation work is still ongoing but these tests serve to validate that the QuantEx framework can run with these architectures. A point of reference (third column) is provided by the Intel Xeon Scalable Processors ("Skylake") of SuperMUC-NG. The table also presents some performance diagnostics collected using LIKWID, namely run time, arithmetic throughput, and measured memory bandwidth of our framework. The test cases used for these tests were computations of amplitudes from 12 and 24 qubit GHZ circuits. We note that the increased run-time in going from a 12 qubit circuit to a 24 qubit circuit reflects a non-trivial increase in both the workload and memory requirements of the computation.

	ARM A64FX	**Intel SKL (SuperMUC-NG)**	**AMD-ROME EPYC 7742**
Run-time (s) - 12 Qubits	0.14	0.061	0.030
Run-time (s) - 24 Qubits	245.4	124.92	59.06
FLOPS DP (MFLOP/s) - 12 Qubits	93.74	196.26	Not available on this architecture
FLOPS DP (MFLOP/s) - 24 Qubits	823.17	511.7	Not available on this architecture
Memory Bandwidth (Mbytes/s) - 12 Qubits	389.44	546.99	496.4
Memory Bandwidth (Mbytes/s) - 24 Qubits	1665.42	4310.1	3110.72
Base-Frequency (MHz)	425	2300	2250
SIMD (bit)	2048	512	256
Cores /node	48	48	64

Table 7: Key features of the different architectures evaluated on BEAST

Beside testing the software on the HPC architectures listed above, a complementary part of our benchmarking efforts has been to deploy it via Charliecloud HPC containers on the BEAST segments and compare the performance against the "bare metal" deployment. For a detailed description of our procedures and results we refer the reader to a forthcoming publication with title "Deploying Containerized QuantEX Quantum Simulation Software on HPC Systems", accepted for the 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC at Supercomputing 2021 (SC21), Sunday, November 14th, 2021.

The use of HPC specific containers significantly reduces the deployment effort of the software on different HPC systems. It is relatively simple to build, configure the quantum gate simulation software developed in Julia on multiple different HPC systems with different CPU architectures and instruction sets. In addition, we were also able to show that the runtime performance difference

for QXContexts between containerized and bare metal versions is negligible using the LIKWID profiling software.

JuliaQX's NVIDIA GPU support was tested using a 16GB NVIDIA Volta V100 GPU on Cineca's Marconi100 system. Here, we measured the time to compute a single amplitude on both a GPU and on one of Marconi100's 16-core IBM POWER9 processors. The time was measured for several, progressively difficult, quantum circuits and the results are displayed in Table 8 below. Each row contains the results of a different circuit with the first column giving the name of the circuit. The second column contains the treewidth of the simulation which is a proxy for how difficult the simulation is as the complexity of the simulation is exponential in the treewidth. The memory column shows the maximum memory footprint of the process over the course of the calculation. The measured times show a clear benefit to using the GPU for larger circuits.

Circuit	Treewidth	Memory	CPU time	GPU time
GHZ	2	160 B	1.29 ms	2.538 ms
RQC 4x4x24	11	18 KB	21.572 ms	53.372 ms
RQC 6x6x24	19	4.2 MB	229.1 ms	112.1 ms
RQC 5x5x32	23	67.1 MB	342.7 ms	167.9 ms
RQC 7x7x24	24	128.5 MB	1.296 s	292.2 ms
RQC 6x6x32	27	512 MB	9.141 s	384.9 ms
RQC 7x7x32	32	16.4 GB	16 s*	1.25 s*

Table 8: Time to compute various quantum circuits on Marconi100 system

Note, for the largest circuit (marked with asterisks in the Table above), the memory footprint exceeded the memory capacity of the Volta V100. In this case, slicing was used to split the calculation into two subtasks and reduce the memory requirements of the calculation. The measured time is the time to complete one of these subtasks.

To port our framework to the Marconi100 heterogeneous system (CPUs and GPUs) the Julia package CUDA.jl was used to add NVIDIA GPU support to QXContexts. This also enabled us to use NVIDIA profiling tools to analyse our software's performance. NVIDIA Nsight profiler tool has been used to measure the performance of our code. This tool facilitates performance analysis and guided optimization, provides memory transfer, I/O as well as load host to device transfer. After configuring and adapting the NVIDIA NVTX instrumentation on the Julia framework the profiling was more detailed and it was very useful for observing the behaviour of our algorithms, and to make informed decisions and solve problems as they appeared. We can see that all host-to-device transfers happen for output commands; moreover, the profiling clearly shows which code parts take the longest runtime, e.g: "ncon" made up of a permute and the cuBLAS libraries: cgemm, gemv or broadcast kernel depending on the case, see Figure 43 below. The optimised version of our framework has been used and the time for each quantum circuit was measured and the results are displayed in the table above.

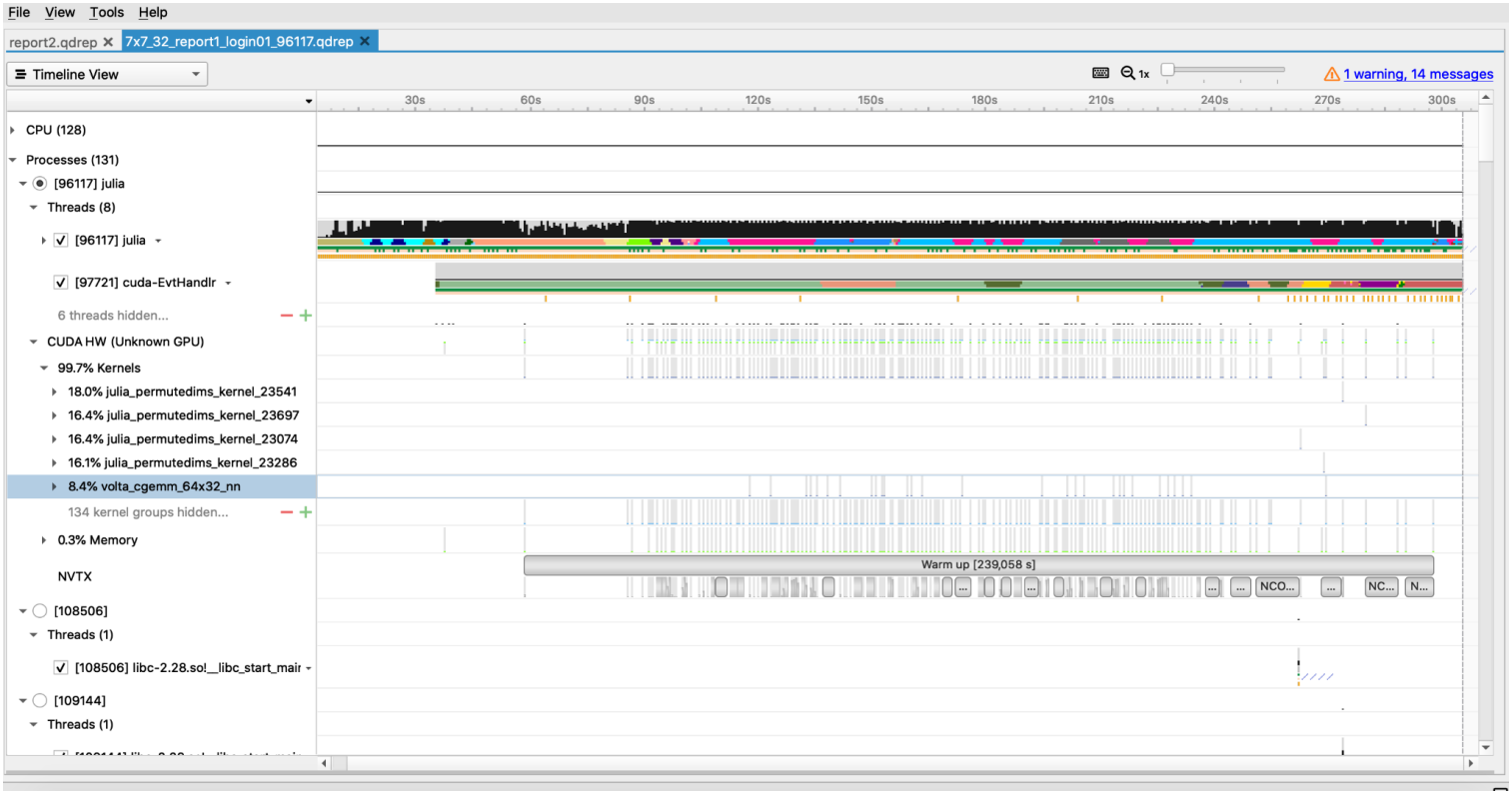


Figure 44: NVIDIA Nsight profiler analysis report of code on heterogeneous Marconi100 System, with an overall summary highlighting the bottlenecks.

9.3 Interactions with stakeholders, users, outreach and publications

At various stages throughout the project there has been engagement with project stakeholders and potential users. This has been in the form of online meetings and a hands-on workshop for external stakeholders where design plans and our initial prototype PicoQuant were presented. A set of introductory Jupyter notebooks were prepared with examples of quantum circuit simulations and explanations of background theory. These allowed the stakeholders to interact with the prototype software and gain an understanding of how it works before giving their feedback and recommendations.

A number of dissemination efforts were also undertaken to promote the QuantEx as a viable quantum circuit simulation tool. To coincide with the full public software release, a number of outreach activities were organised to promote the project and reach out to potential users. These activities include stories on the ICHEC, LRZ and PRACE websites and social media channels as well as a series of workshops to introduce interested users to the tools that have been developed.

In addition to these efforts, virtual posters were presented at JuliaCon 2021 with the titles “Distributed Quantum Circuit Simulation” and “Introducing QXGraphDecompositions”. The first presented the Julia package QXTools.jl as the main QuantEx Julia package for setting up and running quantum circuit simulations. The second presented the Julia package QXGraphDecompositions.jl as a solution to finding efficient contraction orders for tensor networks. The posters were well received and published on the Julia Programming Language YouTube channel [57] [58]. A virtual poster was also presented at the ISC High Performance 2020

international conference illustrating the tensor network techniques used by QuantEx to simulate quantum circuits and the parallel decomposition methods used to break a simulation into smaller tasks.

A containerized version of the software workflow was demonstrated in a cloud-based HPC cluster at the Supercomputing 2020 (SC'20) tutorial "Practical OpenHPC: Cluster Management, HPC Applications, Containers and Cloud" [59]. At FOSDEM 2021 HPC container presentation, "Deploying Containerized Applications on Secure Large Scale HPC Production Systems" [60]. QuantEx software workflow was presented as an important use case for containerized workflows on traditional HPC systems. In addition, the paper "Deploying Containerized QuantEX Quantum Simulation Software on HPC Systems." has been accepted for the 3rd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC at Supercomputing 2021 (SC21), Sunday, November 14th, 2021 (see also Section 9.2). Furthermore, a paper titled "Tensor Network Circuit Simulation at Exascale." has also been accepted for the Second International Workshop on Quantum Computing Software at Supercomputing 2021 (SC21).

As well as engaging with potential users, efforts are ongoing to identify suitable opportunities to integrate the developed tools into commonly used quantum circuit simulation frameworks. One particular direction the QuantEx team is exploring is the possibility of integrating QuantEx as a backend for the popular Yao.jl [61] framework. The Julia package YaoQX.jl [62] was developed with the hope of enabling Yao.jl users to take advantage of distributed systems and pre-Exascale and Exascale HPC clusters to simulate quantum circuits.

9.4 Overall assessment of achievements and future developments

The QuantEx project was successful in developing a modular flexible quantum simulator using modern software development techniques (CI, PR driven development, automatically generated documentation) and leveraging the novel programming language for scientific computing JuliaLang. The simulator is open source and available at [52]. The software includes implementations of state of the art techniques such as using hypergraph representations of tensor networks [63] to reduce both time and space resources required by a simulation and a novel tree trimming method [64] for automatic efficient tensor network slicing to decompose a simulation into independent tasks that can be processed in parallel. The software was successfully tested on Intel and AMD CPUs and on NVIDIA GPUs and future work includes expanding the supported hardware to include AMD GPUs and Intel GPUs.

During the development process, we defined a domain specific language (DSL) to represent primitive tensor network operations. The DSL can be used to describe a quantum circuit simulation as a sequence of tensor contractions and to instruct how a simulation is to be executed on a cluster. In QXTools.jl, the DSL is represented by a tree structure indicating data dependence between tensor operations, allowing for various optimization passes to be made before writing the described simulation scheme to a simulation file. For instance, the tree representation of the DSL can be used to identify the global connectivity of hyperedges in the tensor network leading to further reductions in the space complexity of the simulation. Continuing and expanding this work is of interest to many groups working on intermediate representations for tensor network calculations and could be an impactful future direction to follow.

Other possible future directions focus on optimizing simulations via improved network contraction planning capabilities and better bitstring sampling methods. One method for improving a

contraction plan for a tensor network is that of local optimization [65]. This involves replacing subtrees of a contraction tree with optimal alternatives found using an exhaustive search and potentially offers significant improvements in computational cost of a simulation. For an implementation of local optimization to be integrated with QXTools an exhaustive search method for finding optimal contraction trees would be required which could account for features like hyperedges that are already implemented. Developing such a technique would be of value to other projects that involve searching for optimal contraction plans for hypergraphs. An approach to optimising bitstring sampling using memoization was proposed recently [66] and offers large reductions in the time complexity of a simulation. The method consists of designing a contraction plan for a tensor network with a natural checkpoint which a simulation can return to between samples to avoid recontracting a large portion of the network. A rewarding direction of future work would be to integrate this technique with QXContexts to greatly improve efficiency and possibly generalise the method to identify optimal checkpoints in arbitrary contraction plans, rather than only carefully designed plans, broadening the contraction planning algorithms that can be used with this technique.

10 GHEX: Generic Halo-Exchange for Exascale

10.1 Introduction and summary

Halo exchange is a fundamental component of parallel Finite Difference, Finite Volume, and Finite Element solvers of Partial Differential Equations (PDE). Grid-based PDE solvers are amongst the most widely used numerical methods in scientific HPC, e.g., in atmospheric sciences, astrophysics, structural and mechanical engineering, automotive industries, and geology. In these numerical methods the spatial domain is discretized using a grid, and split into compact subdomains that are assigned to individual Processing Elements (PE). To satisfy the PDE across the sub-domain boundaries, the PEs have to exchange the boundary values with their spatial neighbours. This operation is known as the halo-exchange, or the ghost-cell exchange.

Modellers from different scientific domains use various types of grids: 2D or 3D, regular Cartesian grids, block-structured grids, or unstructured meshes. The implementation of the halo exchange depends on the grid type, and on the underlying data structures. One can use some of the existing libraries (such as PETSc), which perform the halo exchange for certain grid types, but at the same time impose the programming language and the data structures that have to be used by the application. Hence, adapting existing applications may require a large amount of work. Instead, the developers often choose to implement the halo exchange themselves using MPI. This approach has two major drawbacks: 1) the generic halo exchange code is re-implemented and re-optimized multiple times, often by non-experts 2) the performance and its portability is limited to what MPI can provide, which leaves out many optimizations available on today's HPC hardware.

We have developed GHEX - a generic halo exchange library for modern HPC architectures. GHEX features a unified C++ halo exchange API suitable for arbitrary grid types, periodicity, data types, and problem dimensions. It is implemented as an asynchronous communication primitive, hence it allows for overlapping computations and communication. It does not use any global synchronization. The arguments of the API are functions with defined interfaces, which makes it oblivious to the specifics (e.g., the data layout) of the application. GHEX does not impose a specific data structure on the user, but rather adapts to the user’s data structures. Fortran and Python bindings make sure that the library can be used by a wide audience. Figure 44 below shows a general overview of the GHEX ecosystem, with library concepts on the left-hand side (communication pattern and iteration spaces), and user implemented ones on the right-hand side (for describing domain decomposition, halos and data). The communication object is the place where the halo exchange is actually implemented, making use of lower level APIs for the transport layer and the communication strategies.

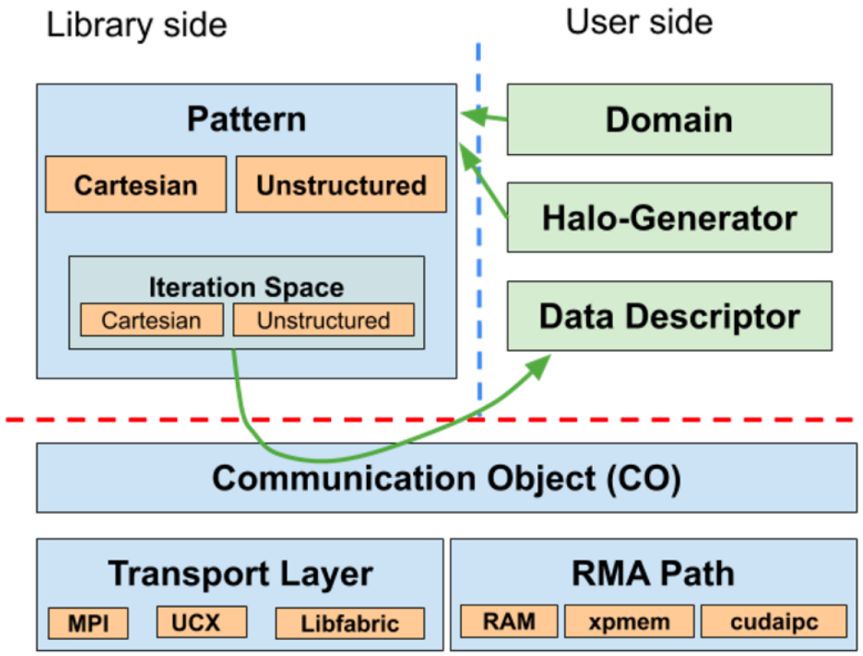


Figure 45: Overview of GHEX

During the development and optimization of GHEX it became clear that to tackle the complexity of modern HPC architectures, several additional software components are needed. Within the GHEX project we have developed a number of general-purpose tools that can benefit a wide range of HPC codes:

- HWMALLOC [67] - a multi-threaded, NUMA-aware memory allocator;
- HWCART [68] - a hardware-aware Cartesian MPI communicator;
- OOMPH [69] - a transport API, which uses fabric-specific transport backends (e.g., UCX, libfabric) and can be used as a partial alternative to MPI;
- GHEX [70] - the halo exchange API, together with optimized implementations for a number of common grid types;
- GHEXBENCH [71] - a sophisticated halo exchange benchmark, which compares a number of GHEX implementations to the standard MPI approach.

Each of the above codes is a separate git repository under the ghex-org github organization.

GHEX has been integrated into two astrophysics codes developed and maintained at the University of Oslo: BIFROST and DISPATCH. BIFROST is a classical finite difference code, in which the MPI ranks are single-threaded, and each rank handles a sub-domain of the global simulation space. DISPATCH is a heavily multithreaded task-based modeling framework, in which the work is assigned to the threads in a dynamic fashion. GHEX has been integrated in unstructured applications as well. Generic unstructured patterns deduction is part of the GHEX core, and explicit user-facing functions are already implemented for common use cases such as unstructured graphs in CSR format and ECMWF's Atlas library.

10.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

For the benchmarking purposes we used the following HPC systems:

- Piz Daint - an Intel Xeon based Cray X40/X50 system with Gemini interconnect, accelerated (NVIDIA Tesla P100) and non-accelerated partitions, located in Switzerland
- Betzy - an AMD Epyc 2 system (BullSequana XH2000) with Mellanox IB HDR100 interconnect, located in Norway

The CPU architecture of Betzy is similar to that of LUMI, a pre-exascale system assembled in Finland, which will host compute nodes equipped with AMD Epyc CPUs and MI200 GPUs, and interconnected with the Slingshot network. Although Betzy does not have GPUs at the moment, we have tested GHEX on other accelerated test machines based on AMD Vega10 and Vega20 architectures.

Efficient memory access is a major performance challenge of modern architectures, and the main optimization target for the halo exchange. Consider Betzy, which consists of dual-socket compute nodes. One Epyc CPU has 4 NUMA nodes, each NUMA node consists of 4 Core Complexes (CCX) of 4 cores, for a total of 64 cores. Cores within the same CCX share an L3 cache. CCX's and sockets are interconnected with the Infinity Fabric. On this NUMAcc system, accessing the memory located in the different memory domains has a different cost. During the halo exchange the data is copied between different memory domains, both in- and off-node. The total communication cost (and thus the performance) highly depends on how the grid sub-domains are assigned to compute nodes and cores. In addition, removing any unnecessary copying of data, as well as avoiding re-allocation of memory also improves the execution time.

Description of developed software components

HWCART is a hardware-aware Cartesian communicator that gives the user the possibility to define an optimal rank-to-memory domain mapping by hierarchically arranging the lower-level domains into grids inside the higher-level domains. For example, on an AMD Epyc 2 system with two sockets per compute node, at the lowest level of the hierarchy are the cores. Groups of 4 cores belong to a CCX (L3 cache domain), 4 CCX modules make a NUMA node, 4 NUMA nodes comprise a socket, finally there are 2 sockets on each compute node. Here, the cores inside a CCX can be arranged into a [4, 1, 1] rank grid. 4 CCX units can then be arranged into a [1, 2, 2] grid, and so on. With this approach the user has a direct control over each rank's neighborhood, which results in the minimization of the off-node communication, and maximization of the performance of in-node, shared-memory data exchange.

HWMALLOC is a multi-threaded, NUMA-aware memory allocator. Similar to other existing solutions, for each memory domain (NUMA, accelerator) we keep several pools with memory blocks of different predefined sizes. What distinguishes HWMALLOC is that it provides fabric-specific hooks (currently MPI, UCX, libfabric) for memory initialization and registration. Hence, the newly allocated memory is immediately ready to be used in RMA operations. Applications that rely on frequent allocation and freeing of large memory buffers (e.g., DISPATCH) are able to do so without any penalty connected with transport-specific memory registration.

OOMPH is a multi-threaded transport library that can be used as an alternative to MPI tagged messages. OOMPH uses low-level transport backends (UCX and libfabric), or a generic MPI backend. In the multi-threaded scenarios, each rank has one shared recv worker, and one private send worker per thread. This implementation results in a significantly better multi-threaded

performance than when using standard multi-threaded MPI libraries. Communication progress and completion is managed either by MPI-like requests, or by callback functions, which get called whenever a request is completed. OOMPH also supports a "send and forget" model, in which the send buffers are automatically freed upon completion. These features significantly simplify the communication stage in task-based codes, like DISPATCH.

The GHEX library provides highly optimized halo exchange implementations for a number of commonly used grids (regular structured, cubed sphere, unstructured). For structured grids, GHEX eliminates unnecessary data copies within a shared memory node by implementing direct memory access to neighbour rank's data structures using XPMEM on CPUs, or CUDAIPC on GPUs. Each process (rank) registers raw memory pointers to its data grids. Other ranks on the same shared memory node can access those pointers directly, if they are neighbours and need to exchange the halos. This way the code combines the simplicity of a pure MPI application with all the benefits of a multi-threaded approach: there is no need for packing/unpacking into/from additional memory buffers when exchanging data within a compute node. To fully benefit from this approach, we use HWCART to maximize the number of per-node neighbours and minimize the surface to volume ratio of in-node sub-domains. For unstructured meshes, where an explicit sparse matrix - vector product is often used, GHEX can exploit optimized memory layouts, in which the non-local vector parts are stored in a contiguous chunk of the vector. In this case, the recv request can be submitted using the destination part of the vector directly, without using an extra buffer and an unpack. For all grid types GHEX can merge the halo data for multiple fields into a single buffer in order to decrease the number of exchanged messages, and improve the effective communication bandwidth.

GHEXBENCH is a sophisticated halo exchange benchmark that allows the users to observe the impact of all the developed software components, and compare the optimized GHEX code with a standard MPI implementation. The benchmark emulates a typical halo exchange pattern which arises in many scientific codes using a 3D cartesian grid, where each subdomain is connected to 26 neighbour domains. GHEXBENCH uses HWCART, or MPICart, GPUs, or CPUs, a pure MPI model, or a multithreaded model. The parameters also include grid size, data type (float, double), halo width, number of exchanged fields, in addition to a number of different halo exchange algorithms and strategies. GHEXBENCH can be used to choose an optimal implementation for a given architecture.

Transport layer benchmarks

The transport benchmarks measure the bi-directional point-to-point communication bandwidth in multi-threaded applications. Two versions of the benchmarks are compared: an OOMPH implementation, and an equivalent MPI implementation. The code launches two MPI ranks, one per compute node. The ranks spawn Nthr threads, and each thread keeps Ninfl messages in-flight. As soon as any asynchronous send/recv requests are completed, new ones are submitted. This behaviour emulates applications, in which communication is fully asynchronous, like DISPATCH. Figure 45 below shows the benchmark results on Betzy. In the sequential case (left plot), all codes, including the standard OSU benchmark, perform similarly. In the multithreaded tests (right plot), OOMPH with the UCX backend shows a much better performance than the MPI equivalents, especially for smaller messages below 100KB, and is able to saturate the HDR100 bandwidth for messages of 50kB and larger. On the contrary, the MPI-based benchmarks visibly suffer from multithreading. Since message sizes below 100KB are realistic in many halo exchange scenarios, OOMPH has a definite impact.

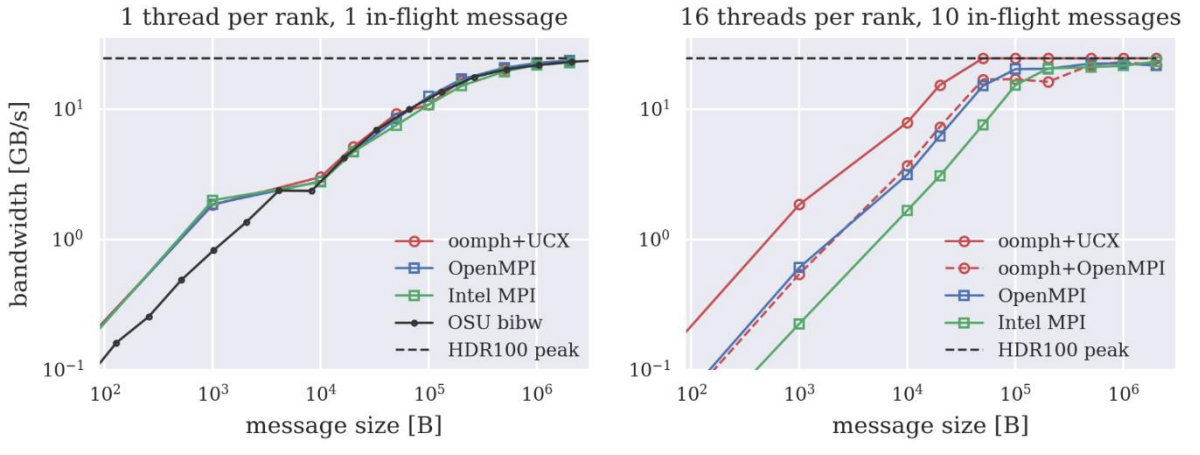


Figure 46: Transport layer benchmark results on Betzy

Halo exchange benchmarks

Performance of the high-level halo exchange API was evaluated using GHEXBENCH, by looking at different isolated tests: impact of the hardware-aware communicator HWCART, impact of RMA and multithreading strategies, and large-scale weak scaling. In order to both assess the intra- and inter-node performance, the benchmarks were run on a single node, as well as on 27 nodes (in a 3x3x3 spatial configuration - the smallest number of nodes in 3D that gives a realistic communication pattern). A larger number of nodes was used for evaluation of the scaling for Cartesian and, separately, unstructured grids.

Hardware-aware communicators

Figure 46 below demonstrates the impact of HWCART on the performance of halo exchange on Betzy. Global halo exchange bandwidth, computed as the number of halo bytes divided by the average exchange time, is shown for different scenarios. In single-node tests, 128 ranks are arranged into a periodic [8, 4, 4] rank grid. In multi-node runs the global rank grid dimensions are [24, 12, 12]. The results show the effective memory bandwidth of halo exchange for one data field of 128^3 double precision numbers per rank. Within a single node the impact of HWCART is small for the pure MPI code, but substantial for the XPMEM implementation. Overall, GHEX can yield up to 50% improvement over MPI for large halos. In multi-node runs the impact of HWCART is much more pronounced also for the pure MPI implementation: the used rank-to-node mapping resulted in compact per-node subdomains, which maximized the number of in-node neighbours and minimized the off-node communication. Overall, using GHEX boosts the performance by up to factor 2 compared to a pure MPI implementation.

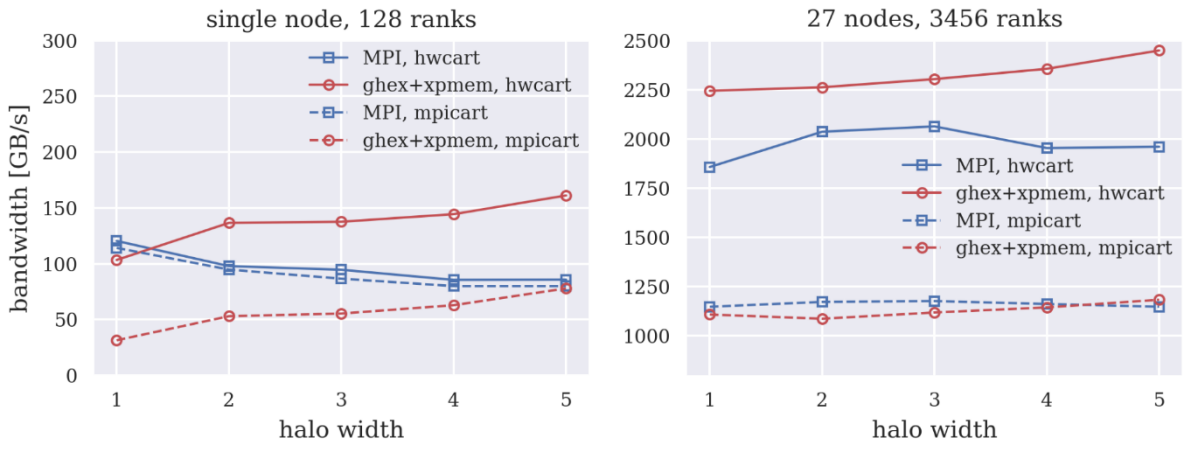


Figure 47: Impact of HWCART on the performance of halo exchange on Betzy

Halo-exchange strategies

Figure 47 below shows the results obtained on the Piz Daint Multicore partition (Cray XC40 compute nodes: dual socket: 2 x 18 cores). We compare a pure MPI implementation with GHEX (using MPI backend for off-node and XPMEM for in-node communication). Both the pure MPI, and the MPI+threads scenarios are tested; the former uses 36 ranks per node, the latter starts 1 rank per socket and 18 threads per rank. The domain size per PE is 128^3 (double precision), 2 fields are exchanged. All results are obtained using a staged exchange algorithm, where the halos (including diagonal corner elements) are exchanged along the XYZ axes in three sequential steps. Although this leads to an increased amount of transferred data compared to a single-stage exchange, it also reduces the number of messages across the fabric and leads to better performance in general for this architecture.

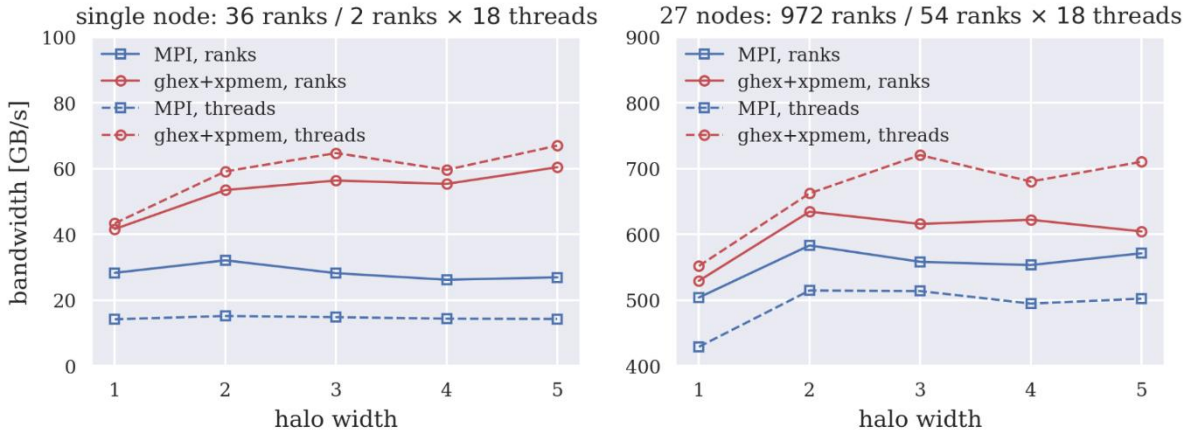


Figure 48: Results obtained on the Piz Daint Multicore partition

The performance advantage of direct memory access (using XPMEM between ranks, and shared memory between threads) is apparent for all configurations and leads to more than three times throughput increase w.r.t. the pure MPI code for some cases. The difference is particularly striking in the multi-threaded configurations, where GHEX achieves the highest performance. The multi-

node benchmarks also show that GHEX can retain the benefits from the intra-node optimizations despite the usage of equivalent calls to the same MPI library for off-node transport.

Figure 48 below shows the results from a similar benchmark on the Piz Daint GPU partition (CrayXC50 compute nodes: 12 cores, 1 NVIDIA Tesla P100). Here, we allocate 4 fields (double precision) of size 64^3 per rank/thread which reside in the GPU memory. The MPI implementation uses cuda pack kernels and GPU-aware MPI calls, while GHEX employs direct memory access among threads (shared memory) and processes (cudaIPC) on the same node and pack/unpack kernels between nodes. Both the multi-threaded and single-threaded results demonstrate the speedup gained by GHEX, especially for large halos.

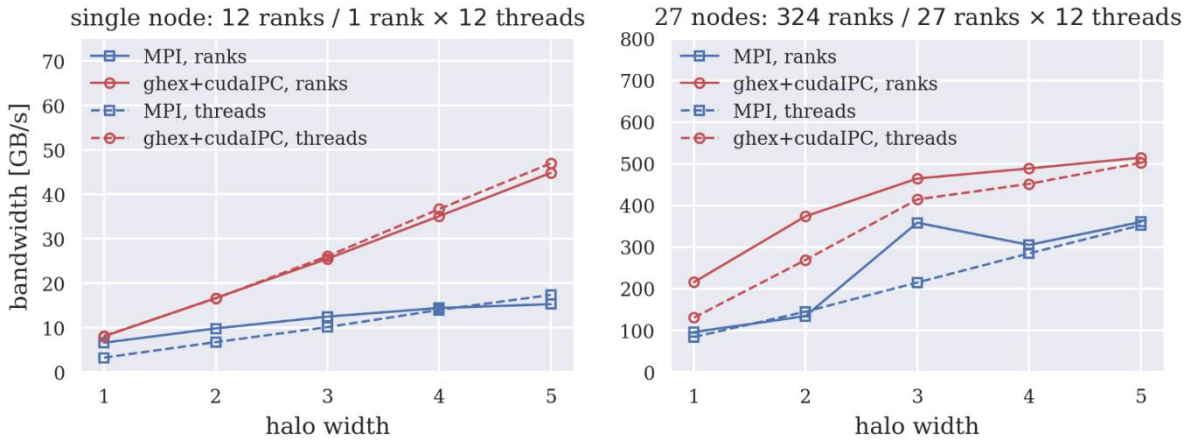


Figure 49: Benchmark results on the Piz Daint GPU partition

Large scale benchmarks

In Figure 49 below, we demonstrate that GHEX scales up to more than 12k cores on Piz Daint multicore partition and 512 GPUs on Piz Daint Hybrid partition, respectively, with a constant load per PE using the GHEXBENCH application (halo width=2, 4 fields, double precision). The execution times are normalized with respect to the fastest time for the baseline of 27 nodes (3x3x3 spatial configuration). We observe that both the multi and single threaded benchmarks scale well and outperform the respective MPI benchmarks in absolute terms. Furthermore, the usage of cudaIPC as well as optimized kernels result in twice better performance for the GPU benchmarks.

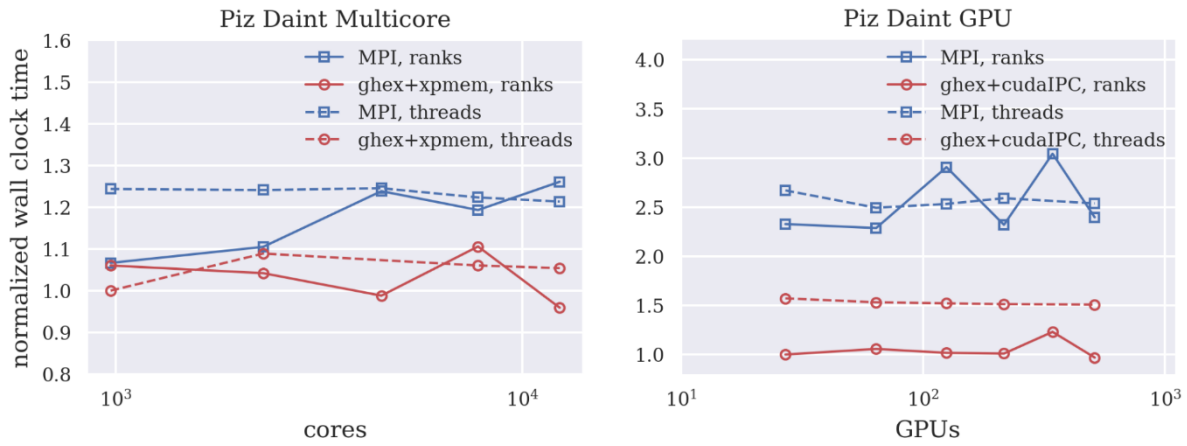


Figure 50: GHEX scaling on the Piz Daint multicore and hybrid partitions

Figure 50 below shows weak scaling benchmarks on the Piz Daint GPU partition using unstructured meshes generated with the Atlas library. The plot on the left shows halo exchange times with CPU storage and 12 ranks per node, while on the right storage is GPU-only and only 1 rank per node is used. In both cases 4 integer fields are exchanged, with halo depth = 2 and MPI as the transport backend. The grid is an Octahedral Gaussian grid (the type used e.g. in the FVM of the IFS model at ECMWF), with 100 vertical layers and respectively 160 and 320 (CPU) and 80 and 160 (GPU) parallels between the Pole and the Equator in the baseline configuration (2 nodes). When reaching the largest run sizes (512 nodes for the CPU storage and 2048 nodes for the GPU storage) this translates in both cases to the same two final grid sizes, with a grid spacing of ~5km for the smaller grid and ~2km for the larger one. The implementation relies on Atlas only for mesh generation, and uses the GridTools storage module as the backend for the fields.

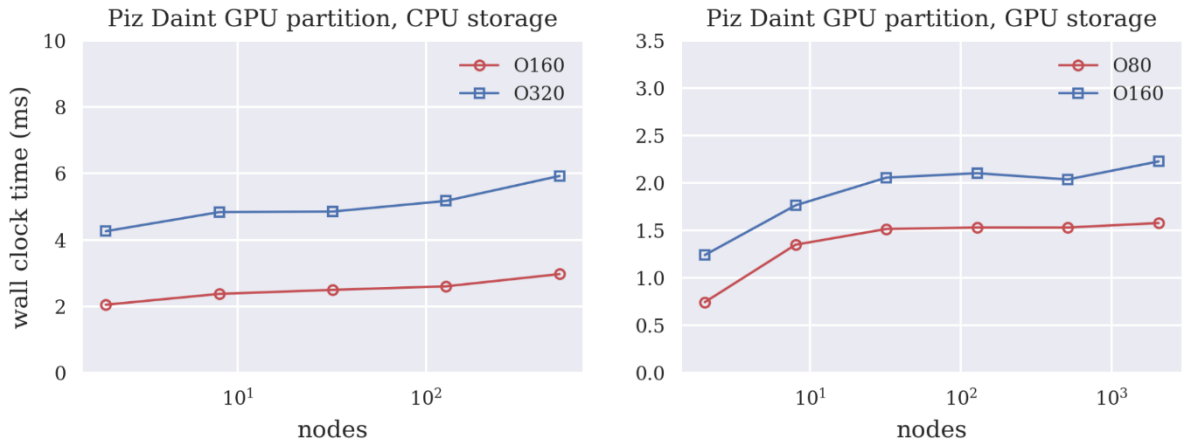


Figure 51: Weak scaling benchmarks on the Piz Daint GPU partition using unstructured meshes

10.3 Interactions with stakeholders, users, outreach and publications

Throughout the development of GHEX we have been in active contact with BIFROST and DISPATCH developers at the University of Oslo. The Fortran interface to GHEX has been designed in such a way as to minimize the amount of work needed to adapt both codes. Both

DISPATCH and BIFROST have been modified to make GHEX calls instead of the native MPI calls. In both cases the codes have been adapted with little effort, and in a flexible manner: the user can decide whether to use GHEX, or the original implementation at compile time. We have also started GHEX branches in the main development repositories of both BIFROST and DISPATCH at the University of Oslo. Consequently, any new development on the scientific side can be quickly merged with our communication backend.

GHEX integrates with unstructured applications as well by supporting communication patterns based on unstructured meshes. By passing to the halo-exchange setup functions the proper function to gather the halo information, GHEX can handle those cases similarly to the Cartesian ones. As for the Cartesian use cases, we developed user-side functions for typical cases, for instance general unstructured graphs in CSR format, and for interfacing the mesh and domain decomposition done by the Atlas library. Atlas is developed at the ECMWF where it is used in weather and climate production applications such as IFS. As a proof of concept we also developed user-side functions for inflated-cubes meshes, used in other weather and climate models, such as FV3, which is the NOAA global weather forecasting model. The latter is being evaluated to speed up the communication operations from the Python driven models.

GHEX has been introduced in GTBench, which is a mini diffusion-advection application to represent typical weather and climate computations. The application uses GridTools [72] to allow the execution on both CPUs and GPUs, and GHEX can be used to perform halo-update operations. This code had been used in the procurement phase of the LUMI pre-exascale computer.

An important effort actively pursued at ETH is the development of a Python framework for developing weather and climate applications, whose main component is GT4Py, a high-level interface to specify computational kernels in the field. The collaboration with the GT4Py team led to the implementation of the Python bindings for GHEX, and improved the implementation quality of GHEX itself by making it more uniform and amenable to automatic code generation, which is an important target for that project.

GT4Py is also used by another group to develop a Python version of FV3. The use of GHEX is being introduced to overcome a limitation due to the ability of GHEX to reduce the number of communications steps in their application, which requires frequent round-trips to Python. The initial results are very encouraging.

GHEX has also been actively presented in workshops, such as SIGMA2 LUMI workshop [73], or PADAL [74], where international researchers working in HPC applications discuss the hard topic of data locality and movement. GHEX received great interest from the attendees. While there was skepticism on the adoption of UCX, our benchmarks show now that it was the right choice to make.

The library was presented at several venues, including: PASC’21 conference; PRACE Inter-WP Topical Session “Exascale for European Datacentres"; and WP8 Online Session and Project Input for Yr1 Progress Report; vulcan Inc, USA; WholeSun workshop 2021 where partners working on DISPATCH, BIFROST and other applications attended.

Publications

GHEX has been presented at the PASC21 conference (“GHEX: Performance Portable Communication for Grid Applications”, Marco Bettiol, Fabian Bösch, Mauro Bianco, John Biddiscombe & Marcin Krotkiewski).

We are currently in the process of describing the details of the implementation, optimization strategies and benchmarking results in a paper ("GHEX: Generic Halo-Exchange for Exascale", not yet published).

10.4 Overall assessment of achievements and future developments

The major goal of the GHEX project was to develop a future-proof and performance-portable halo exchange library for modern and future HPC architectures that will be relevant for a large scientific audience. To achieve this goal, the design and the development were performed in a close collaboration with the strategic scientific communities: the developers at ETH and their collaborators in the weather and climate modelling, and the researchers at the RoCS astrophysics Centre of Excellence at the University of Oslo. From the functional point of view, GHEX now supports a number of common grid types, the modern C++ implementation is clean and overhead-free, and the Fortran and Python bindings further extend the reach of the code. The code is portable and has been tested on Intel and AMD CPUs and GPUs. It supports the pure MPI programing model, as well as the multi-threaded model, and is suitable for both classical grid-based applications, as well as task-based environments. Modern software development methodologies have been adopted, from agile development to the implementation of CI/CD workflows, which are currently based on GitHub actions. Finally, we've adapted BIFROST and DISPATCH to use our interfaces and made sure that the integration is smooth and relatively simple.

The major focus points throughout the development of GHEX were scalability and performance. GHEX was an experimental arena, where we tested several ideas on how to improve the performance of the halo exchange: using direct memory access instead of MPI-like communication for threads and ranks within the same compute node (using XPMEM for CPUs and CUDAIPC for GPUs), using different communication backends in addition to MPI (UCX and libfabric), implementing a range of different exchange algorithms (direct 26 neighbours vs. staged, exchanging field by field vs. packing all fields into a single buffer). We compared the performance to simple synthetic benchmarks, the roofline model (the memory and interconnect bandwidth), as well as a pure MPI implementation. During the project it became clear that to properly optimize the code for some architectures, and to provide a decent speedup over standard MPI, several more software components in addition to the originally planned halo exchange library were needed. Consequently, we have developed a range of general-purpose tools that we believe are useful and can benefit many HPC applications.

HWMALLOC is a hardware-aware memory allocator for HPC. The existing memory allocators only manage CPU memory, and only keep allocation pools for relatively small allocations. Large buffers, such as the ones used in DISPATCH, are immediately released to the OS upon freeing. Since DISPATCH relies on frequent (de)allocation of message buffers, standard allocators result in large overheads connected with memory allocation. Moreover, the existing allocators do not provide user-specified hooks to initialize the memory. HWMALLOC manages both CPU and GPU memory domains, and provides a method to automatically register the memory for use with RDMA operations of modern interconnects, which makes it more suitable for HPC.

The need for HWCART arose when optimizing GHEX for AMD Epyc based nodes, where the memory hierarchy consists of more levels than on most other architectures to date. Depending on how the ranks are bound to the cores, the performance can differ by a factor 2, especially when using direct memory access between the ranks/threads. The native MPI Cartesian communicator assigns ranks to PEs in a linear, Z-first order. Instead, HWCART can be used to build compact

cubes of ranks, minimize the surface to volume ratio of per-node grid subdomains, maximize the shared memory communication, and minimize the off-node data transfer. HWCART can benefit all codes that rely on Cartesian communicators, including pure MPI codes that do not use GHEX halo exchange.

OOMPH has been developed as an alternative API for the exchange of tagged messages, suitable for the task-based programing model. An MPI program checks for asynchronous communication completion by actively testing the request handles. This complicates the code, because the requests have to be stored, maintained, and explicitly checked. Modern communication libraries, such as UCX and libfabric, offer the users an alternative completion mechanism based on callbacks. The user only has to progress the communication backend using, e.g., a progress() function, and the callbacks are invoked upon the completion of any request. OOMPH provides the callback functionality as a C++ and Fortran API. Following the modern C++ approach, OOMPH also implements futures, to which the user can attach continuations. Finally, OOMPH implements a “send-and-forget " communication model, in which the backend tracks the state of the send requests and automatically frees the message buffer upon completion. This further simplifies the task-based codes like DISPATCH. When used together with HWMALLOC, there is no (re)allocation and registration overhead connected with this model. Finally, benchmarks have shown that using OOMPH with UCX and libfabric backends substantially improves multithreaded communication performance over MPI.

To study the performance and scalability of the developed tools on various architectures, we have developed GHEXBENCH. This benchmark allows the user to test the performance of GHEX throughout the large parameter space that includes all the discussed tools, communication backends, RMA optimizations, exchange algorithms, GPUs vs. CPUs, threads vs. ranks, halo width, number of fields, and data type.

The goals of the project outlined in the proposal have been reached, and even more useful tools and knowhow have been developed than initially planned. However, the project is not yet finished and several more directions can be explored. More performance analysis and optimizations may bring further improvements, especially for the in-node exchange using RMA. We observe that in some cases the pure MPI implementation is still faster, although the back of the envelope performance estimates say it should not be. This is most likely due to some non-trivial memory and cache effects, which we do not yet understand. Also, more optimizations and tuning might be necessary for LUMI: although we have tested GHEX on Betzy, which is a similar architecture CPU-wise, and on AMD and NVIDIA based GPU systems, the actual architectures of the pre-exascale computers are not available. We will benchmark GHEX on pre-exascale computers as they become available, especially LUMI, thanks to the project extension.

11 ParSec: Parallel Adaptive Refinement for Simulations on Exascale Computers

11.1 Introduction and summary

One of the key capabilities required by CFD codes to take advantage of leading-edge computing resources is the automation of the mesh generation and adaptation processes. Manual mesh generation and tuning is not feasible in an Exascale simulation workflow, Adaptive Mesh Refinement (AMR) automates this process providing higher efficiency and robustness to the codes.

ParSec brings together well-known CFD practitioners with the aim of sharing best practices, and collaboratively modernise the AMR implementation of three leading-edge CFD community codes. The partners involved in the project are Barcelona Supercomputing Center, the KTH Royal Institute of Technology and together Cenaero and Université de Liège. The community codes brought by these institutions are: Nek5000, the scalable high-order solver for computational fluid dynamics from KTH/UIUC, Alya, the high performance computational mechanics solver from BSC, and Argo, the high order multiphysics solver from Cenaero. These three CFD solvers cover the main approaches for the solution of PDEs using both structured and unstructured meshes: finite element, finite volume, and spectral elements.

Beyond the AMR-enabled CFD codes, some libraries providing mesh functionalities are included in the project, namely: MAdLib, an open source library including all the mesh functionalities of Argo, Gmsh an OS finite element mesh generator with a large community of users and GeMPa an OS including all the geometric mesh partitioning tools available in Alya.

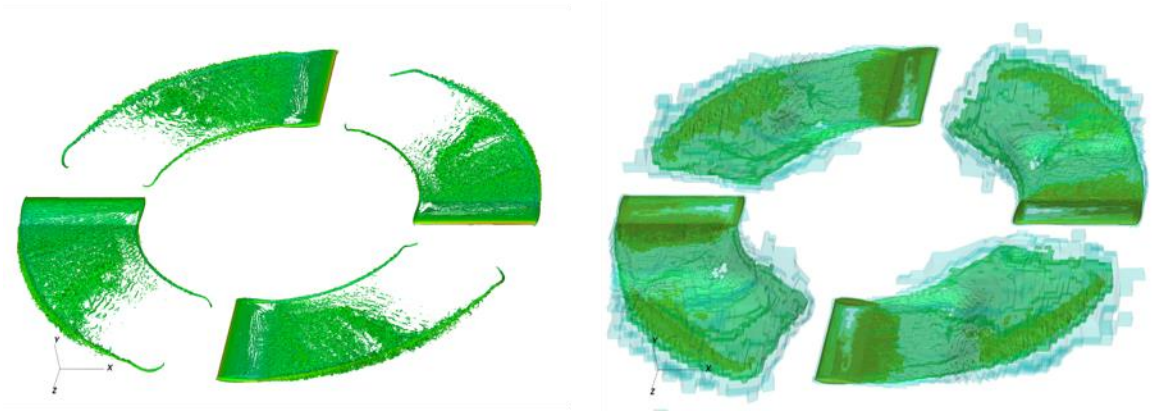


Figure 52: A vortical structure of the flow around a simplified rotor (left) and the refinement level structure for the same simulation (right). Work performed in collaboration with CINECA.

Throughout the project, we have achieved qualitative advances in all the codes. Alya and Argo (via MAdLib) have evolved from a sequential version of AMR to a finished and tested parallel implementation. The developers of both codes have opted for an interface freezing approach that interleaves interface displacement and local remeshing. For MAdLib, a load balancing mechanism has been included in the remeshing phase. It has been tested for the generation of anisotropic meshes using up to 768 CPU cores. For Alya, scalability results have been reported using up to 4096 MPI processes.

Nek5000 started the project with an already working parallel AMR implementation, which has been optimized by reimplementation and modularisation of the code: exploring different

partitioning libraries, improving pressure preconditioning, and investigating various strategies for generating high-order hex-based meshes. Moreover, a GPU-version of Nek5000 relying on an OpenACC/Cuda framework for NVIDIA accelerators has been developed and tested in various systems.

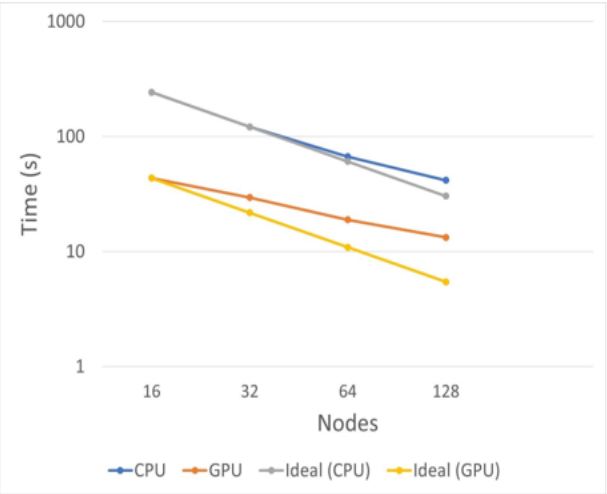
Regarding Gmsh, several improvements have been implemented and tested in the mesh generation pipeline. Both the coarse-grained parallelism of 1D and 2D algorithms and the fine-grained multithreading of the new 3D Delaunay mesher and optimizer have been improved. The developments have been tested in a 420 million tetrahedra mesh of a nozzle, courtesy of NASA Glenn Research Center. Moreover, during the project, Gmsh has been integrated within MAdLib and Alya. Finally, the mesh partitioning functionality of Alya has been extracted into the stand-alone OS library GeMPa.

11.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

Nek5000 - KTH

There are two main aspects of the Nek5000 development within ParSec: an improvement of an Adaptive Mesh Refinement (AMR) algorithm and parallel performance on the heterogeneous architectures. Although we did not consider the mortar elements or p-refinement strategies, we have significantly enhanced AMR making it a robust tool for performing industrially relevant simulations. We achieved it by reimplementation and modularisation of the code, exploring different partitioning libraries, improving pressure preconditioning and investigating various strategies for generating high-order hex-based meshes. Regarding heterogeneous architectures, we have been working on a GPU-version of Nek5000 relying on a OpenACC/Cuda framework for use on NVIDIA accelerators and we are also currently porting this to a OpenMP/HIP framework for use with AMD systems.

The performance of Nek5000 has been examined on various systems. For these measurements we used the turbulent pipe case (conformal mesh) case. In Figure 52 (left), we consider the performance on the JUWELS Booster system based on NVIDIA A100 GPUs and AMD EPYC 7402 CPUs. The results demonstrate scalability up to 512 GPUs (128 nodes) with good efficiency.



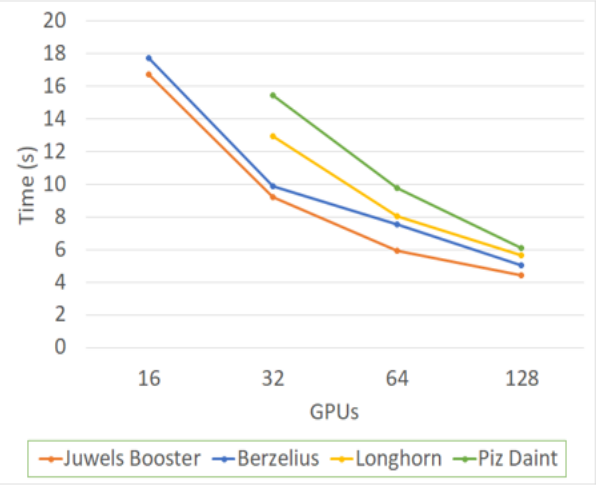


Figure 53: Scaling of the OpenACC version of Nek5000 on several GPU enabled machines. (left) Results in JUWELS Booster. (right) Results including JUWELS Booster at Jülich in Germany, Berzelius at NSC in Sweden, Longhorn at TACC in the USA and Piz Daint at CSCS in Switzerland.

In Figure 52 (right), good scaling results up to 128 GPUs is shown in several systems. The speedup from using 32 GPUs NVIDIA P100 GPUs on Piz Daint to using 32 NVIDIA A100 GPUs on JUWELS booster is also consistent with the increase in the dual precision flops on P100 (4.7 TF) to the A100 (9.7 TF). For larger numbers of GPUs, the effect of the network becomes more significant and the increase in performance from the faster GPUs is lower.

MAdLib/Argo - CENAERO

During the project, a parallel mesh adaptation approach has been implemented and tested within the mesh adaptation library MAdLib. Though the adaptation features of this library were initially available in sequential setting only, it has been possible to take advantage of the sequential remesher for distributed purposes. Keeping the former implementation as it was, the method consists in iteratively storing the already adapted part of the mesh and load-balancing the remaining part of it. Each time the adaptation procedure is called, the interfaces between partitions are fixed, so that the sequential remesher can work properly. The load-balancing algorithm is based on the ParMetis library, which has been interfaced within MAdLib. Several 3D test-cases have been run, involving size-field based mesh adaptation. Two significant aspects have been considered: the anisotropy handling and the distribution of the meshing load. In order to get a mesh which properly complies to a strongly anisotropic size-field, the recursive partitioning process described above evaluates the remaining adaptation work and modifies the partitioning accordingly. The distribution of the work between the partitions is computed by the evaluation of the difference between the current mesh and the size-field the adapted mesh should comply to. By combining these two strategies, the remesher can modify the initial mesh so that it satisfies size-fields which can be both anisotropic and non-uniform in terms of size-field complexity.

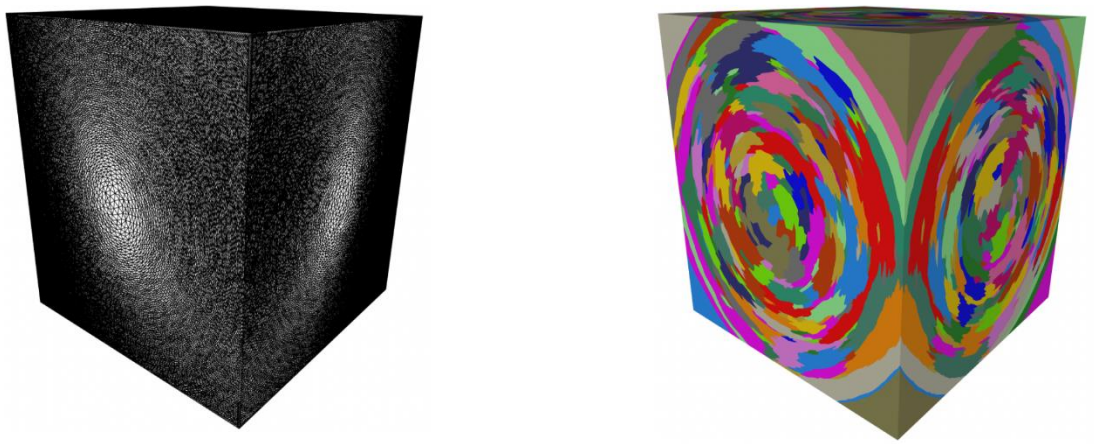


Figure 54: Cube adapted mesh with respect to a radial anisotropic size-field, obtained with a 768 cores parallel mesh adaptation. Left: the final mesh with around 7.2 million elements. Right: partition represented by different colours.

Figure 53 shows an anisotropic mesh over a cubic domain used as a test case. It has been run on an increasing sequence of number of cores, from 24 to 768 cores on Intel Ivy Bridge at 2.7Ghz. An initial partitioned mesh with around 3 million tetrahedra is considered. Then, a size field is defined at the vertices of this mesh, and the complexity of it is tuned so that an adapted mesh with respect to this size-field would be 1,5x finer than the initial mesh. The overall process is repeated twice, resulting with a mesh which is 2,25x more refined than the initial mesh. As shown in Figure 54

below, the method scales well up to several hundreds of cores. This functionality is currently being integrated within Argo.

#cores	Time (s)
24	445.75
48	234.91
96	131.59
192	76.27
384	49.15

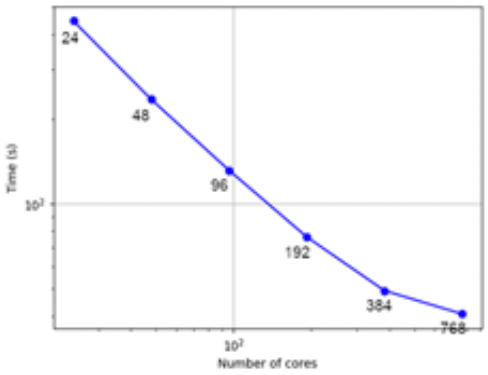


Figure 55: Strong scaling test for anisotropic refinement of a mesh within a Cube, through two refinement steps the mesh size is multiplied by 2.25x.

Gmsh - Université de Liège

Several improvements were implemented and tested in the mesh generation pipeline of Gmsh. Both the coarse-grained parallelism of 1D and 2D algorithms and the fine-grained multithreading of the new 3D Delaunay mesher and optimizer were improved. The previously sequential evaluation of mesh size constraints was fully parallelized using OpenMP, and the partitioning of large meshes using Metis was also improved, including the creation of the full partition topology. Various GPU offloading opportunities have also been investigated, in particular for high-order curved mesh generation. Figure 55 shows typical results obtained with the improved meshing pipeline applied to a CAD model of a nozzle, courtesy of NASA Glenn Research Center. The full meshing pipeline (1D, 2D and 3D meshing, followed by mesh optimization) was carried out on AMD Epyc Rome 7542 CPUs at 2.9 GHz. A quality mesh of about 420 million tetrahedra, adapted to a priori mesh size constraints, was generated in about 15 minutes with 64 threads (see Table 9), using 52 GB of RAM. While there is still room for improvement, this new level of performance clearly allows one to envision the generation of quality initial meshes for future exascale applications on complex geometries.

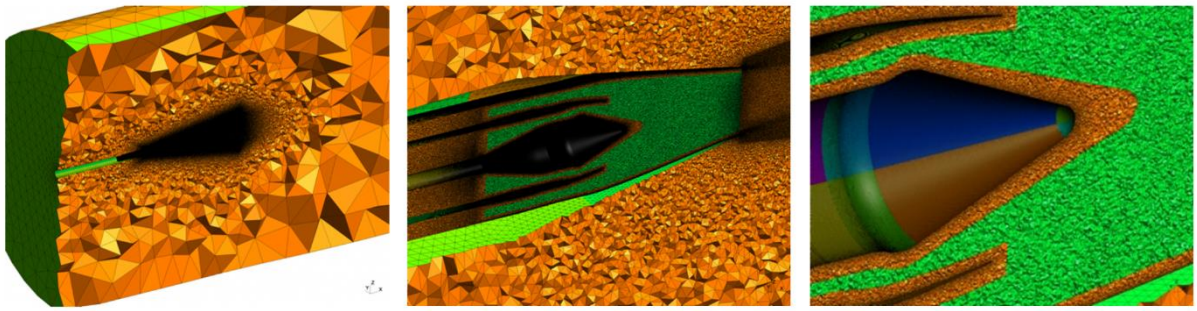


Figure 56: Mesh of a nozzle (courtesy NASA Glenn Research Center) with a priori mesh size constraints, leading to about 420 million tetrahedra. Three zoom levels on a visualization of the mesh.

#CPU-cores	Time (s)
1	11888
4	4744
8	2405
16	1326
64	924

Table 9: Scaling test for the generation of a 420M elements mesh of a nozzle. Parallelization base on threading using OpenMP on an AMD Epyc Rome 7542 CPUs at 2.9 GHz.

Alya - BSC

In the course of the ParSec project, the AMR parallel workflow of Alya has been completed; only a naïve sequential approach was available before the project start. As shown in Figure 56, the workflow is a composition of various functionalities. In short, once the solution of the PDE under study is available, an error estimator is used to define the refinement requirements. Then starts the process of mesh adaptation/refinement. We have implemented an interface freezing approach where, at each time step, the nodes at the interface between two subdomains are blocked, and the interior of the subdomains are remeshed. Then the interface is moved, generating new subdomains, and the process is repeated until no frozen nodes remain. Once the new mesh is obtained, the fields defined on the initial mesh are interpolated to the new one through a parallel interpolation module. Finally, if the adapted mesh distribution is significatively unbalanced, a dynamic load balancing process based on mesh repartitioning is activated.

In the course of ParSec we have implemented the parallel mesh adaptation process and optimized the parallel interpolation. For the remeshing part, we have integrated into Alya both MAdLib and Gmsh. This task has been done in close collaboration with the partners hosting each library. On the other hand, the error estimation part has been left for application scientists since is a physics-dependent functionality; we have had a close collaboration with the EXCELLERAT and CoEC Centers of Excellence in this regard. The parallel load-balancing mechanism was available in Alya before the project start. All these developments are now available in the open source version of Alya [75].

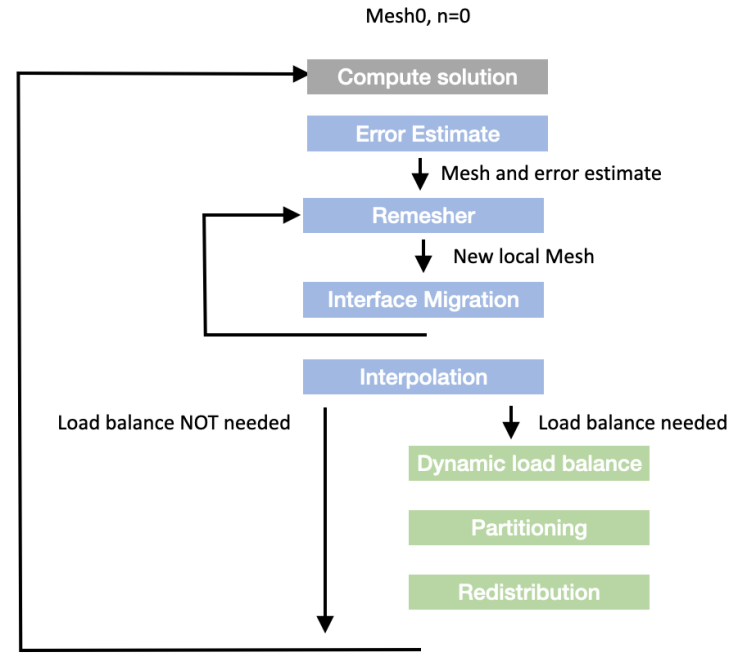


Figure 57: Parallel AMR workflow implemented in Alya.

In Figure 57, a strong scaling test performed in the Hawk supercomputer from HLRS is presented. The mesh under consideration has 16M tetrahedra and the number of CPU-cores used ranges from 512 to 4096. The parallel efficiency obtained for the simulation with and without AMR is the same when using 4096, but for 2048 CPU-cores there are significant differences, further investigation is required to understand this behaviour and extend the scalability to larger number of CPU-cores.

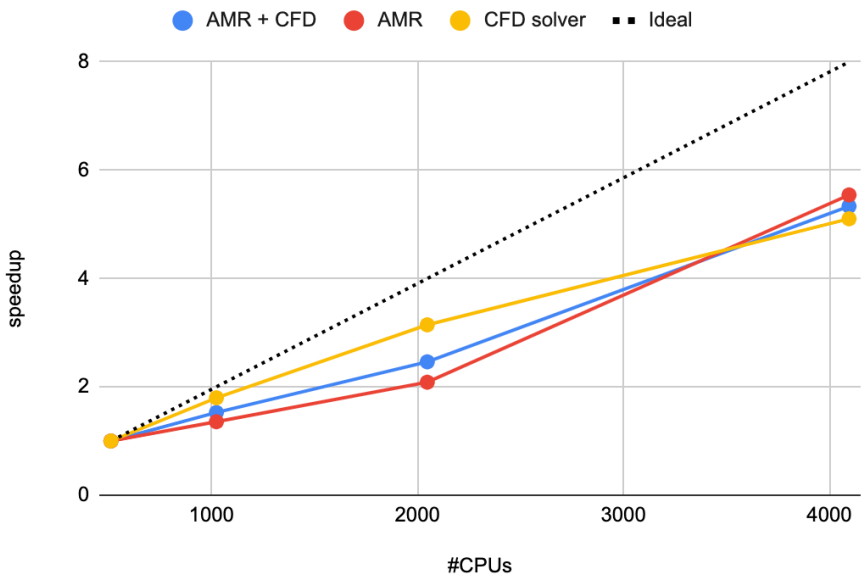


Figure 58: Strong scaling of the AMR implementation of Alya in a tetrahedral mesh of 16M elements, using the Hawk supercomputer from HLRS.

Finally, in ParSec, the SFC-based mesh partitioning tools of Alya have been extracted into the stand-alone open source library GeMPa [76]. The geometric mesh partitioning implemented in GeMPa is used in Alya for mesh partitioning and dynamic load balancing purposes. The following table shows a comparison of the Alya SFC partitioner vs. the well-known library Zoltan (version 3.8.3). Tests were run on the MareNostrum IV supercomputer, and both libraries were compiled using the intel/18.4 compiler. For Zoltan, the Zoltan_LB_Partition was used with the HSBC option. The problem under consideration is the mesh around an airplane with 250M elements.

Partitions	Nodes uses	Time Alya (s)	Time Zoltan (s)	Speedup
384	8	0.25	0.87	3.5x
768	16	0.15	0.54	3.6x
1536	32	0.10	0.48	4.8x
3072	64	0.07	0.50	7.1x
6144	128	0.08	0.79	9.9x

Table 10: Comparison of Alya SFC-based mesh partitioning vs Zoltan v 3.8.3. Partition of a 250M elements mesh around an airplane.

These tests show that although at some point the scalability stalls, the cost of generating a partition is almost negligible: 0.08 seconds to partition a mesh of 250M elements into 6144 subdomains. It also shows that the SFC-approach implemented in GeMPa clearly outperforms Zoltan. These tests were performed with Alya, however we have shown that the result and the performance obtained with GeMPa is equivalent to that obtained with Alya.

11.3 Interactions with stakeholders, users, outreach and publications

Nek5000 - KTH

Although both an AMR and GPU implementations are in the development phase, it has been shared already with external users e.g., within the framework of EXCELLERAT EU project, or Swedish e-Science Research Centre (SeRC). One of the examples is collaboration with CINECA (Italy), where both implementations are tested and developed. In the case of the GPU branch our collaboration focuses on implementation and optimization of the OpenMP GPU Offloading version. Two main aspects considered here are: the overlap between data movement between device/host and computation, and the overlap of multiple kernels over GPUs. We performed detailed profiling analysis looking for subroutines without data dependences and kernels that do not occupy a full GPU. Unfortunately, some limitations come from the OpenMP offloading model, which does not support the explicit execution of multiple kernels in different streams.

Another aspect of our collaboration is joint effort with Texas Tech University, where we investigate a fully turbulent straight pipe simulation using GPU-based computer systems in the US. In this case an OpenACC version of Nek5000 is used. We performed a number of benchmarks and validation tests comparing e.g., profiles computed by Nek5000 and OpenPipeFlow (an open source code aiming at turbulent pipe simulations). Results of both codes are in perfect agreement. Our goal is to perform full-scale production runs utilizing as much of the system as is possible. Two

specific benchmark runs we can mention here are: 0.3M and 7.2M elements simulation performed on 128 (Longhorn at TACC) and 4096 (Summit) NVIDIA V100 GPUs respectively.

Regarding the AMR branch, collaboration took place with CINECA that aimed at performing AMR simulation of the rotating parts e.g., of a drone rotor. This project deals with different aspects of a whole simulation work-flow starting with a hex-based meshing of a relatively complex object, through efficient mesh partitioning, and ending with data visualisation. This code version is used as well by communities in SeRC. In addition, we also prepared a paper for the HPC Asia 2022 Conference further describing the strong scaling results for Nek5000 [77].

MAdLib - Cenaero

In addition to the developments of the parallel adaptation features, MAdLib's API has been deeply modified and completed. Indeed, in order to make easier the use of the library with external software packages, the user interface has been made compatible with the C language, whereas it was only usable in C++ beforehand. Besides, this API now covers the mesh adaptation features, both in sequential and parallel settings. These functionalities are already available in the development trunk of MAdLib, and will be included in the next release 2.3.0. Particularly, the C API has been used by BSC in this project to integrate MAdLib into Alya as an anisotropic mesh adaptation tool.

Gmsh - Université de Liège

Gmsh is widely used both in academia and industry, and the improved parallelism resulting from the Parsec project will benefit all users requiring large scale mesh generation. Moreover, during the project an initial integration of Gmsh within MAdLib was carried out using the Gmsh C++ API, which will enable the seamless transfer of meshes between both libraries. This will provide new opportunities for flexible and efficient meshing, adaptation and remeshing pipelines using both libraries in the future. Finally, a new Fortran port of the Gmsh API was developed during the project to complement the existing C++, C, Python and Julia APIs, which will ease the integration of Gmsh in Fortran HPC codes. In fact, the Fortran API has been validated through the integration of Gmsh with Alya. All the developments carried out during the project will be available in Gmsh 4.9 [78], scheduled to be released by the end of 2021.

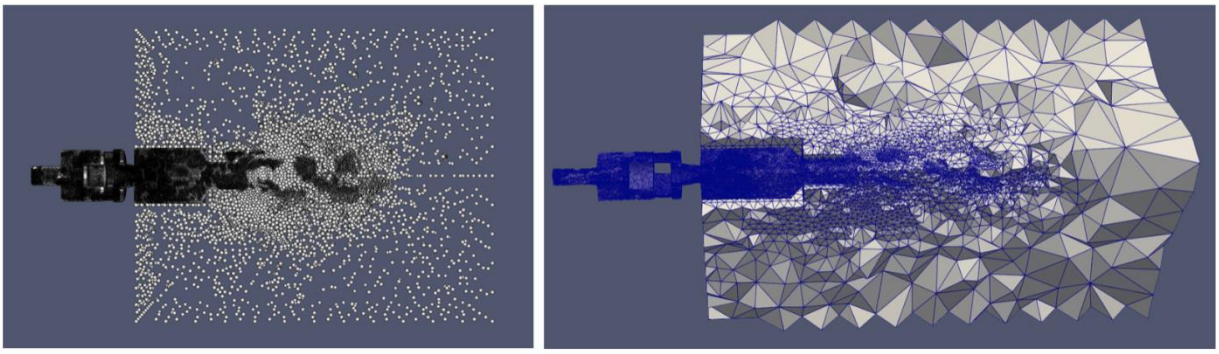


Figure 59: 20M elements mesh of the Preccinsta burner obtained through adaptive mesh refinement from 3M elements mesh. Test case studied in the CoEC Center of Excellence.

Alya - BSC

Alya is part of the software hub of various Centers of Excellence, the use cases of those CoEs directly benefit from the AMR capabilities developed within ParSec. In particular, we have set up a collaboration with the European Center of Excellence for Engineering Applications (EXCELLERAT) [79] in which it is being considered a use case for airplane aerodynamics and flow control via jet actuators; and also a collaboration with the Center of Excellence in Combustion (CoEC) [80], where the test case under consideration is the Preccinsta burner. An illustrative image of this test case is shown in Figure 58. The mesh resulting from the AMR process has about 20M elements, and it was generated from an initial mesh with 3M elements. In the left part of the figure are depicted the mass centers of the cells as particles, and in the right part, the tetrahedral cells. We can see how the mesh is well adapted to the vortex structures of the turbulent flow generated within the combustion cavity. In both cases, the error estimators to determine the refinement requirements have been studied from the application point of view in the CoEs, while the usability and performance of the code have been delivered by ParSec. Finally, note that all the developments carried out in this project are available in the open source version of Alya.

On the other side, the GeMPa library has been realized in the last phase of ParSec, and we have plans for using it in the context of the EuroHPC project NextSim [81]. NextSim focuses on preparing the flow solver CODA (the new reference solver for aerodynamic applications inside the AIRBUS group) for future Exascale computing systems.

11.4 Overall assessment of achievements and future developments

Overall, the four objectives set up for the ParSec proposal were:

1. Analysis of the various separation of concerns (SoC) used for the AMR implementation;
2. Analysis and optimization of the performance of the codes on (pre) Exascale architectures;
3. Deliver self-contained OS software components solving different steps of the AMR process;
4. Deliver three AMR-enabled CFD legacy codes to exploit (pre) Exascale systems.

Those objectives have been mainly achieved and delivered into production-ready software, taking advantage of a collaborative partnership. Further details on the various software packages included in ParSec are provided in the following paragraphs.

Nek5000 - KTH

The main goal of Nek5000 related development within ParSec was to make this code a robust solver using AMR and heterogeneous architectures and capable of solving industrially relevant problems. Building on previous work of CRESTA EU project, and through extensive collaboration with existing EU projects (e.g., EXCELLERAT) and various national organisations (e.g. SeRC) we have achieved most of our goals, although some problems require future investigation. One of the achievements is a GPU branch built on OpenMP/OpenACC. It reduces computational time 3-5 times with respect to the CPU version of Nek5000. However, there is still room for improvement, as only velocity solvers were optimised and a pressure solver needs more attention. This will be investigated in the future. On the other hand, there was a significant development of the AMR branch starting from code refactoring and modularisation, through testing different mesh partitioners (ParMETIS vs ParRSB) and working on hex-based meshing. It allowed us to perform the fully AMR simulations of the moderately complex cases e.g., a toy rotor with Reynolds number equal to 100000.

The main issue we have encountered is adaptation of the AMR framework to GPUs. It is because AMR work-flow relies on various libraries and communication intensive algorithms that pose problems porting to GPUs. The other important observation is the importance of the proper error indicator/estimator. Although the whole AMR framework allows dynamic mesh modification, the final quality of the solution depends mostly on the information where the mesh should be resolved/coarsened. That is why we devoted more time to the investigation of the solution sensitivity to the applied refinement (e.g., turbulent statistics in a turbulent pipe simulation) leaving development of the mortar elements to the future.

MAdLib - Cenaero

The parallel mesh adaptation developments introduced in MAdLib within the Parsec project constitute a strong basis on which further developments will be done. Indeed, though the parallel mesh adaptation does not scale on thousands of cores, it has shown great robustness and extends most of the serial capabilities to parallel settings. In order to improve the scaling for a larger number of cores, several ways should be investigated. As expected, when the number of cores increases, the code performances are slowed down by the large number of communications between partitions, whereas the actual adaptation work represents a smaller part of the full process. To overcome this issue, the number of interfaces between partitions should be minimized by a load-balancing correction procedure. This kind of algorithms already exists in the literature and already showed encouraging results in early tests.

Gmsh - Université de Liège

The improved performance of the parallel mesh generation pipeline in Gmsh introduced during the Parsec project allows one to envision the generation of quality initial meshes for future exascale applications on complex geometries. Indeed, quality meshes with hundreds of millions of tetrahedra can now routinely be generated in a matter of minutes. Also, the new Fortran Gmsh API developed during Parsec nicely complements the existing C++, C, Python and Julia interfaces, which will make it easier to access these pipelines from more HPC codes. Future developments should target further performance improvements in (especially high-order) mesh optimization, which could benefit substantially from GPU acceleration.

Alya - BSC

In ParSec, the parallel AMR workflow of Alya has been completed and also tested in various applications in collaboration with the Centers of Excellence EXCELLERAT and CoEC. The project's main goal was to end up with an AMR-enabled CFD solver, which has been achieved. Further development will be required to improve some aspects of the implementation, such as optimizing the interface motion to minimize the iterations needed to complete the mesh adaptation. Moreover, some communication episodes should also be optimized in order to obtain good parallel efficiency using larger numbers of MPI-processes. It is important to note that collaboration with other project partners has supported the progress in the AMR capability of Alya: in the course of ParSec the libraries Madlib and Gmsh have been integrated into Alya with the support of Cenaero and Université de Liège. Finally, GeMPa has been successfully extracted as a stand-alone library for SFC-based mesh partitioning.

12 NB-LIB: Performance portable library for N-body force calculations at the Exascale

12.1 Introduction and summary

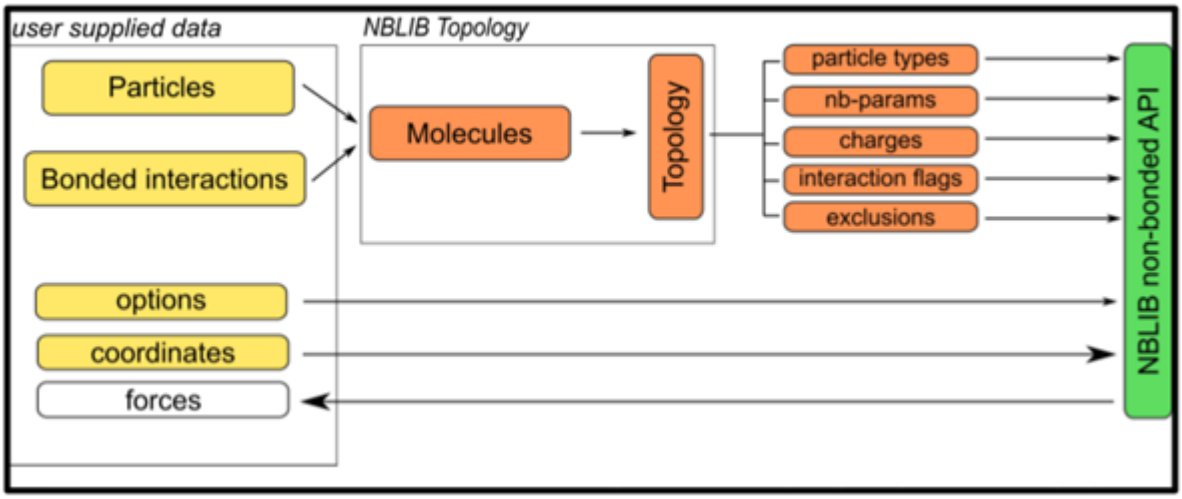


Figure 60: NB-LIB interface and data flow. A series of pre-processing steps prepares user supplied data so that the NB-LIB interface is generic, accepting only elementary types.

A large number of scientific applications use particle-particle interactions such as Molecular Dynamics, Monte Carlo or multiscale simulations in life sciences or materials. Further, several smaller codes or combinations of codes expose unique feature sets. However, while computers have become more specialized, many codes are not optimized for GPUs or other accelerators and it is increasingly hard to achieve parallelization. It is also very difficult to offer in a single application all the unique features and niche use-cases that the various existing many-body codes taken together support. This makes these codes increasingly difficult to use on new systems which are large and can be heterogeneous in terms of hardware.

One of the codes currently undergoing Exascale optimization efforts is GROMACS, also among the benchmark codes for pre-Exascale machines that have been coming online in recent years. While it has a long track record as a widely used and highly performant HPC code, it has not been designed with the goal of interoperability and many modules contain deep dependencies on other modules. In practice, this means that a potential caller of the GROMACS non-bonded forces code will have to provide valid instances of a wide range of GROMACS specific data structures that contain much more information than what would be required for computing non-bonded forces.

The goal of the NonBonded-LIBrary (NB-LIB) is to make the cutting-edge performance of GROMACS available through a high-level C++ API to its non-bonded force kernels. This goal has been achieved while also showing that it is possible to have a more BLAS-like interface for non-bonded force calculations. This interface only takes arrays of elementary types (integer and floating point numbers) and demonstrates a path forward for reducing module dependencies in GROMACS, while also making it potentially much easier to interface to other particle simulation codes, implement novel scientific workflows, or performance tune for specific hardware.

In order to provide even more performance and utility to users, NB-LIB has also exposed an API for other types of particle-particle calculations besides non-bonded forces, such as bonds or other types of two particle interactions, angles or other types of three particle interactions, and so on for four and five particle interactions. While still using the performant GROMACS kernels under the hood, this new listed force calculator API significantly simplifies the call stack relative to that in GROMACS, while also exposing the possibility of future performance optimization to allow cache reuse by dispatching particles that participate in multiple interactions simultaneously. This API also significantly streamlines the addition of new multi-particle interactions since only kernels and parameters need to be added, without the need to modify the dispatch logic.

In combination with the system setup functionality that NB-LIB offers, users are able to implement arbitrary workflows that might be required for their special use case while leveraging the performance of GROMACS for the force calculations. This way, future acceleration, porting, and library features will benefit all applications.

12.2 Benchmarking results on pre-exascale/petascale/Tier-0 systems

Librarization vs. Performance Optimization

The goal of the NB-LIB project has been to expose the underlying performance of the GROMACS non-bonded force calculation. This has been achieved with the implementation of a very low overhead abstraction layer that hides the complexity of the multistage setup code required for building the GROMACS non-bonded force calculation object. Additional complexity in terms of computing the neighbor lists which requires multiple function calls in GROMACS is also abstracted behind a single function call in the NB-LIB API. The construction of lists of neighboring particles, and attendant partitioning of particles across cores on a single node, and multiple nodes on a high performance computer, is a primary driver of GROMACS' performance. The other primary driver of GROMACS' performance is the fact that so much effort goes into performance tuning the non-bonded calculations to different hardware.

With the advent of NB-LIB, the excellent single node performance of GROMACS is usable by researchers in different settings. Firstly, those who want to either perform novel scientific workflows using a performant particle-particle calculation. Also, by researchers in an HPC context who want to do performance engineering without having to understand much about the internals of the GROMACS non-bonded force calculation.

Towards Performance Engineering

In the context of performance engineering, NB-LIB opens up a number of exciting possibilities. Two main challenges face particle-particle simulation codes in the pre-exascale era. The first is the proliferation of different accelerators and different compute architectures. The second is the acute need to be able to scale up calculations to a sufficient size to fully utilize available hardware. NB-LIB is an important step on both of these fronts.

Taking first the question of targeting heterogeneous architectures, modularization efforts such as NB-LIB are critical. The reason for this is that different hardwares do and will continue to have different limitations in terms of latency of data transfers, size of available cache and registers, and overall throughput per execution unit and device. The way that NB-LIB can be helpful in such a complex and dynamic environment is that in addition to exposing generic functions for constructing and using a non-bonded force calculator, it allows complete flexibility in terms of where computations are performed, in terms of CPU or GPU. It also exposes directly to

performance engineers the ability to manage data transfers between host and device. The ability to manage both where computations are performed, and how data flows between computations is an important breakthrough that was latent in the existing GROMACS codebase but that is exposed as a result of the NB-LIB API. With these tools it is now feasible to write custom schedules of non-bonded force calculations that are finely tuned to the needs and requirements of different hardware architectures.

NB-LIB also exposes the performance of the GROMACS listed forces (bonds, angles, etc) calculations. In the case of the non-bonded force calculator, NB-LIB is a thin interface layer to the underlying GROMACS call stack. In the case of the listed forces calculations, NB-LIB is actually a rewrite of the call stack, only keeping the force kernels from GROMACS. This is actually highly relevant for performance. The calculation of listed forces is entirely memory bound. The kernels are rather simple, and the key aspect for performance is exploiting cache locality so that positions of particles do not have to be loaded repeatedly to compute forces for the numerous interactions that a single particle may participate in. While the existing GROMACS implementation does a decent job of exploiting cache locality, the call stack does not allow for dispatching all the interactions of a single particle simultaneously. The redesigned call stack in the NB-LIB implementation, on the other hand, allows complete flexibility in terms of how interaction computations are allocated. This means that, just like in the case of the NB-LIB non-bonded force calculation API, it is possible to finely tune force computation schedules to different hardware in terms of the number of interactions that are grouped, all the while having optimal cache reuse so that particle positions do not need to be reloaded multiple times for the same particle. Due to the heavy emphasis on code quality and testing in NB-LIB, we have implemented end-to-end performance tests which use the Google test framework to ensure that the listed forces calculations have similar performance in GROMACS and NB-LIB.

In addition, the listed forces implementation is designed such that the same kernels can be reused for the GPU implementation thereby reducing code duplication and increasing maintainability. Currently, GROMACS uses an entirely different implementation to compute the listed forces on the GPU, so new features take longer to be supported on all hardware platforms.

Addressing Challenges of the Exascale

Above we mentioned that another main challenge is that of fully utilizing available compute resources. This problem will only be compounded as exascale machines come increasingly online. The various force calculation APIs exposed by NB-LIB are also a step forward in this area. Current workflows using GROMACS require large amounts of file reading and writing to run multiple calculations of the same molecular system, and it is not possible to share data between multiple calculations in memory, without writing to and subsequently reading from disk. This is a major bottleneck for utilization of large compute resources given that the observables of interest in particle-particle simulations are statistical quantities that are best computed from large numbers of statistically independent samples. As such, a necessary workflow for the exascale era is the ability to simultaneously run very large numbers of calculations sharing system information and computing statistical properties on the fly in memory, allowing for better and also faster sampling. The NB-LIB APIs allow such workflows to be written. In this case, the target user is not performance engineers, but domain experts, who are able to benefit from the performance that NB-LIB exposes without requiring detailed knowledge of the underlying implementation. In short, NB-LIB allows a separation of concerns between performance tuning and scientific workflow generation.

In this context, it is important to mention that NB-LIB is designed to be agnostic of domain decomposition and does not itself provide any routines to perform communication between multiple MPI ranks. Distributing particles across multiple compute nodes is left to the calling code which in turn may call the NB-LIB API on each of these nodes to compute non-bonded forces. In the case of GROMACS, these two concerns are not well separated and therefore, in order to take advantage of the NB-LIB API, the underlying GROMACS implementation requires refactoring that is currently ongoing.

We have put most emphasis on having an intuitive interface that does not impose significant overhead, so the main thing to be demonstrated is that the NB-LIB force calculations are similar in performance to those in GROMACS. This turns out to be challenging for a number of reasons. First, GROMACS is a complete molecular simulation code, while NB-LIB only exposes some of the most critical parts of such a code. In particular, NB-LIB has put little emphasis on the ability to update the positions of particles. We implement a basic update functionality, but have not put much work into performance in this area, and have also not implemented nor exposed a full update functionality that would allow such features as temperature and/or pressure coupling, or constraints of bonds which is an important performance optimization for a full particle-particle simulation. This means that direct comparison of performance between NB-LIB and GROMACS is somewhat challenging. The NB-LIB API is a thin interface with very low start-up cost, while GROMACS has a higher start-up incurred in order to do some moderate hardware detection and targeting. Currently in NB-LIB it is the responsibility of the user to select the best runtime configuration, so there is a trade-off between speed and ease of use.

Performance Data

Being primarily an effort for librarizing key features of a molecular simulations package, one way to evaluate NB-LIB is to perform simulations using standard test cases with it and compare the throughput with GROMACS. This gives an indication of the performance on a single node which one can expect when doing a simulation in a custom novel workflow.

The selected examples are run with the same TPR input file, both with NB-LIB and the main GROMACS simulation engine.

In the field of molecular simulations, a common metric to evaluate real-world performance outcomes is the number of nanoseconds of simulated time per day in wall-clock (ns/day). In this metric, higher is better as it represents real-world throughput. We present the following comparative benchmarks selected to isolate specific features of the library:

- Van der Waals gas simulations on CPU (purely non-bonded interactions)
- Van der Waals gas simulations on GPU (purely non-bonded interactions)

Each of these benchmarks was performed on a single node of the Piz Daint GPU partition. Each node has an Intel Xeon E5-2690 CPU paired with an NVIDIA P100 GPU. The performance data is an average of 5 runs of the benchmarking script. Benchmarks 1 & 2 used a system of 157464 Argon atoms.

The whole MD simulation consists of two steps in every iteration, the first being the force calculations and the second being the integration (or update) step that uses the previously calculated forces. Of these, the force calculation stage is significantly more computationally demanding.

The integrator is not intended to be a part of NB-LIB, but we implemented a simple CPU version for benchmarking purposes. By default, GROMACS also performs the integration step on the host such that we can meaningfully compare the performance with NB-LIB in that case. When passing

a special flag to the executable, GROMACS can also perform the integration step directly on the accelerator, thereby omitting any data transfers between the host and the accelerator. For completeness, we also report the performance for this configuration. NB-LIB also offers a device-side API that may be employed to implement an equivalent simulation, but we do not yet provide an integrator on the GPU to support that use case.

Target, ns/per day	NB-LIB	GROMACS
CPU-only	28.112	23.898
GPU	57.788	56.848
GPU-integration	n/a	71.413

Table 11: Benchmark performance of NB-LIB vs GROMACS using a system of 157464 Argon atoms

Given the clear separation of concerns, and developer-friendly abstractions, it is easier to tune the NB-LIB implementations for higher performance. Strategies like overlapping compute tasks with data movement and calculations on the host can be envisioned for complex problems.

The metrics above should give a potential user an idea of the expected performance on NB-LIB for large systems. Users of NB-LIB can be confident that their programs largely retain the performance advantages of GROMACS and will inherit the newer updates on the backend automatically as it is packaged with the latest upcoming releases.

12.3 Interactions with stakeholders, users, outreach and publications

The NB-LIB project was conceived by core developers of the GROMACS package. This made the GROMACS developer community to be among the key stakeholders. NB-LIB's development exposed functionality in GROMACS that required significant refactoring and co-design with the GROMACS development team. Naturally this resulted in design discussions, code review, and code contributions to the project.

New emerging HPC hardware motivates performance engineering and porting activities within the GROMACS team. Staying on top of these developments and how they may impact the non-bonded calculations happening under the hood, especially on accelerators, was crucial for the translation layer.

NB-LIB primarily consists of two APIs. One side faces the researcher community, which enables description of molecular topology with a user-friendly interface. This API allows one to write simple MD simulation programs with a clear data flow. The second side encapsulates the details of non-bonded calculations behind an interface that uses arrays of elementary data types. The second one is useful for developers who want to develop performant force calculation routines using the NB-LIB abstractions as a building block.

In addition to non-bonded interactions, which are distance cut-off based, NB-LIB also has a modular implementation of particle-specific interactions, which are referred to as listed interactions for this reason. This interface reuses the same kernels as GROMACS, but the interface allows users

to add their own custom potential functions to the simulations. A two-way translation layer is provided to move from NB-LIB's description to GROMACS and vice-versa. The same kernels are readily usable on GPUs as well. This provides another opportunity for reuse within GROMACS which currently uses two entirely different implementations for the listed interactions for use on CPU-only and GPU-accelerated systems. This code adoption and integration is a part of the ongoing co-design efforts for the forthcoming releases of GROMACS.

The BioExcel CoE provided avenues to interact with various users of the GROMACS package and the broader computational biochemistry and biophysics community. Events organized in this umbrella led to fruitful networking opportunities. This included a presentation of the researcher-level API for scripting novel workflows using NB-LIB. This covered examples of Monte Carlo simulations, computing subsets of interactions, swarm simulations among others. This received encouraging responses leading to further discussions about specific use cases.

The pandemic severely limited possibilities for in-person attendance of conferences. However, a poster was presented virtually at PASC21 in July describing the features and benefits of NB-LIB for the research community. This also opened up discussions with potential users looking into methods for running swarms of protein simulations.

Exploring use-cases with the researcher community led to an understanding of the extent of gaps between what simulation packages offer as compared to the complex needs of the research community. Often, we see researchers resorting to fragile concatenations of different packages to achieve their goals. These modifications become obsolete very soon with each new release, while compromising on reproducibility and performance.

Another key area where NB-LIB got some positive attention from the user community was by researchers who do molecular docking, namely the developers of the popular toolkit called HADDOCK (High Ambiguity Driven bio-molecular DOCKing). Docking is an integral part of workflows in various applications, especially drug design where it helps predict the preferred orientation of one molecule to another when bound to form larger complexes. One critical task in this process is model evaluation, where they rank different configurations based on their energies. The current workflow uses a suite of python scripts, and a proprietary commercial software to compute the energies of a specific molecular configuration. NB-LIB can offer a simpler, free and performant way for evaluating energies of various configurations of a molecular topology. Their feedback further prioritized exposing energy calculations in our development features roadmap.

Community engagement also provided useful feedback for our development. For instance, interactions with the OpenFF (Open Force Fields) project gave us concrete constraints on our topology specification. Design discussion with members of the OpenFF project led to a revision of the NB-LIB topology specification so that while also being backwards compatible with GROMACS force field specification, it is also forwards compatible with the novel, generic force fields now under development. This makes NB-LIB well positioned to be broadly interoperable in the particle simulation community. One interesting new direction in terms of particle-particle force fields is the use of machine learning and artificial intelligence to develop potentials. NB-LIB, with its generic and forward facing topology API is well positioned to be able to play an important role in these exciting new developments.

Finally, work is currently underway to write up the results of the NB-LIB project so that they can be published either in scientific software journals or conference proceedings. This way, the important results achieved in NB-LIB will become even more visible to the broader particle simulation community.

12.4 Overall assessment of achievements and future developments

The primary and most important achievement of the NB-LIB project has been showing that it is possible to have a BLAS-like interface to non-bonded force calculation. This is an extremely important result, not just for future GROMACS development, but for the larger particle simulation community. The ability to have an interface that can be constructed with only elementary types has the potential to facilitate future work on particle simulation code interoperability. In addition to pointing the way toward future efforts in terms of interoperability of different codes, the work carried out in NB-LIB has demonstrated that it is possible and desirable to have modular code in the gromacs codebase itself. Repeated design discussions between the NB-LIB team and the core GROMACS developers have taken place. In these discussions, the requirements for completely separating the partitioning of particles into a grid, allowing for parallelization, and the computation of non-bonded forces has been elucidated. Follow-up work to the NB-LIB project, and/or in the scope of general GROMACS development has the possibility to actually enact this separation of concerns, which is one of the key principles of modern software engineering. A more modular codebase helps in increasing the developer community and simplifies feature development and performance engineering.

NB-LIB already allows for GPU acceleration for the non-bonded interactions and OpenMP accelerated listed interactions. The listed forces backend has been designed to allow a GPU port with minimal effort. Once this work would be completed, it would be possible to upstream the NB-LIB listed forces implementation into GROMACS, and repeated discussions with GROMACS developers has indicated that this would be a welcome addition to the GROMACS codebase.

A further area where close interaction with the GROMACS developer community has resulted in increased modularity within GROMACS is in the area of non-bonded free energy calculations, which is vital for fields such as drug discovery and development. After close interaction between the NB-LIB and GROMACS developers, it was determined that a generic interface for non-bonded free energy calculations could be directly exposed within GROMACS. This interface, while not yet a part of the NB-LIB API, could be called directly from the same types of scripts that NB-LIB users would write, and adding this functionality to the NB-LIB API would be a natural extension to pursue.

One important result of the efforts in the NB-LIB project, is that now it is possible to perform "embarrassingly parallel" simulations that use multiple MPI ranks to do independent simulations like in Monte-Carlo workflows without having to read and write files on disk, which is the current standard workflow when using the GROMACS binaries. However, an important future direction is the work necessary to expose multi-node force calculations as they exist within GROMACS. Currently multi-node simulations are possible but only with orchestration by the user. While this is useful in cases where the desire is to interface with other molecular simulation codes, further work within the GROMACS project is needed to separate partitioning from non-bonded force calculation and expose automatic partitioning. The work in the NB-LIB project allowed the requirements for such a decoupling to be elucidated and co-design between NB-LIB and GROMACS developers has led to a general agreement on the direction needed to make this a reality.

NB-LIB supports reading full molecular system descriptions from GROMACS input files. This allows for more flexibility with using NB-LIB in custom user code because users can either write their own system description or use existing GROMACS machinery to produce a system description. This is especially helpful in cases where the interest is performance tuning for

particular hardware and/or systems, as it is possible to directly set to work on performance aspects without having to write lots of setup code. Since the NB-LIB APIs are generic, it was little work to expose the functionality of reading GROMACS input files. This also suggests that future work could add reading input files from other popular simulation codes. Another potential future direction that NB-LIB could facilitate would be the ability to read in system descriptions from GROMACS, or other MD codes', input files and then edit the system and/or interaction types. Since system setup is one of the most challenging parts of molecular simulations, this would likely be a very welcome functionality.

Future development goals include interfacing with the long-range interactions using methods such as PME that are already built into GROMACS. This would expand the use-cases for NB-LIB further. Our abstractions that cleanly split the concerns of various force calculators would allow easy integration of new methods, such as the Fast Multipole Method which the research community is keenly exploring due to its promise of higher scalability. Indeed, with the increasing amount of accelerator compute capacity per node in HPC systems, codes that do not use FMM for long-range interactions will have an increasingly difficult time fully utilizing available hardware.

NB-LIB has demonstrated that librarizing of an MD package is very much possible and the process leads to more modularity within the parent codebase. It also leads to tangible benefits for the researchers whose novel workflows can readily benefit from the performance improvements that developers work so hard to expose.

13 Conclusions

Significant investments are currently being made in Europe to provide pre-exascale and exascale computational resources to the research communities. As already argued in the PRACE position paper on 'Software Strategy for European Exascale Systems' [82], the return on investment will be directly linked to the productivity of end-users in academia, in industry, and in the public sector. Key to this productivity is an ecosystem of user-oriented software: scientific applications and workflows that act as significant multipliers for the investment in hardware. Investments in software should be a top priority of any HPC strategy. The extent to which these investments are needed and their real impact are often underestimated.

The work in WP8 is an important pillar in PRACE's software strategy, as it allowed for significant, strategic, and long term investments in software, with a direct impact on the scientific software deployed on Tier-0 systems in Europe. Indeed, WP8 has been able to deliver across 10 projects, several of which involved multiple partners across Europe, open source software, of high quality and ready for deployment on large scale infrastructures. This is evident in Table 1 and Table 2 that demonstrate the deployment on, or readiness for, the top systems in Europe, that are already installed or, in the case of pre-exascale systems, will be online in the near future. Several of the projects (see also Annex A), have been able to make codes ready for the new architectures that have become available for scientific computing (notably AMD GPUs, which, at the time of writing, power the two machines Europe has in the top 10 on the top500 list [83]). Similarly, projects report on speedups and improved scalability on other available systems. Rather than list all of these here, some examples can be highlighted which include the DLA-Future eigensolver outperforming SLATE and DPLASMA on the PRACE Tier-0 supercomputer Piz Daint, the refactored version of EPOCH ~30-45% faster on the PRACE Tier-0 supercomputer JUWELS than the original version, and the integration of GHEX into the atmosphere simulation BIFROST leading to ~20% improvement in performance on the CPU partition of the LUMI EuroHPC pre-exascale system.

In addition to these common metrics of efficiency, it is noteworthy that these projects also increase the HPC software engineering expertise, with collaborations between software engineers of various institutions, as well as pushing the approaches to software engineering adopted in HPC. Examples of this are the use of containers for deployment, the adoption of new programming languages such as Julia, the use of message passing formalisms beyond MPI, the use of advanced tasking mechanisms in line with the C++ language standards, the adoption of CI/CD frameworks and high levels of unit and coverage testing. It must be emphasised that most of the developments performed in WP8 have found their way in upstream packages or are already being directly used by scientists. Additionally, there have also been a number of productive interactions between the WP8 projects and other actors in the European landscape, one notable example being the Centres of Excellence.

Finally, as part of this last deliverable for WP8, it is worthwhile to look back at what worked well and what can be improved in the future for similar projects. As such, firstly the two challenges that have had a negative impact on the project, with the most visible of these being the global COVID-19 pandemic. This made it significantly more difficult for the teams to meet in person, for example for hackathons, project meetings, conferences, but also for hiring. Despite this, software engineers have managed remarkably well to shift to online collaboration and to continue the development of complex pieces of software. Training the future workforce how to collaborate remotely, with tools and techniques, will definitely contribute to the resilience of the society. A second challenge has been the delayed installation of the pre-exascale machines, as WP8 was aligned with the original plan to start operation before the end of 2020. The projects adjusted by shifting to testing on

existing Tier-0 machines and readying for the to-be-installed architecture, rather than for the particular installed machine. In doing so, projects have found it difficult to access a wide range of machines for testing and development. Whilst all projects had their own resources, or could access other resources via development project proposals, a streamlined process to allow access to Tier-0 or EuroHPC systems would be beneficial for future software development projects to ensure timely and adequate access to the HPC infrastructure. Finally, we highlight two successful aspects of WP8 that are worth repeating in the future. First, is the concept of funding long-term, strategic software engineering efforts, with deployed software as the main deliverable. This perspective and the associated funding have led to developments that cannot be made incrementally. Second, comes the idea of organizing these developments based on a competitive call within the project. The resulting projects had well defined targets and roadmaps, and could start to work on their goals very quickly and efficiently, and were continued at the end of this project or taken up by users on HPC systems in Europe in the majority of cases.

Annex A: Benchmarking and performance results obtained on leading HPC systems during WP8 extension

A.1 PiCKeX: Particle Kinetic codes for Exascale plasma simulation

In the PiCKeX project, we have enabled substantial refactoring work of hybrid Particle-in Cell/Monte-Carlo (PIC/MC) codes and tested ways to add support for multiple GPUs. The aim has been to work this code in a fully scalable GPU version. In the extension period, we have focused on producing a multiple GPU prototype PIC code, with a view to future implementation in OOPD1. The current GPU optimisation of the OOPD1 code is working on a single GPU. We generate all particles on the GPU to avoid the memory transfers. When the new particle positions are calculated, all GPU outputs are copied back to the CPU version to start the field solver. For this reason, we call the new version semi-optimised GPU code.

OOPD1: Objected Oriented Plasma Device 1D is a particle-in-cell code with a Monte Carlo algorithm for the calculation of particle interactions and collisions. The code was run on the VIZ supercomputer at the Faculty of Mechanical Engineering in Ljubljana, the Marconi M100 supercomputer at CINECA, and on the EuroHPC MeluXina supercomputer.

For further optimisation, we have implemented a prototype PIC code for testing all GPU optimisation steps. The previous prototype PIC code written in OpenMP was rewritten in OpenACC and benchmarked, see Figure 60 below, proving that the code has better scaling with OpenACC. In addition, using OpenACC is beneficial in terms of future deployment, as although we are presently running the code on NVIDIA GPU systems, in the future we plan to test the OOPD1 code on the LUMI supercomputer which uses AMD GPUs.

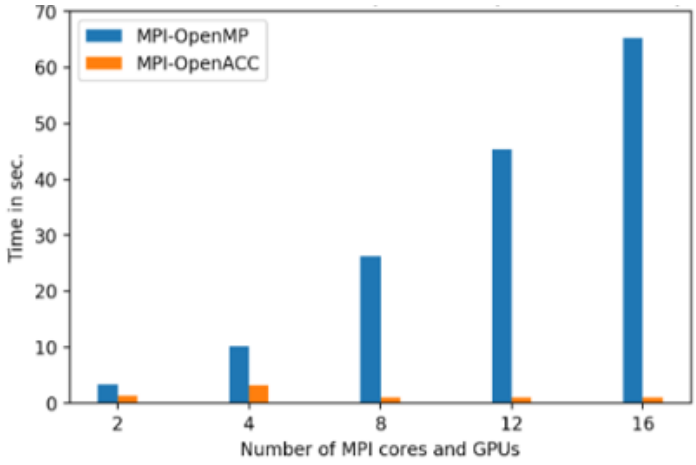


Figure 61: MPI communication time for 100,000 particles MPI-OpenMP vs MPI-OpenACC

The new version of the prototype PIC also works with multiple GPUs. We ran the same benchmarks cases comparing single with multiple GPUs, as shown in Figure 61, with 100,000 particles per GPU. However, it should be noted that whilst there is weak scaling, the prototype PIC code is not as complex as OOPD1, for which different behaviour might be expected.

It should also be noted that whilst Figure 60 above is a comparison of MPI communication time for OpenMP vs OpenACC, Figure 61 below refers to total time in the particle mover for single vs multiple GPUs.

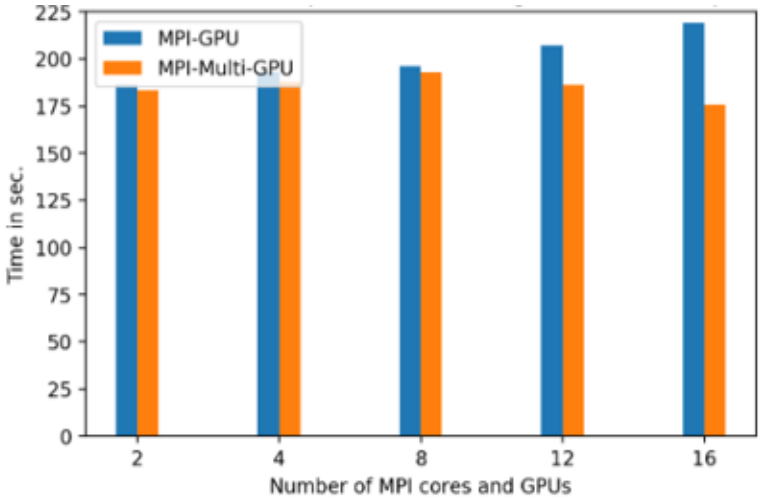


Figure 62: Particle mover time for 100,000 particles single GPU vs multiple GPU

Finally, it should be added that further developments are planned to improve the OOPD1 code with a GPU version of the field solver, using the same set of tests done on the prototype PIC code for OpenACC and using the multi-GPU implementation of OOPD1.

A.2 MoPHA: Modernisation of Plasma Physics Simulation Codes for Heterogeneous Exascale Architectures

In the MoPHA project, we have explored task-based parallelism for plasma simulations and tested ways to add support for GPUs or other accelerators to plasma simulation codes, targeting the three codes ELMFIRE, GENE and Vlasiator. The aim has been to work towards making plasma simulation codes ready for the upcoming pre-exascale and exascale systems. In the extension period, the focus has been on benchmarking selected codes on the LUMI pre-exascale supercomputer and continuing related porting efforts. Results from the CPU partition of LUMI, called LUMI-C, are included below. LUMI-C consists of 1536 nodes each with two 64-core AMD EPYC 7763 CPUs.

Vlasiator: Two benchmarks cases (one in 2D with 2.3 10^{11} phase-space cells and one in 3D with 4.9 10^{11} phase-space cells), were run on LUMI-C to test the scalability of Vlasiator on the system. As can be seen from the results shown in Table 12 and Figure 62, for the 2D benchmark Vlasiator scales well up to 300 nodes (38400 cores) and for the 3D benchmark Vlasiator shows reasonable scaling all the way until the maximum number of nodes tested (1452 nodes, 185856 cores).

Nodes	Cores	2D total time (s)	3D propagation time (s)
100	12800	273.7	6079
200	25600	142.7	4055
300	38400	106.0	3752
600	76800	83.8	3060
900	115200	81.5	2458
1452	185856	91.5	1855

Table 12: Vlasiator scalability on LUMI-C

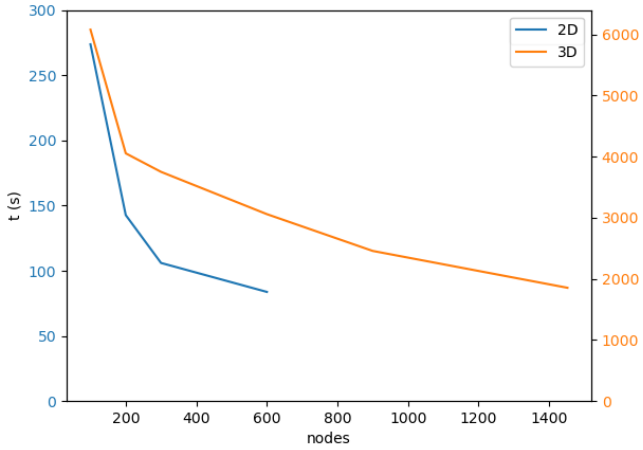


Figure 63: Vlasiator scaling on LUMI-C

StruGePiC / SymPiFE-VMax: Two mini-apps, StruGePiC and SymPiFE-VMax, were also ported to LUMI-C. Both mini-apps simulate plasma fusion systems with the same Vlasov-Maxwell equations for full-orbit (6D) charged particles. StruGePiC is implemented using the AMReX framework and SymPiFE-VMax using the MFEM framework.

As can be seen from Figure 63 below, on LUMI-C StruGePiC has good scaling up to 128 nodes at which point the performance deteriorates.

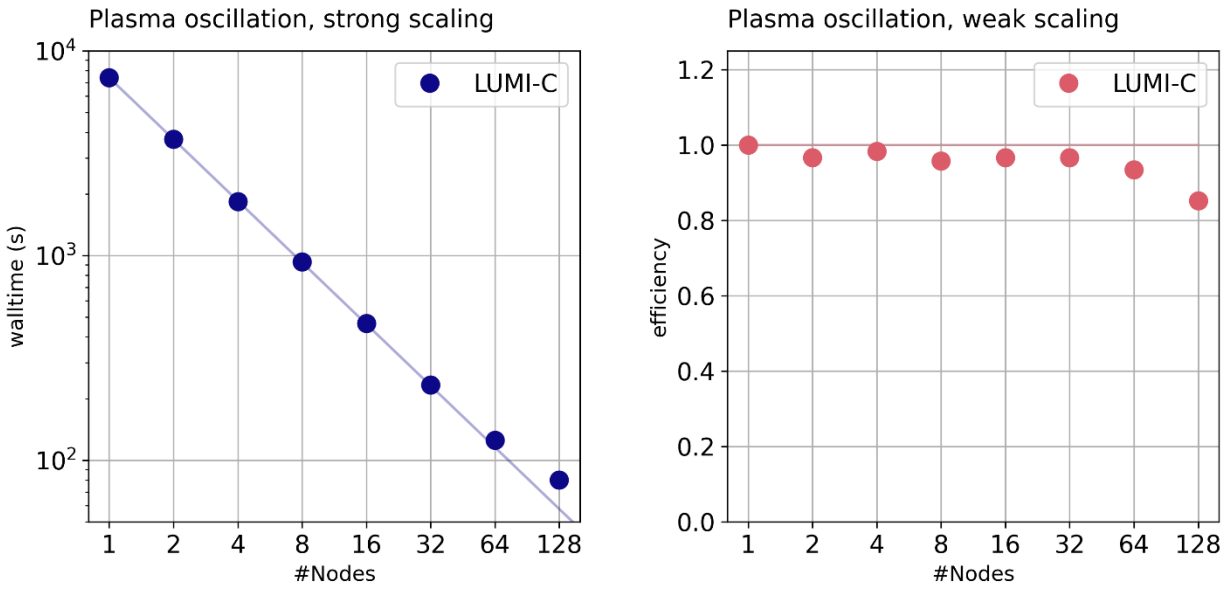


Figure 64: Scaling of StruGePiC on LUMI-C

All in all, StruGePiC showed excellent performance and portability, but is currently limited (in the field of fusion) to more fundamental applications due to the choice of Cartesian domains in AMReX. Conversely, SymPiFE-VMax is still facing performance challenges, but already possesses a number of features that can enable its use in simulating fusion devices. The performance issues of SymPiFE-VMax, which are related to features not included in MFEM, are being addressed at the moment. Due to the excellent scalability of MFEM, we expect that when these issues have been resolved properly, SymPiFE-VMax will have comparable performances to STruGePiC. Further developments are planned both to improve performance and scaling, and to add features expanding the domain of applicability of the simulation (such as collisions or relevant boundary conditions). A GPU-capable prototype for a structure-preserving collisions solver has been developed by the ELMFIRE team at Aalto University and is being integrated to the applications.

A.3 Performance portable linear algebra

The activities on DLA-Future carried out during the extension can be grouped in three main groups:

- development of missing algorithms,
- porting to AMD GPUs,
- benchmarks.

For the algorithm development the focus was on finishing the implementation of the single node generalized eigensolver. The missing single node algorithms have been implemented, and the single node generalized eigensolver pipeline is available for multicore and GPU architectures. To prove the benefits of overlapping, we compared the execution of the algorithms of the eigensolver one by one with the execution of the full overlapped pipeline in Figure 64 below. The difference is visible in the traces, however it is also visible that some of the algorithms are not well optimized yet and suffer from the micro-tasking problem.

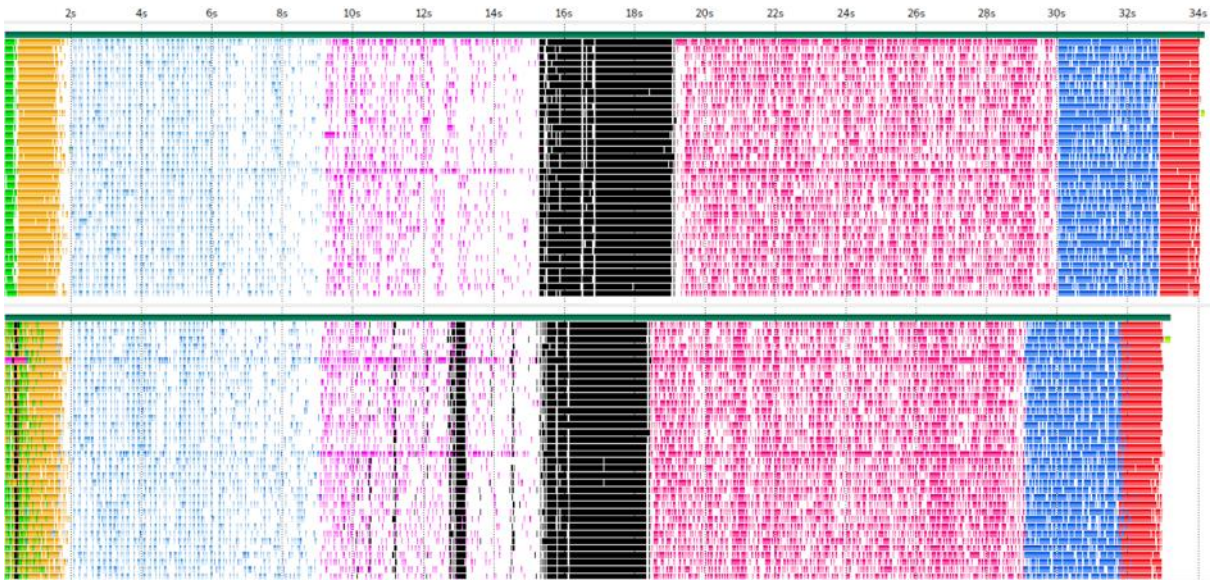


Figure 65: The trace of the execution of a generalized eigensolver for a 10240×10240 matrix. Above the algorithms are executed sequentially, below they are allowed to overlap.

The task environment introduces overheads in the execution. In particular queue handling, stack creation, and context switching, contribute to a 1-10 μs overhead in HPX. Therefore, the minimum task size should be set to 1 ms to make the overhead negligible. Combining multiple small tasks in a larger task helps to solve this problem, however it can limit the parallelism achievable as extra dependencies are introduced.

Support for AMD GPU has been added. We tested the library on LUMI's GPU Early Access Partition (64-core AMD CPU with 4 MI100 GPUs). Due to the limited number of nodes available it has not been possible to carry out a full scaling analysis (jobs are limited to maximum two nodes). However, as Figure 65 below shows, the results are encouraging.

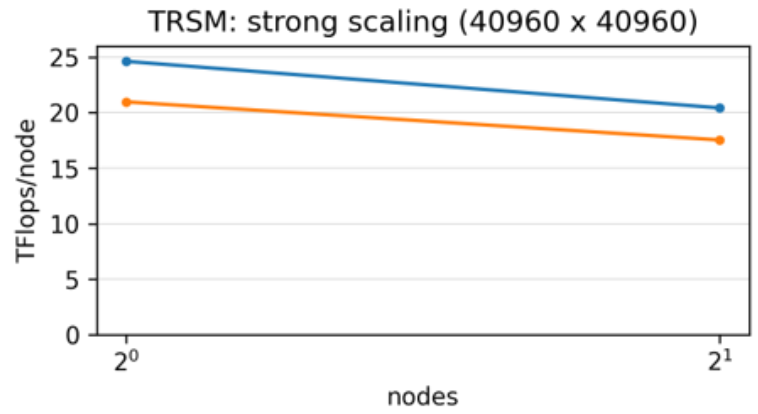


Figure 66: Triangular solver on LUMI-EAP. We present the strong scaling for a matrix of size 40k.

On the other hand, results of Cholesky decomposition are not as promising. On a single node the performance is 7 TFlop/s for a 40960x40960 matrix and 3 TFlop/s for a 20480x20480 matrix, which is only a small fraction of the theoretical peak performance of 4 MI100 GPUs (46 TFlop/s). We identified the cause in the rocSOLVER library, in particular in the xPOTRF implementation. As this kernel is used in a task in a critical path of the DAG it has a great impact on the whole execution time.

We also tested on a test cluster which is part of the Alps system (a HPE Cray Ex) at CSCS. The nodes are composed of a 64-core AMD CPU (EPYC 7713) and 4 NVIDIA Ampere GPUs with 95 GiB on HBM memory and 7936 CUDA cores, with HPE Slingshot 10 interconnect. As Figure 66, Figure 67 and Figure 68 show, the results are promising.

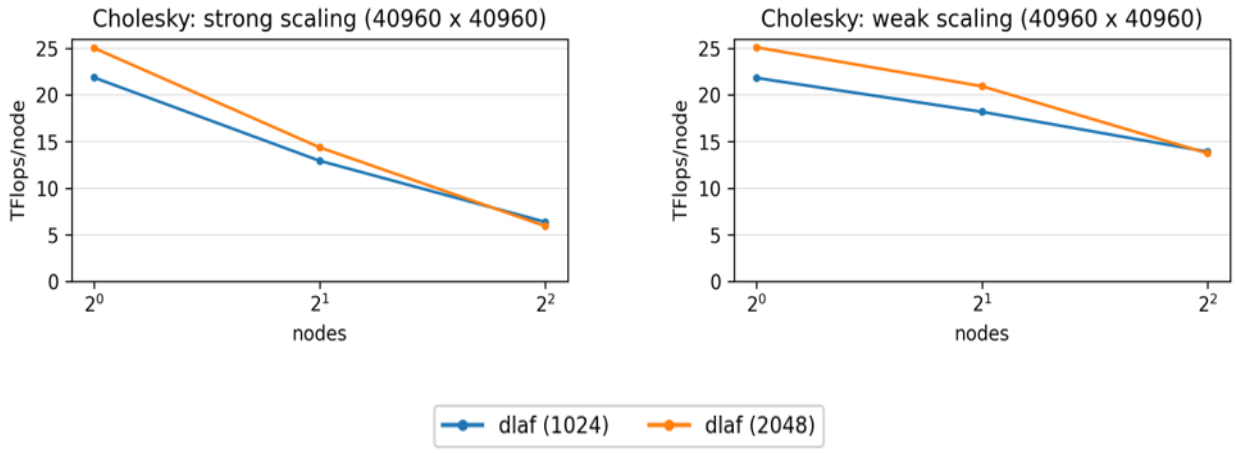


Figure 67: Cholesky decomposition on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node).

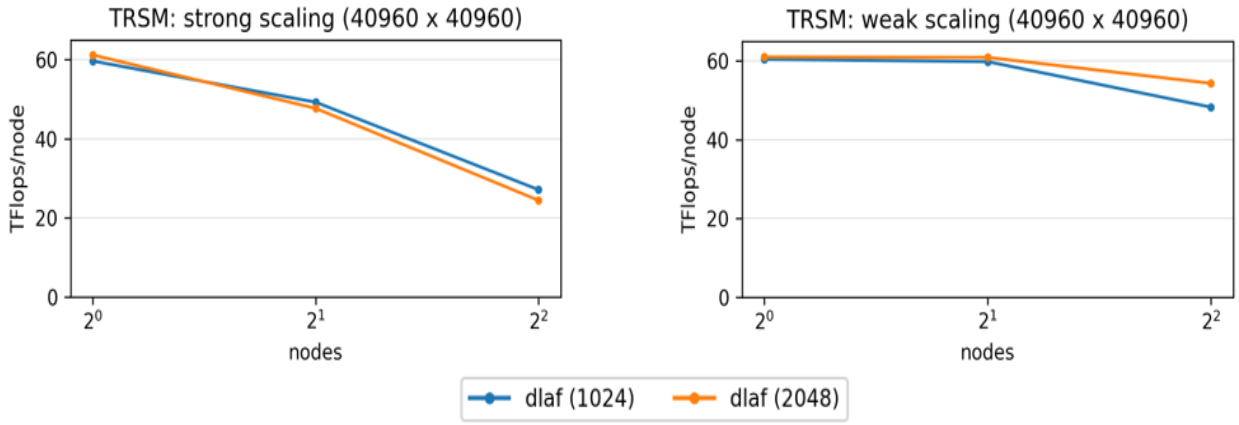


Figure 68: Triangular solver on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node).

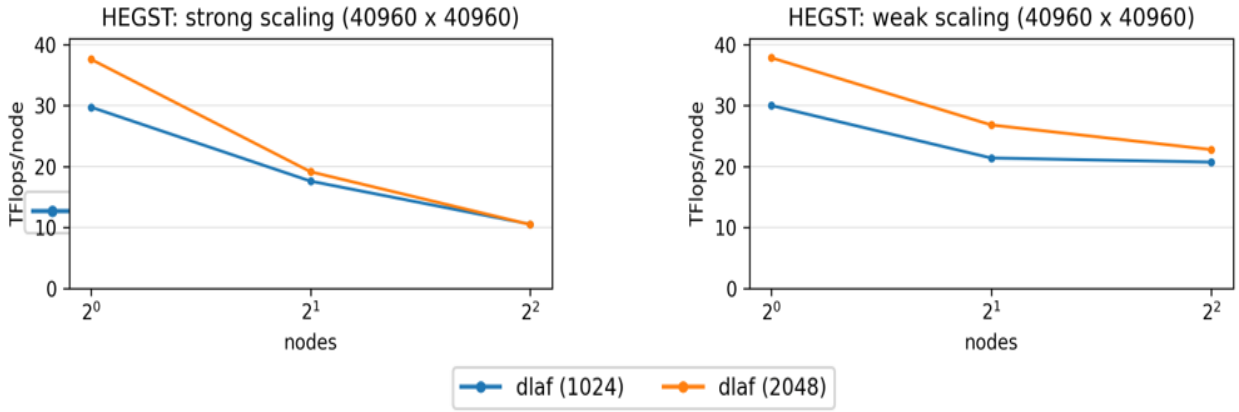


Figure 69: Transformation from generalized to standard eigenproblem on Ampere GPUs. Left: we present the strong scaling for a matrix of size 40k. Right: we present the weak scaling for 1.6G elements per node (40k x 40k matrix for the run on a single node).

A.4 LyNcs: Linear Algebra, Krylov methods, and multi-grid API and library support for the discovery of new physics

During the extension of PRACE-6IP, we continued the developments of the software stack with a focus on lyncs-API and librsb. Moreover, we benchmarked several kernels, developed within or related to the project, on EuroHPC-JU machines and corresponding architecture. This includes strong scaling tests on Karolina as well as performance tuning on novel CPU architecture and performance checks on AMD Instinct Mi100 GPUs.

DDalphaAMG: In addition to the earlier performance results reported for DDalphaAMG-rhs, which are mainly obtained on the PRACE Tier-0 machines, we used a preparatory access on the CPU part of the EuroHPC petascale system Karolina to collect results for AMD Epyc CPUs. Using a configuration of size V=128x64x64x64 generated at light quark masses tuned to the physical pion mass, we perform a strong scaling test from 16 nodes to 128 nodes for different numbers of right hand sides (rhs). We found comparable improvements obtained on HPC machines equipped with Intel Xeon CPUs, such as the LRZ system SuperMUC-NG.

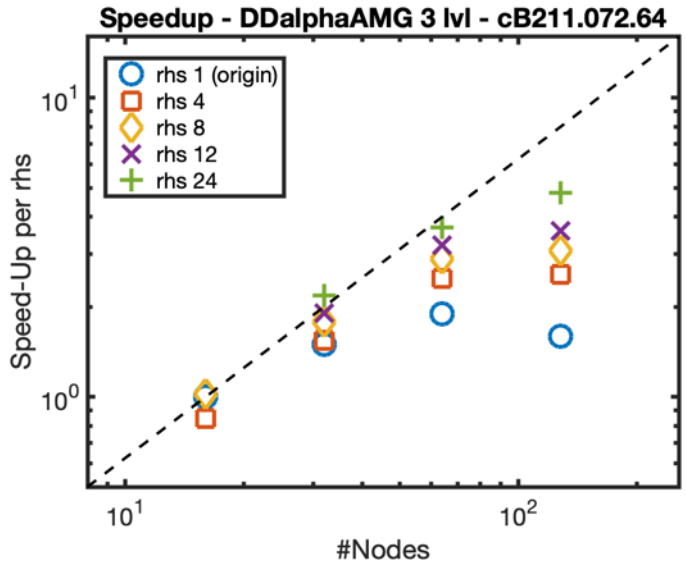


Figure 70: Strong scalability of DDalphaAMG-multiple rhs on IT4I system Karolina

Lyncs-API: Within LyNcs we developed a novel python API with a focus on achieving portability. This is done through linkages to different software packages each optimized for different architectures. A major focus during the extension was given to the module lyncs-quda which provides a python interface to highly optimized linear solvers and computational kernels for lattice QCD on NVIDIA GPUs implemented in the software package Quda. In the last months we have increased the coverage of the python API and enlarged the test suite. A major component still missing to interface is the multigrid solver provided by Quda which we will focus on in the next months.

Moreover via the lyncs.quda interface it is straightforward to utilize highly optimized computational kernels within Machine Learning applications, e.g. within novel update algorithms.

This provides researchers an easy-to-use tool to develop machine learning applications within a python framework, such as PyTorch or TensorFlow for lattice QCD. In this direction we are collaborating with researchers at TU-Berlin and DESY-Zeuthen on the application of machine learning techniques to Lattice QCD simulations.

Additionally, for most of Quda there exists a HIP port, due to the DoE Exascale project. This potentially gives lyncs a possibility to utilize AMD Instinct GPU architectures. We tested this HIP-ported QUDA kernel on the partition of CSC early access system LUMI-EAP based on 4 nodes of Mi100 and collected performance numbers on up to 8 Mi100 accelerator cards. Performance results of the Wilson dslash operator applications are comparable to numbers obtained on NVIDIA's V100. However, lyncs support for AMD GPUs will depend also on underlaying python modules for which HIP/rocm support is necessary. This will require efforts beyond PRACE-6IP. Lyncs-API is currently the only lattice QCD python interface which provides access to Quda kernels. This will help to further support the development and we expect a larger engagement by the community in the future.

librsb: During the extension, a new version librsb 1.3 was released, which comes with new features, such as multiple rhs support and performance improvements for the majority of the supported kernels compared to the pre lyncs version librsb 1.2. Using the updated optimiser, performance tuning was carried out on different architectures, including AMD Epyc and Fujitsu A64fx. As can be seen in Figure 70 below, this has resulted in improved performance in the latest version of librsb for sparse matrix computations across various HPC architectures including BEAST and SuperMUC-NG systems.

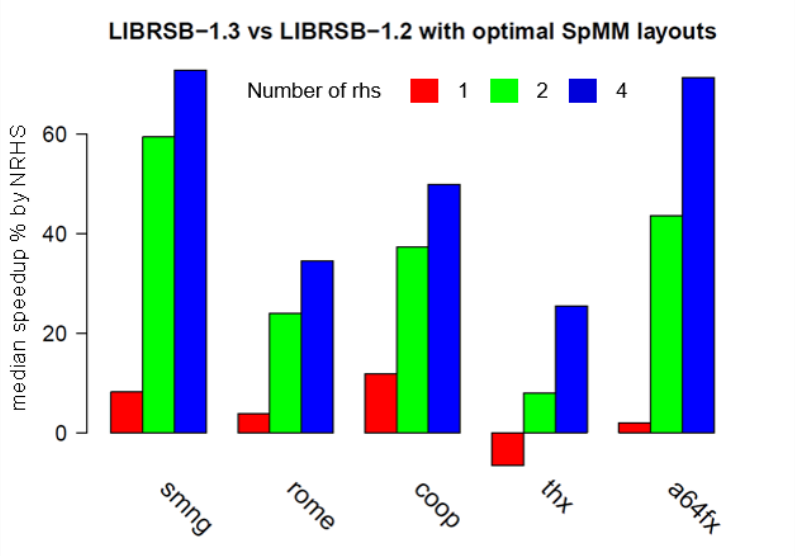


Figure 71: librsb 1.3 vs librsb 1.2

A.5 QuantEx: Efficient Quantum Circuit Simulation on Exascale Systems

The QuantEx code simulates quantum circuits. To this end, the core computational problem executed is the computation of probability amplitudes of possible measurement outcomes of a quantum circuit. This is achieved by representing the output quantum state of the circuit by a tensor network and then contracting the tensor network to produce the desired probability amplitude.

Within the QuantEx code, QXTools can be used to set up a simulation of a particular quantum circuit and generate a set of simulation files which define the simulation as a sequence of tensor contraction operations. QXContexts can then be used to parse the simulation files and distribute the contraction operations amongst compute resources to produce the relevant output.

Weak and strong scaling experiments were carried out on the JUWELS Booster system. The behaviour of QXContexts is shown below in Figure 71 and Figure 72. As test cases, a number of probability amplitudes were computed for various quantum circuits developed by Google. QASM descriptions of the circuits simulated can be found on the qflex repository [84]. The figures show our application achieves close to ideal scaling when simulations are decomposed across multiple compute nodes.

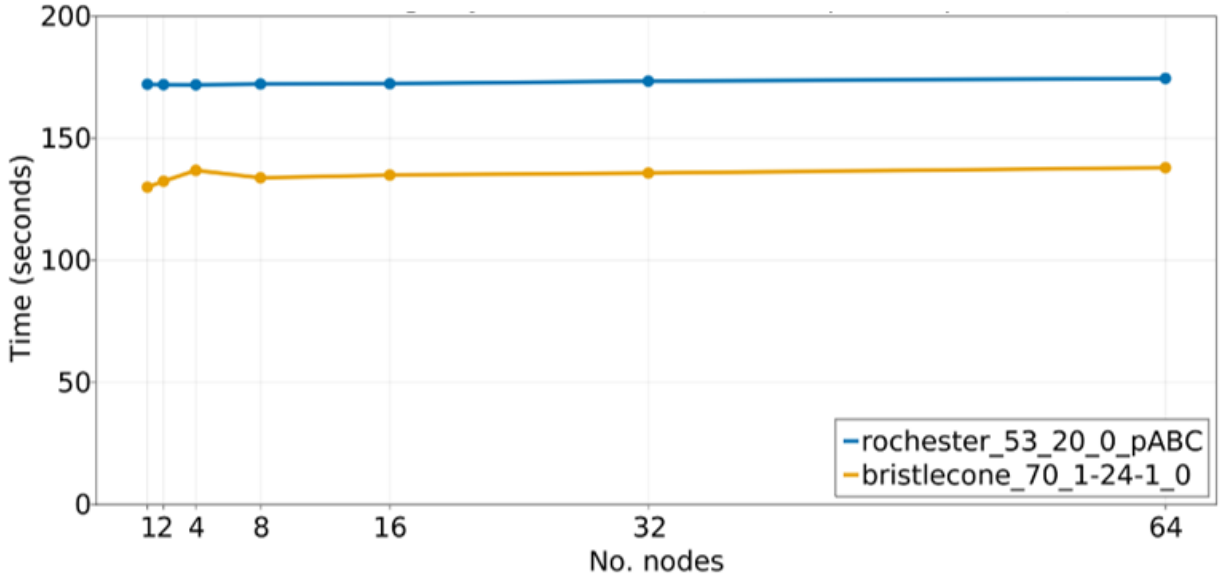


Figure 72: Weak scaling of a QXContexts simulation on JUWELS Booster. As test cases, 1024 probability amplitudes per node were computed for both a 53 qubit rochester circuit and a 70 qubit bristlecone circuit.

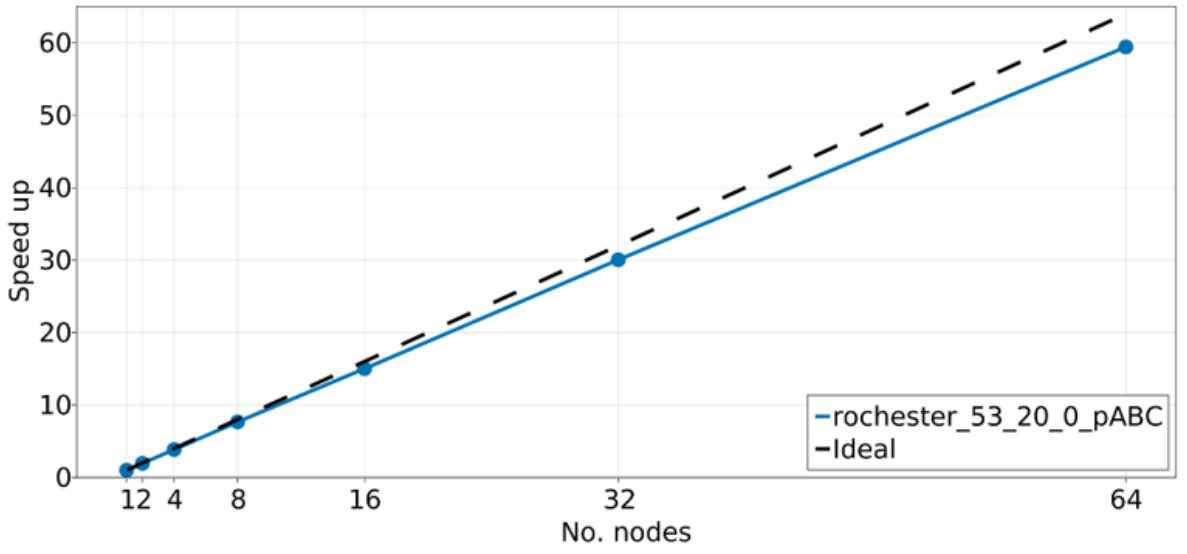


Figure 73: Strong scaling of a QXContexts simulation on JUWELS Booster. A total of 65536 probability amplitudes of a 53 qubit rochester circuit were computed as a test case.

However, it should be noted that our application initially exhibited poor single-node scaling results as can be seen in Figure 73 below.

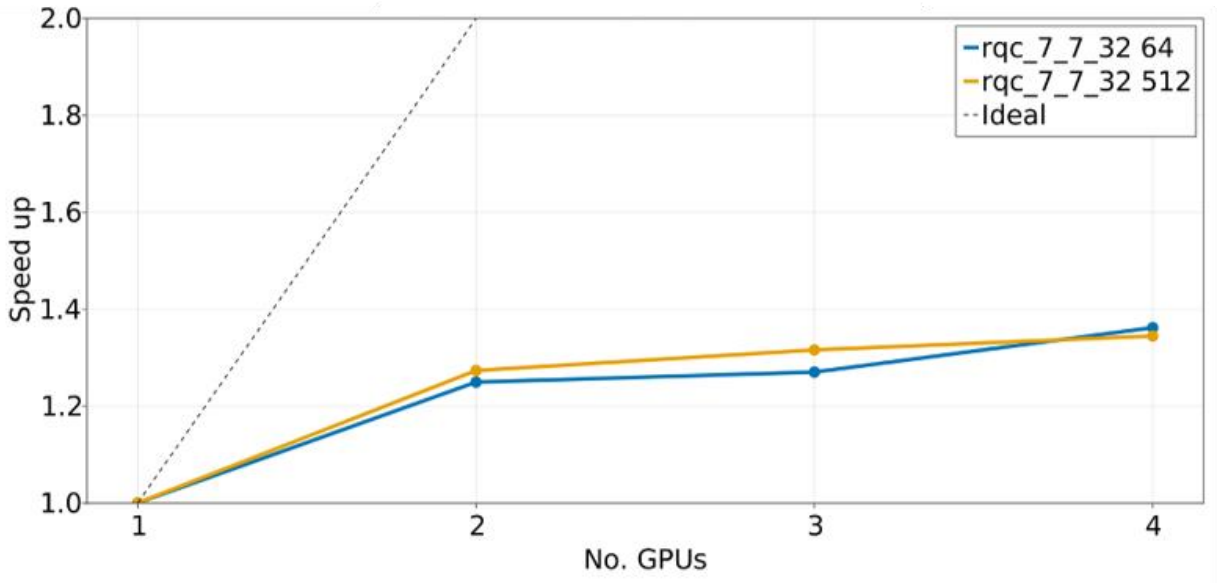


Figure 74: Strong scaling of a simulation within a single node on JUWELS Booster. As test cases, both 64 and 512 probability amplitudes were computed for a 49 qubit random quantum circuit with 32 layers of entangling gates.

After consulting with the assigned mentor for the project, we identified the issue as a failure to use multiple cores to execute the processes spawned by our application, resulting in all processes running on a single core. With the aid of our assigned mentor we were able to rectify this issue, assign each process to its own CPU core, and obtained close to ideal scaling on a single node as

shown in Figure 74 below. It should be noted that the results do not depend on the particular circuit simulated, the circuit simulation shown below in Figure 74 is different from the simulation shown in Figure 73.

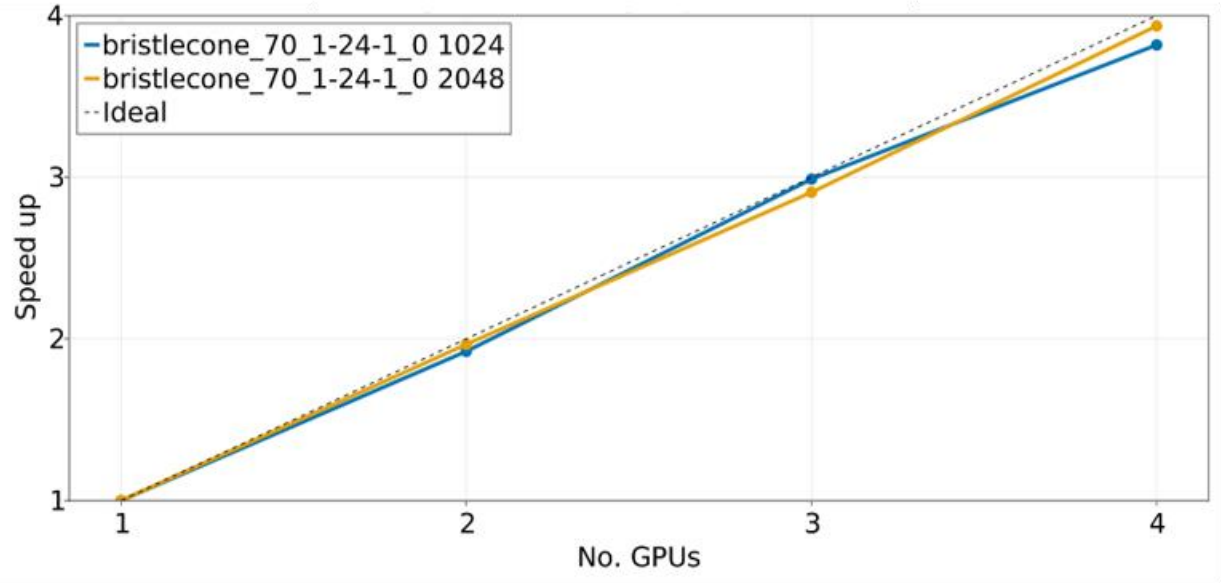


Figure 75: Strong scaling results for a simulation of a 70 qubit circuit on a single node on JUWELS Booster. Results are shown for both the computation of 1024 and 2048 probability amplitudes. We observe close to ideal scaling with respect to the number of GPUs used on a single node.

A.6 GHEX: Generic Halo-Exchange for Exascale

GHEX is a library for halo-update in scientific applications capable of interfacing with arbitrary applications regardless of domain decomposition organisation, grid/mesh type, memory layout, and halo description. GHEX achieves this by providing interfaces that accept, instead of pointers to data, pointers to functions to retrieve the relevant information. The extension work focused on extending the range of machines where GHEX can be supported as well as testing applications using GHEX. In particular, we ran GHEX basic transport-layer benchmarks on LUMI-C. The results, shown in Figure 75 below, show the raw bandwidth achievable with multithreaded executions in MPI, UCX and Libfabric (labelled as OFI). These benchmarks demonstrate how performance varies depending on the use-case (number of thread, message sizes, etc.), which makes the ability to switch transport layer at execution time very valuable.

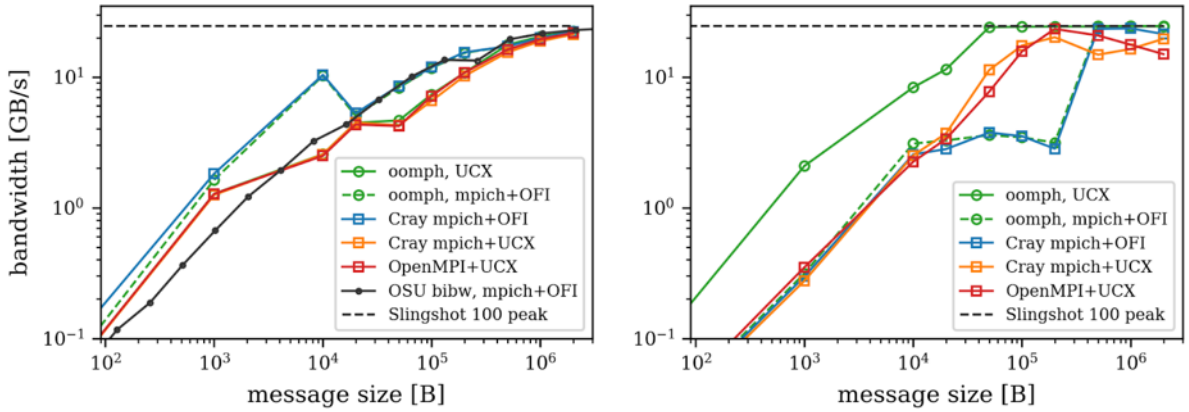


Figure 76: Bi-directional communication bandwidth on LUMI-C (Slingshot 10, 100Gb/s)for messages of different sizes. Left: 1 thread, 1 message in-flight. Right: 16 threads, 10 messages in-flight

We also tested GHEX on the IBM Power architecture of Marconi100, for which we were able to run the GHEX unit and integration test suite, as well as a simple set of benchmarks.

When we run halo-exchange benchmarks at larger scale, we can see that GHEX offers good weak-scalability, while the choice of how to arrange computing ranks (e.g. using a custom-built hardware-aware Cartesian MPI communicator - HWCART, also available at the GHEX GitHub repository) and transport strategies (e.g. using inter-process communication via XPMEM) offers opportunities for improving performance.

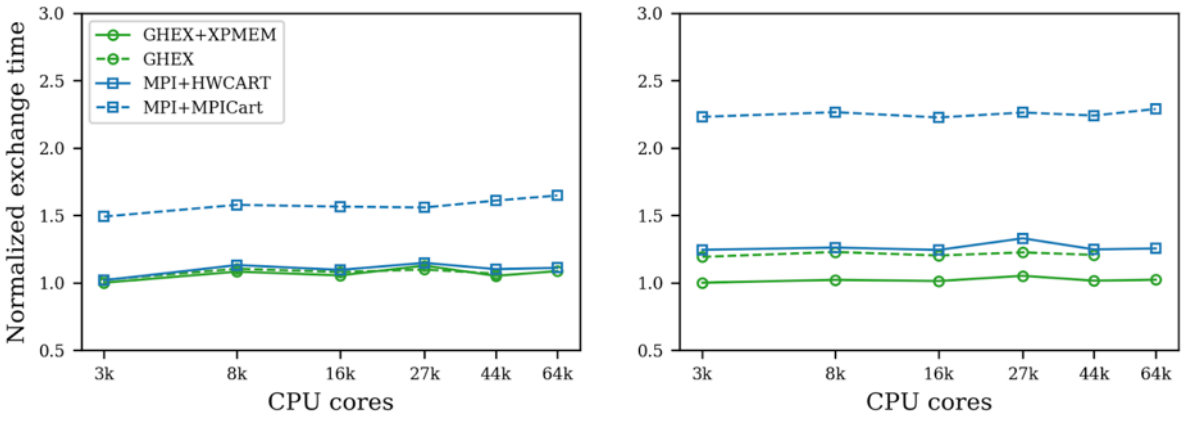


Figure 77: Weak scaling of HE with GHEX and native MPI on LUMI-C. Each rank handles double-precision fields of size 128^3. Left: 1 data field, halo width 1. Right: 5 data fields, halo width 5.

We also tested applications adapted to use GHEX. In Figure 77 below we show BIFROST, a solar atmosphere application developed in Fortran, that uses GHEX Fortran bindings. The executions have been carried out on Betzy, an AMD system at University of Oslo, and on LUMI-C. Interestingly, the GHEX version of the application is capable of out-performing the original version by a large margin.

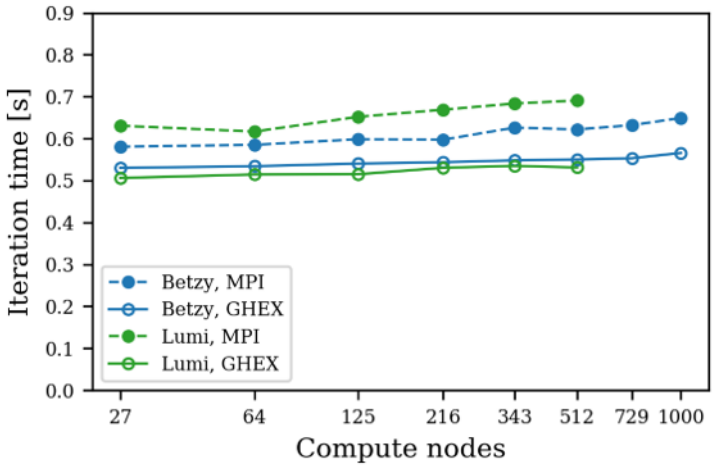


Figure 78: Weak scaling of BIFROST with GHEX and native MPI halo exchange on Betzy and LUMI-C. Single-precision, 64^3 grid points per rank

A.7 ParSec: Parallel Adaptive Refinement for Simulations on Exascale Computers

The Cenaero and ULiège activities carried out during the ParSec extension focused on the following activities:

- Porting of the mesh libraries Gmsh and MAdLib and the flow solver Argo on the LUMI-C system at CSC and installation of dependencies;
- Ongoing improvement of scalability in various parts of the code, including blocking initialization issue;
- Strong scaling tests up to 512 nodes/65536 cores;
- Preparation of two variants of the smooth backward facing step test case for production runs on 256 LUMI-C nodes, in support of PRACE and EuroHPC allocation requests.

The strong scalability study for the Lagrange-based Discontinuous Galerkin solver Argo and the mesh library MAdLib on the LUMI-C system at CSC is now briefly presented. The large-scale parallelism is based on the spatial partitioning of the computational domain. This study has been performed with an implicit time integration scheme. Computations use a Jacobian-free Newton-Krylov nonlinear solver with block-Jacobi preconditioning. The test case considered here is the Taylor-Green vortex (TGV) which ran on a structured cube mesh of 1923 vertices in each direction (corresponding to 7M hexahedra) with a 3rd order polynomial approximation (P3), corresponding to a 4th order solution with 0.45 billion dofs/eq.

The results of this strong scaling study performed on the system LUMI-C at CSC are presented in Table 13 and Figure 78. These results are compared to those obtained on the Hawk system at HLRS, which is based on a similar architecture, although it should be noted that LUMI-C is equipped with a slightly higher-clocked and newer generation of AMD CPU. The baseline case ran on 4096 cores. The lower bound is simply limited by the required memory of the test-case. Four configurations of the solver are considered with 1, 2, and 4 threads per MPI process on both systems, leading to a total of 128 threads used per node. Perfect or even super scaling is observed up to 65,536 cores on LUMI-C (4 core doublings), with only ~100 elements per thread on the largest core count. On Hawk, the speedup is still 13.83 out of 16 for 4 threads per MPI process on 65,536 cores, corresponding to an efficiency of 0.87.

AMD Rome LUMI-C @ CSC					AMD Rome Hawk @ HLRS			
# cores	**absolute timing (s)**	**speedup**	**efficiency**		**# cores**	**absolute timing (s)**	**speedup**	**efficiency**
4096	99.22	1	1		4096	104.11	1	1
8192	48.65	2.04	1.02		8192	50.71	2.05	1.03
16,384	23.53	4.22	1.05		16,384	24.62	4.23	1.06
32,768	11.96	8.30	1.04		32,768	12.55	8.24	1.03
65,536	6.26	15.85	0.99		65,536	7.50	13.83	0.87

Table 13: Strong scaling of Argo on AMD Rome architecture with 4 threads per MPI and 32 MPI per node: LUMI-C@CSC (left) and Hawk@HLRS (right).

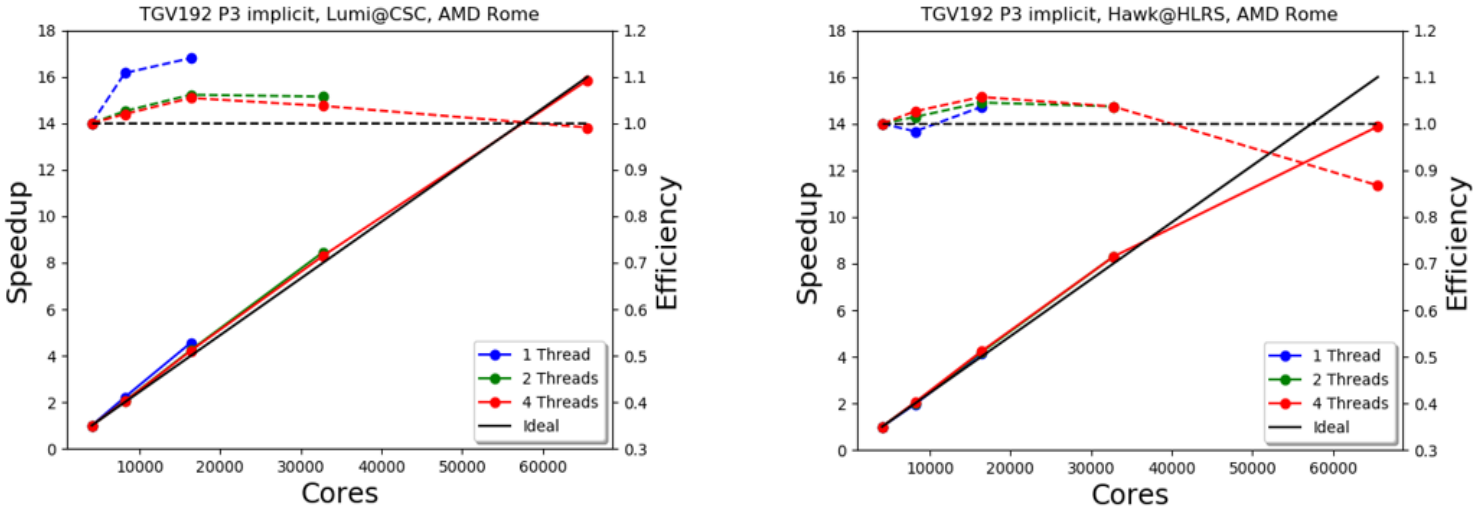


Figure 79: Strong scaling of Argo on LUMI-C@CSC (left) and Hawk@HLRS (right), it should be noted that the dashed lines refer to efficiency

A.8 NB-LIB: Performance portable library for N-body force calculations at the Exascale

The goal at the outset of the NonBonded-LIBrary (NB-LIB) project was to make the cutting-edge performance of GROMACS available through a high-level C++ API to its non-bonded force kernels. This goal has been achieved while also showing that it is possible to have a more BLAS-like interface for non-bonded force calculations. The NB-LIB API allows users to compute non-bonded forces on both CPU and NVIDIA GPUs. In order to provide even more performance and utility to users, NB-LIB has also exposed an API for other types of particle-particle calculations besides non-bonded forces, such as bonds or other types of two particle interactions, angles or other types of three particle interactions, and so on for four and five particle interactions. These interactions are known as listed forces since, unlike in the case of non-bonded interactions, the interactions are fixed by the system topology.

Given that there already exists a SYCL-based backend for non-bonded forces in GROMACS which targets AMD GPUs on upcoming pre-exascale machines such as LUMI-G, we decided to focus our limited effort during the extension on ensuring performance portability of the NB-LIB listed forces implementation. To achieve this performance portability, we implemented the NB-LIB listed forces kernels as plain C++ functions that can be called on any architecture, including CPUs and NVIDIA or AMD GPUs with support for common programming frameworks such as CUDA, HIP and (HIP-)SYCL. This design principle of having shared kernels and mostly shared call stacks for hardware from different vendors, as well as different types of hardware, CPU or GPU, is critically important for codes that can run on exascale machines. This allows scientists to add new types of interaction potentials without having to understand the full complexity of the ever-growing profusion of architectures. It also allows performance engineers to much more easily port code to new architectures. If a specific kernel turns out to run poorly, it is also very easy with this setup, which leverages the extraordinary power of template metaprogramming in C++, to specialize individual kernels without needing to change the overall call stack.

In order to run on AMD GPUs, the existing CUDA kernel can either be compiled unchanged with HIP or the aforementioned plain C++ portable interaction kernels can be called from a HIP-SYCL specific loop over the listed interactions. While the duration of the extension did not ultimately include the delivery of LUMI-G, we were able to test that the listed forces API gets good performance on existing machines with Intel CPU and NVIDIA GPU, such as Piz Daint, and existing machines with AMD CPU, such as Dardel, which has a similar architecture to LUMI-C. The power and flexibility offered by the NB-LIB force calculation APIs are a major step forward for particle simulation codes and the extension of the PRACE project has greatly aided setting NB-LIB on a firm footing with regard to software uptake and sustainability.