E-Infrastructures
H2020-EINFRA-2015-1

EINFRA-5-2015: Centres of Excellence
for computing applications

EoCoE

Energy oriented Center of Excellence
for computing applications

Grant Agreement Number: EINFRA-676629

D1.4 M36
Application Support Outcome

Project and Deliverable Information Sheet

EoCoE	Project Ref:	EINFRA-676629
	Project Title:	Energy oriented Centre of Excellence
	Project Web Site:	http://www.eocoe.eu
	Deliverable ID:	D1.4 M36
	Lead Beneficiary:	Juelich JSC
	Contact:	Paul Gibbon
	Contact e-mail:	p.gibbon@fz-juelich.de
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M36 30/09/2018
	Actual Date of Delivery:	30/09/2018
	EC Project Officer:	Carlos Morais-Pires

* - 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 2991/844/EC.

Document Control Sheet

Document	Title:	Application Support Outcome
	ID:	D1.4 M36
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	Achilles (JSC), Breuer (JSC, WP1), Brömmel (JSC, WP1), Haefele (MdlS, WP1), Halver (JSC, WP1), Kuhn (INRIA, WP1), Lanteri (Inria, WP1), Latu (CEA, WP5), Leleux (CERFACS, WP1), Lührs (JSC, WP1), Ould-Rouis (MdlS, WP1), di Serafino (UNICAMPANIA, WP1), Sharples (JSC, WP4), Tamain (CEA, WP5), Torun (IRIT-CNRS, WP1)
	Contributors:	Aeberhard (IEK-5, WP3), Agullo (INRIA, WP1), D'Ambra (CNR, WP1), Béréziat (LIP6, WP2), Berndt (IEK-8, WP2), Bigot (CEA, WP1), Brdar (JSC, WP4), Bruckmann (RWTH, WP4), Büsing (RWTH, WP4), Duff (CERFACS, WP1), Filippone (WP2), Frings (JSC, WP1), Gageat (MdlS, WP3), Gimenez (BSC, WP1), Giraud (INRIA, WP1), Gobé (Inria, WP1), Görgen (IBG-3, WP4), Haefele (MdlS, WP1), Hassan (WP2), Hastaran (WP1), Herlin (INRIA, WP2), Houzeaux (BSC, WP2), Kaliszan (PSNC), Knobloch (JSC, WP1), Kollet (IBG-3, WP4), Kuhn (INRIA, WP1), Kulkarni (JSC, WP4), Lanteri (Inria, WP1), Latu (CEA, WP5), Lührs (JSC, WP1), Marait (WP1), Marin-Lafleche (MdlS, WP1), di Napoli (JSC, WP4), Naz (IBG-3, WP4), Niederau (RWTH, WP4), Ould-Rouis (MdlS, WP1), Owen (BSC, WP2), Passeron (WP5), Poirel (WP1), Poorthuis (JSC, WP4), Roussel (WP1), Ruiz (IRIT, WP1), Steinbusch (JSC, WP1), Tamain (CEA, WP5), Viquerat (Inria, WP1), Wylie (JSC, WP1), Zhukov (JSC, WP1)
	Reviewed by:	Haefele (MdlS), Gibbon (JSC), PEC members

Contents

1. Overview

Deliverable D1.4 reports on all activities concerning application support, including applications the consortium decided to give support to prior to the start of the project, but also applications which partners requested work on during the course of the project.

Table 1: Contribution of Deliverable D1.4 to impacts 1.1, 1.2, 1.3, and 3.1

Code	Lead institute	Performance gain (%)	CPUh saved (MCPUh) in 2018	in total	EoCoE tools usage
Alya	BSC	10	0.4	1.8	4
Eirene	FZJ	-	-	-	
Esias	FZJ	30	3	18	
Gysela	CEA	100	90	146	2
MDFT	CEA	1200			
Metalwalls	CEA	1440	13.6	60.6	
PVnegf	FZJ	295	-	-	
Parflow	FZJ	<10	N/A	N/A	2
Shemat	RWTH Aachen	7	ε	ε	
SolarNowcast	INRIA	440		ε	
Telemac	EDF	65		ε	
Tokam3X	CEA	6			1
Total	-	-	107	226.4	9

Table 1 is an update of the table in D1.3 and lists all applications with triggered application support contributing to impacts 1.1, 1.2, 1.3, and 3.1. Where applicable, performance gain and the CPU time saved in 2018 and over the complete project period are updated. The perfomance gain relates to the speed-up of the codes during EoCoE, comparing wallclock times for test-cases at project start, t_{start}, and project end, t_{end}: $t_{\text{start}}/t_{\text{end}} - 1$. The saved CPU time assumes how much time would have been required in addition to the one used for current projects if codes had not been improved in their performance. Over all codes, an estimated total of 107 MCPUh could be saved in 2018 and 226.4 MCPUh in total by the groups using these applications. So impact 1.1 whose goal was to save 50 MCPUh/year was reached once again.

ESIAS and Gysela are the main contributors here as they use significant CPU time quotas each year. Even moderate performance improvements on these codes compared to some others have a strong impact on the better usage of underlying computing infrastructures. Metalwalls, on the other hand benefited from a large performance gain and moderate CPU time allocations. Alya benefited from a small acceleration thanks to the optimisation of the matrix assembly part of the code. As the main bottleneck is related to linear algebra, the Alya team is in strong contact with the linear algebra team of the transversal basis and could test four different packages in the code.

A number of codes (MDFT, Parflow, PVnegf, Shemat, SolarNowcast, Telemac, Tokam3X) did show a moderate to large performance gain at the end of the project but did not save a lot of CPU hours because they do not run on very large CPU time allocations. However, their performance increase had a big impact on the daily routine of the scientists, the type of problems they were able to treat, and made them more future-proof. Out of those, the very large performance gain of MDFT, PVneg, and SolarNowcast was possible because these two applications were serial or with an inefficient OpenMP implementation. Support activities on these two codes enabled them to leverage the full computing power of a single node.

Finally, the impact on codes Eirene and PVnegf has to be seen differently. While they do not show an immediate performance gain (and hence also no saved CPU hours), their application support resulted in guidelines on what needs to be considered for their future developments on different architectures. In case of PVnegf, a first proof-of-concept prototype is already in place that employs hybrid parallelisation, effective vectorisation, and scales to 458,752 cores.

2. Alya

Overview

The Alya System is the BSC simulation code for multi-physics problems. It addresses problems of incompressible and compressible flows, non-linear solid mechanics, species transport equations, excitable media, thermal flows, N-body collisions ...

The application submitted to EoCoE is a coupling of 2 modules: NASTIN, a FE solver for incompressible Navier Stokes and TURBUL, a FE solver for turbulence equations. It applies to wind simulation for wind farms.

It is pure MPI. Each module contains a matrix assembly part, that is perfectly distributed, and a solver part that requires communications at each iteration.

Code team:

- Herbert Owen (BSC), WP2
- Guillaume Houzeaux (BSC), WP2
- Yacine Ould Rouis (MdlS), WP1

Performance metrics

Benchmark characteristics:

Domain size	1 Million elements
Number of timesteps	30
Compile options	-O2 -xHost -DNDIMEPAR
Resources	1 node on JURECA (24 cores)
IO details	default sequential IOs, parallel hdf5 output is tested in a second step
Type of run	the size of benchmark aims to be faithful to the regular use of the program, in terms of number of elements per node

Application support

Code optimization for the matrix construction

Activity type	WP1 support
Contributors	Y. Ould-Rouis (WP1), H. Owen (WP2), G. Houzeaux (WP2)

The application support on Alya focuses exclusively on the matrix construction parts of the two modules involved in the application submitted to EoCoE: NASTIN and TURBUL. The work has been conducted based on the conclusions of the performance evaluation, with a strong collaboration and communication with the Alya team.

In the following, I describe point-by-point the different actions and steps that brought improvements to the code's performance.

Table 2: Performance metrics for Alya on the JURECA HPC system.

	Metric name	original	after App Support
Global	Total Time (s)	385.4	346.3
	Time IO (s)	0.5	0.4
	Time MPI (s)	99.7	90.1
	Memory vs Compute Bound	1.3	1.3
IO	IO Volume (MB)	2449.9	2449.9
	Calls (nb)	97655	97573
	Throughput (MB/s)	5069.0	6423.6
	Individual IO Access (kB)	4.9	4.9
MPI	P2P Calls (nb)	154493	151985
	P2P Calls (s)	4.1	4.3
	Collective Calls (nb)	100071	98609
	Collective Calls (s)	0.7	0.8
	Synchro / Wait MPI (s)	94.2	84.9
	Ratio Synchro / Wait MPI	94.5	94.2
	Message Size (kB)	15.4	15.4
	Load Imbalance MPI	20.6	19.9
Mem	Memory Footprint (B)	584 mB	584 mB
	Cache Usage Intensity	N.A.	N.A.
	RAM Avg Throughput (GB/s)	N.A.	N.A.
Core	IPC	N.A.	N.A.
	Runtime without vectorisation (s)	383.2	362.9
	Vectorisation efficiency	1.0	1.0
	Runtime without FMA (s)	392.7	353.5
	FMA efficiency	1.0	1.0

1. Detection of loop level pathologies using VTune.
2. Loops reordering: in order to take profit of data locality in the cache by contiguous accesses.
3. Refactoring of potential redundant calculations.
4. Helping vectorization: by data restructuring and getting rid of dependencies inside inner loops.
   For example: in *nsi_elmmat*, building elauu matrix as 9 distinct arrays that are assembled together at the end allowed an 11% improvement of this routine, in addition to the 15% obtained with the 2 previous steps.
5. Memory padding (or data structure alignment). tested only on some local array variables.
6. Eliminating while/conditional loops.
7. Element to sparse matrix pre-mapping: This step was motivated by the observation that *csrase* subroutine contains a very costly while loop (2.7% of total CPU time), c.f. Figure 1. This subroutine is used in turbul matrix assembly, to copy

Table 3: Detailed time performance on JURECA - original performances.

mode	CPU_time	Start_ops	NSI_total	NSI_mat	NSI_sol	TUR_total	TUR_mat	TUR_sol
ref	**384.66**	37.9	**203.57**	**67.24**	**132.27**	**125.46**	**87.87**	**32.21**
darshan	385.34	37.48	204.16	67.19	132.68	125.79	87.6	32.25
scatter	311.28	36.15	148.91	65.71	79.04	111.22	84.81	20.33
compact	396.5	35.89	207.43	68.42	134.05	130.53	88.75	36.35
memory	384.95	38.17	202.98	67.12	131.99	125.96	88.35	32.22
scalasca	**477.1**	49.96	213.28	76.08	133.24	187.04	149.02	32.97
no-fma	392.03	38.44	207.12	70.93	132.15	127.32	89.76	32.35
no-vec	**381.93**	38.94	**199.97**	60.92	134.94	125.83	85.25	34.95

Table 4: Detailed time performance on JURECA - after matrix assemblies optimization.

mode	CPU_time	Start_ops	NSI_total	NSI_mat	NSI_sol	TUR_total	TUR_mat	TUR_sol
ref	**345.65**	37.79	**180.85**	43.85	130.4	108.29	**68.04**	31.61
darshan	346.04	37.16	181.72	44.04	130.37	109.66	67.95	31.6
scatter	279.39	35.97	131.09	43.37	79.23	96.62	66.27	20.32
compact	351.33	36.14	184.36	44.46	131.54	113.0	68.8	34.72
memory	348.77	38.4	182.63	44.04	130.92	108.88	67.65	31.9
scalasca	424.13	49.35	190.1	51.91	131.02	155.94	114.58	32.15
no-fma	352.59	37.86	185.82	47.89	130.43	110.44	69.1	31.66
no-vec	361.56	40.01	193.52	56.74	128.78	110.75	68.84	31.99

the coefficients calculated by element into a larger sparse matrix, and the while loop finds for each coefficient of indices (inode, jnode) in a local element's matrix the right index 'izsol' in the CSR matrix. Same is done other way around, and in other parts/modules.
The idea is to calculate this relation once in a pre-processing step.

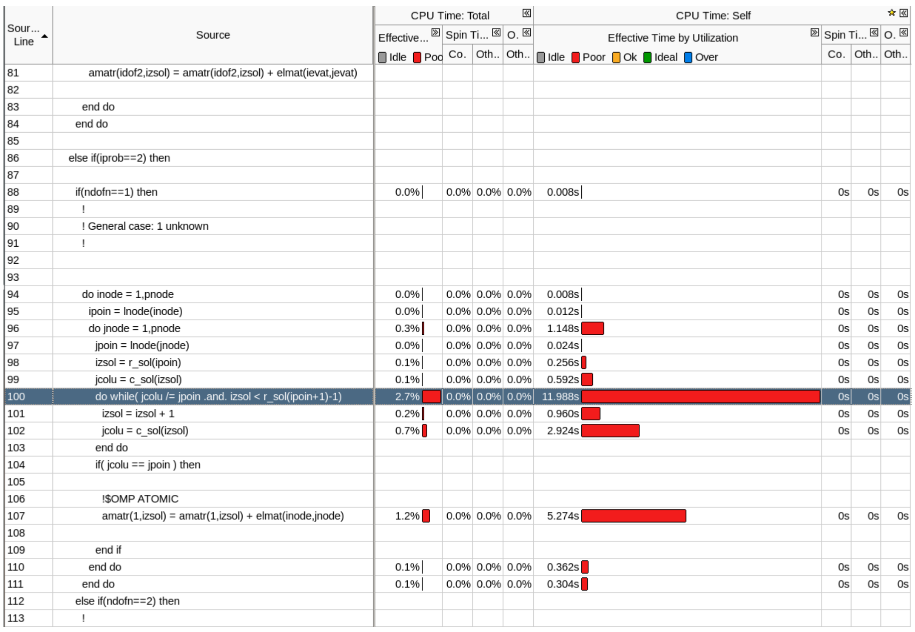
- advantages: avoid a costly redundant while loop.
- neutral: it won't solve the problem of indirections when writing in the sparse matrix.
- negative: an estimated memory cost of 8*8*integer_size for each element. With long integers (8 bytes), this means 50MBytes for each 100 000 elements. This is reasonable when compared to the total memory footprint.

Results

The core level optimization has been successful in securing 10 to 13% gain in total, depending on the hardware. The Figure 2 shows the evolution of the performance for the successive versions of Alya. The stage 0 shows the original times. Stages 1 to 6 show the results of steps 2 to 6 described above, applied to the NASTIN matrix assembly. Stages 7 to 11 are the results of steps 2 to 6 applied to the TURBUL module combined with other efforts.

The final results on JURECA are as follows:

- NASTIN matrix assembly: 35% improvement.
- TURBUL matrix assembly: 22% improvement.
- Total: 10% global improvement.



Sour... Line	Source	CPU Time: Total				CPU Time: Self			
		Effective...	Spin Ti...		O.	Effective Time by Utilization	Spin Ti...		O.
		⬛ Idle 🟥 Poo	Co.	Oth..	Oth..	⬛ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Co.	Oth..	Oth..
81	amatr(idof2,izsol) = amatr(idof2,izsol) + elmat(ievat,jevat)								
82									
83	end do								
84	end do								
85									
86	else if(iprob==2) then								
87									
88	if(ndofn==1) then	0.0%	0.0%	0.0%	0.0%	0.008s	0s	0s	0s
89	!								
90	! General case: 1 unknown								
91	!								
92									
93									
94	do inode = 1,pnode	0.0%	0.0%	0.0%	0.0%	0.008s	0s	0s	0s
95	ipoin = lnode(inode)	0.0%	0.0%	0.0%	0.0%	0.012s	0s	0s	0s
96	do jnode = 1,pnode	0.3%	0.0%	0.0%	0.0%	1.148s	0s	0s	0s
97	jpoin = lnode(jnode)	0.0%	0.0%	0.0%	0.0%	0.024s	0s	0s	0s
98	izsol = r_sol(ipoin)	0.1%	0.0%	0.0%	0.0%	0.256s	0s	0s	0s
99	jcolu = c_sol(izsol)	0.1%	0.0%	0.0%	0.0%	0.592s	0s	0s	0s
100	do while(jcolu /= jpoin .and. izsol < r_sol(ipoin+1)-1)	2.7%	0.0%	0.0%	0.0%	11.988s	0s	0s	0s
101	izsol = izsol + 1	0.2%	0.0%	0.0%	0.0%	0.960s	0s	0s	0s
102	jcolu = c_sol(izsol)	0.7%	0.0%	0.0%	0.0%	2.924s	0s	0s	0s
103	end do								
104	if(jcolu == jpoin) then								
105									
106	!$OMP ATOMIC								
107	amatr(1,izsol) = amatr(1,izsol) + elmat(inode,jnode)	1.2%	0.0%	0.0%	0.0%	5.274s	0s	0s	0s
108									
109	end if								
110	end do	0.1%	0.0%	0.0%	0.0%	0.362s	0s	0s	0s
111	end do	0.1%	0.0%	0.0%	0.0%	0.304s	0s	0s	0s
112	else if(ndofn==2) then								
113	!								

Figure 1: Part of *csrase* subroutine instruction level profiling, unveiling a costly loop.

The methodology and the interesting results of this work generated new interests in the Alya team, that triggered new optimization efforts on the other modules composing Alya.

In early 2018, we updated the Alya performance evaluation JUBE script to the latest state of art. The runs with the latest Alya version available, and with Intel 2018, on JURECA, show an improvement of 3% in the execution time (10 seconds) compared to the previous latest results. While the general picture stays roughly the same, the MPI time shows an improvement of 30%.

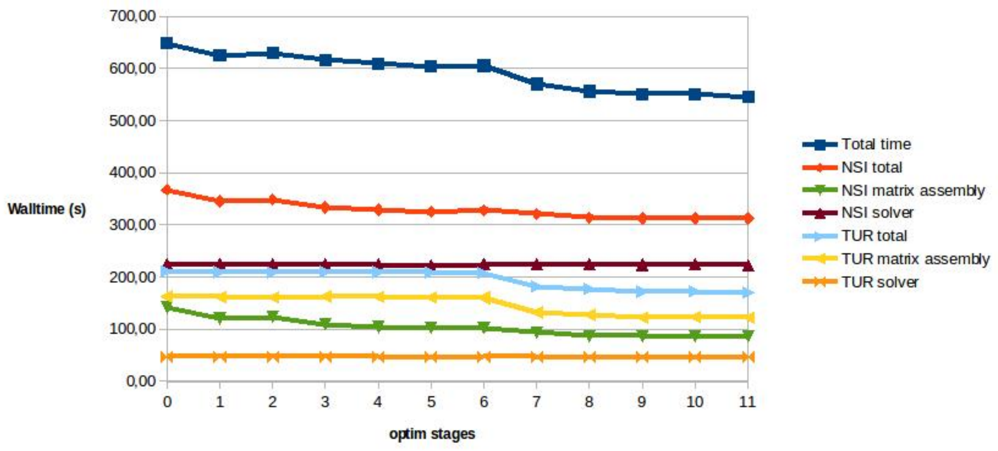


Figure 2: ALYA NSI+TUR perf evolution - 1 Melem, 30 timesteps, 1 node (16 processes) on MareNostrum.

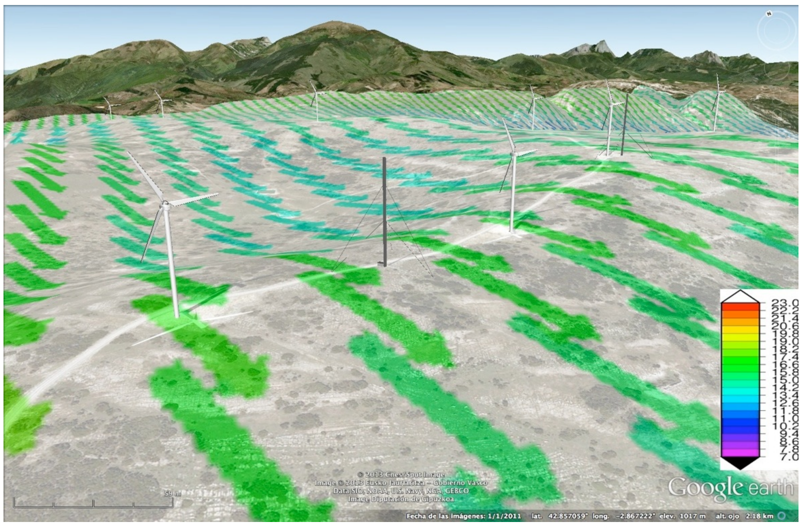


Figure 3: The windfarm test case.

Integration of the Maphys solver in Alya

Activity type	Consultancy or WP1 support
Contributors	E. Agullo (WP1), L. Giraud (WP1), G. Houzeaux (WP2), M. Kuhn (WP1), G. Marait (WP1), L. Poirel (WP1)

This section deals with the usage of `Maphys` [5] into Alya high performance computing simulation code. Our goal with this study was to evaluate and improve the performances of `Maphys` coarse grid correction mechanism into an applicative context.

For that purpose, we confronted Alya's internal solvers and `Maphys` solver for linear algebra on test cases implemented into Alya, with a particular focus on test cases leading to find the solution of symmetric positive definite systems. In this case, coarse grid correction or deflation mechanisms can be used into both Alya's internal solvers and `Maphys` .

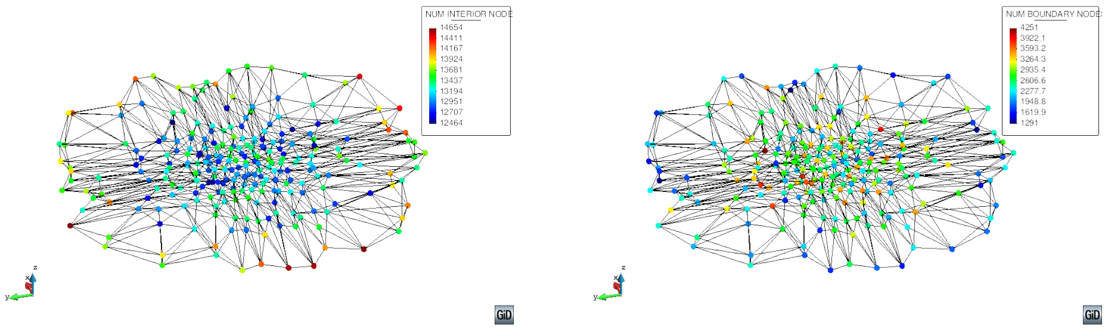
Two test cases have been chosen for this study:

- the simulation of a wind farm,
- and the simulation of the airflow through the nose during a sniff.

To better understand the results presented in this section, please refer to the available description of `Maphys` in Deliverable 1.7 of the EoCoE project.

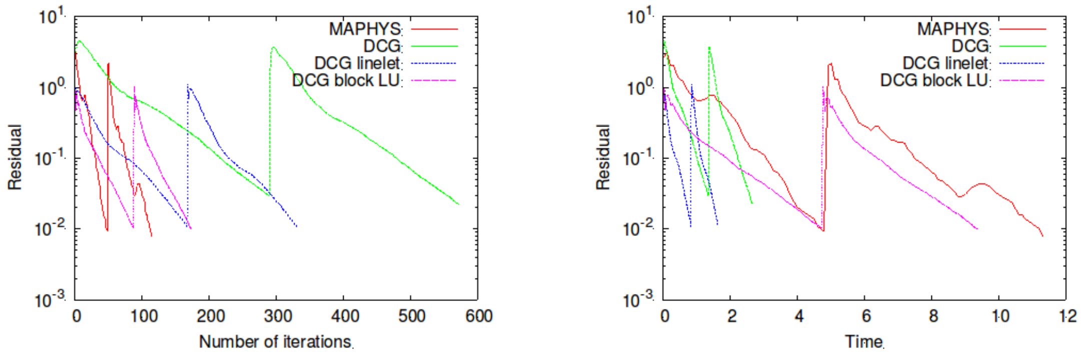
Simulation of a windfarm. The simulation of a windfarm, Figure 3, has first been chosen as a candidate for a detailed analysis of `Maphys` solver in the frame of Alya simulation code. This simulation involves the Navier-Stokes equations together with a k-e turbulence model.

The mesh consists of a circled and flat domain with boundary layer elements. Only HEX08 elements are used for its discretisation. The basic mesh contains 3.7M elements, 3.8M nodes. The number of elements can be increased through mesh division into Alya to reach a better accuracy. An example of domain decomposition on 255 subdomains is given by Figure 4, where one can observe that the domain decomposition is almost 2-dimensional.



(a) Number of interior vertices per domain (b) Number of interface vertices per domain

Figure 4: Windfarm test case: pseudo-2D domain decomposition into 255 subdomains.



(a) Number of iterations (b) Time

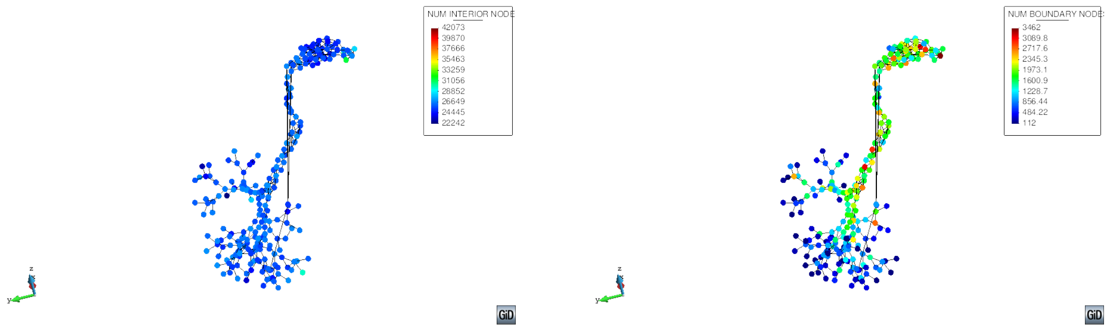
Figure 5: Windfarm test case: convergence history on 511 subdomains. Maphys without factorization time.

The equation concerned by Maphys into this test case is the pressure equation. Its discretisation leads to find the solution of an SPD linear system. In this case, it is possible to consider the use of Alya's deflated CG and of Maphys coarse grid.

For more details on the simulation of a windfarm into Alya, please refer to [1].

Figure 5 shows a convergence history on this windfarm test case, simulated on 512 computational cores. The figure plots the residual as a function of the number of iterations on the left and the time to solution on the right. For the Maphys solver (in red), the basic (without coarse grid) configuration has been considered, with a local dense preconditioning technique. For Alya, the deflated CG algorithm has been employed, jointly with three preconditioning techniques (in green, blue and purple). As can be seen on the left figure, the number of iterations is lower for Maphys than for any of Alya's deflated CG version. However, on the right figure, the time to solution for Maphys is approximatively 5 times larger than the best Alya's deflated CG configuration.

These last results motivate the need of a performant and scalable coarse grid correction study into Maphys . This study has been performed on a more suitable test case for coarse grid or deflation technique study, allowing to better illustrate the benefit of using the coarse grid mechanism or deflation technique. This other test case involves a pseudo-1D domain decomposition instead of the windfarm test case's pseudo-2D one, and is the topic



(a) Number of interior vertices per domain (b) Number of interface vertices per domain

Figure 6: Respiratory test case: pseudo-1D domain decomposition into 255 subdomains.

of the next paragraph. Indeed, for a 1D decomposition the coarse grid correction plays a critical role on the numerical behaviour since the condition number growths linearly with the number of domains, while the growth is $\mathcal{O}((\#\,domains)^{\frac{1}{2}})$ ($\mathcal{O}((\#i\,domains)^{\frac{1}{3}})$) for 2D-decomposition (resp. 3D-decomposition). Because we do not have yet access to very large computer with large number of cores we prefer to consider 1D-decomposition where the critical numerical behaviour will be easy to observe already for a moderated number of cores.

Simulation of the airflow through the nose: the Respiratory test case. The simulation of the airflow through the nose has been chosen to perform an evaluation of different coarse grid implementations into `Maphys` . This test case simulates the airflow through the nose and large airways by solving the incompressible Navier-Stokes equations.

Three types of elements are in use for the mesh discretisation: TET04, PYR05 and PEN06, for a total of 17.7M elements and 6.9M nodes. The mesh is characterised by a very elongated geometry with small passages in the nasal cavity, leading to a pseudo-1D elongated domain decomposition when parallelising through partitioning the mesh, see Figure 6. This property makes this test case a very good candidate to evaluate the coarse grid of `Maphys` in an applicative context.

On the algebraic solver side, the discretisation of the problem leads to a coupled algebraic system to be solved at each time step. This algebraic system is split to solve independently the momentum and the continuity equations. Due to the splitting strategy, it is necessary to solve the momentum and the continuity equations twice per time step. As the problem is non-linear, the matrix changes between each time step.

The continuity equation is considered for the solver comparison study. This equation leads to the assembly of a SPD linear system. Due to the elongated geometry, low frequencies are hardly damped with a classical one level domain decomposition approach. Hence, coarse grid or deflation mechanisms are investigated to solve the continuity equation.

For more details about this test case, please refer to [2].

All the simulations presented into this section have been performed on the GENCI's OCCIGEN cluster, hosted by the CINES. The part of the cluster in use is composed of 2 Dodeca-core Haswell Intel Xeon E5-2690 v3 @ 2.6 GHz nodes with 64 and 128 Go RAM per node. The code was compiled with Intel compiler version 17.0.0, and linked with the multithreaded Intel MKL version 2017.0.0 and Intel MPI version 2017.0.0. All the runs are made such that the nodes of the cluster are fully occupied (hence the number of cores is always a multiple of 24). Notice that on the OCCIGEN cluster, memory swapping is disabled by default. The simulation campaigns were realised with the help of JUBE Benchmarking Environment, allowing to explore parameters and analyse results comfortably.

The parallel benchmarks have been performed in mono-threaded configuration, on 264, 528, 1056 and 2112 MPI processes, leading respectively to 265, 527, 1055 and 2111 subdomains in the domain decompositions (as Alya has a master process). The iterative solvers' stopping criterion is set to 10^{-6}, to be reached in a maximum of 2000 iterations. For each experiment, 10 time steps are performed, each time step requiring two substeps.

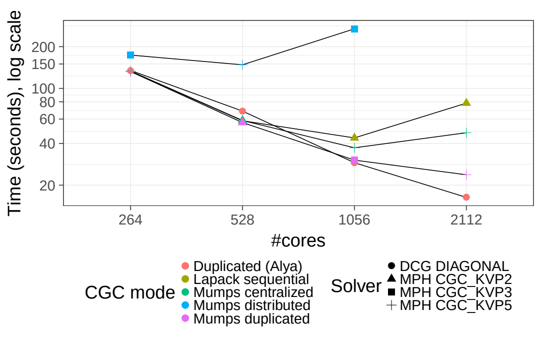
Results are displayed on Figure 7. This figure consists of four quadrants, showing the solver total time (Fig. 7a), the global preconditioner application time for `Maphys` (Fig. 7b), the speedups (Fig. 7c) and the efficiencies (Fig. 7d) of the solvers depending on their preconditioning strategies.

On the Alya's internal solver side, a Deflated Conjugate Gradient method is employed, together with a diagonal preconditioner, `DCG DIAGONAL` on the figures, with a duplicated coarse space of fixed order 10000. The corresponding Coarse Grid Correction mode (CGC mode) for Alya is unique, denoted by `Duplicated (Alya)` on Figure 7.

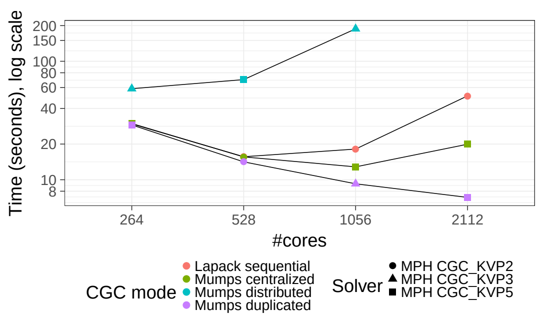
On the `Maphys` side, several two level preconditioning techniques with coarse grid correction are considered for the iterative solution to the Schur system: `MPH CGC_KVPn` on the figures, with `n` the number of eigenvalue/eigenvector pairs computed for the coarse grid per subdomain. The order of the coarse problem to be solved is then $n \times \#cores$. The four formerly coarse grid correction implementations are displayed on Figure 7. For each CGC mode, only the number of eigenvalue/eigenvector pairs `n` leading to the lowest total computation time is displayed. For the `Mumps centralized`, 12 MPI processes were in use to solve the coarse problem. For the `Mumps duplicated` mode, the coarse problem has been replicated on disjoint groups of 12 MPI processes. As the matrix changes between each time step, `Maphys` has to perform several times its factorization step in order to factorize the local interior problems and to compute the local Schur complements. The preconditioner (local and coarse) are set up to remain fixed through the time steps. If necessary, it could be set up to be recomputed at a predetermined fixed frequency.

By focusing on the first `Mumps distributed` implementation of the coarse grid, one can observe on Figure 7a (in blue), that `Maphys` coarse grid correction performs poorly in front of Alya's internal deflated CG solver. Into this CGC mode, `Maphys` was not able to scale beyond 528 cores, and did not give a solution for 2112 cores (Out Of Memory (OOM) event on the compute nodes). When having a look at the performances of `Mumps distributed` CGC mode concerning the global preconditioner application on Figure 7b (still in blue), one can identify the required computation time for this part of the iterative process of `Maphys` increases with the number of processes, representing then an increasing ratio of the total computation time. The main reason of these results is that the coarse problem is solved with Mumps sparse direct solver with its distributed entry on too many MPI processes, leading to a too fine granularity hence implying poor performances.

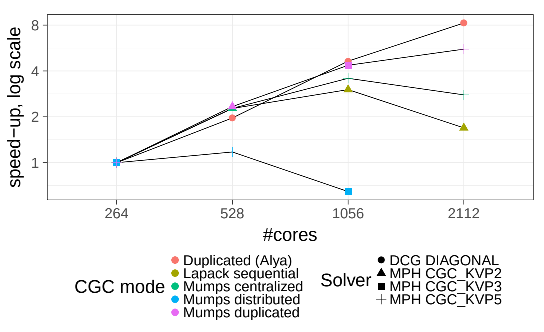
In order to improve performances, two other CGC modes have been implemented, namely `Lapack sequential` and `Mumps centralized`. These CGC modes are displayed in greeny-yellow and in green on Figure 7. These two implementation strategies allow to compete with Alya internal solver up to 1056 cores, giving better results on both 264 and 528 cores, see Figure 7a. Notice the results for the `Mumps centralized` version become better than the `Lapack sequential` version when increasing the number of cores. This is due to the order of the coarse problem that increases with the number of domains in use which makes it worth to exploit the sparsity pattern of the coarse matrix. However, these strategies do not scale beyond 1056 cores. This is mainly due to the global MPI communications required at the beginning and at the end of the global preconditioner application, whose computation time again increases with the number of processes, representing then an increasing ratio of the total computation time, see Figure 7b in greeny-yellow and in green.



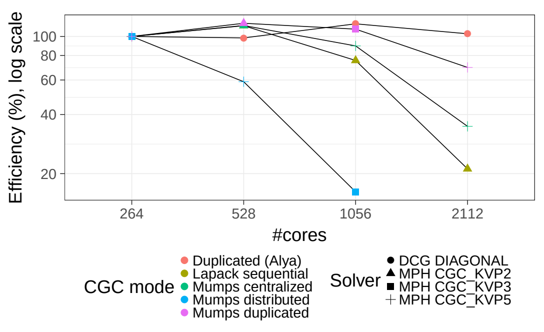
(a) Solver total time to solve the continuity problem



(b) `Maphys` only: global preconditioner application time



(c) Solver speedup in solving the continuity equation



(d) Solver efficiencies in solving the continuity equation

Figure 7: Evalutation of `Maphys` scaling with different coarse grid implementations on the respiratory test case.

To go beyond the former limitation, a last CGC mode has been implemented: `Mumps duplicated`. This coarse grid parallel implementation is closer to Alya's deflation implementation strategy, and allows to save one global MPI communication in the global preconditioner application process of `Maphys` ' iterative solve part as a comparison to the three former parallel algorithms. On Figure 7b, in purple, one can observe this last global communication bypass allows the global preconditioner application to scale up to the 2112 cores in use for these parallel experiments with this implementation strategy. However, for the solver total time on Figure 7, there is still a gap of approximatively 10 seconds between this last version and Alya's internal solver. This gap is mainly due to the non-ideal scaling of `Maphys` ' solve phase despite the new strategy and because of the factorization phase which also scale less successfully between 1056 and 2112 than before to reach this amount of computing resources.

To sum up, the developments in the frame of this comparative study enabled to significantly improve the efficiency and the scalability potential of `Maphys` (see Figures 7c and 7d) with the use of its coarse grid correction mechanism in the case of SPD linear systems. Indeed, on 1056 cores, the first coarse grid parallel implementation, i.e. `Mumps centralized` CGC mode, led to 13 % efficiency relatively to 264 cores against 109 % with the most performant CGC mode `Mumps duplicated`. The `Mumps duplicated` CGC mode also enabled to obtain results on 2112 cores with 69% efficiency against 34 % (respectively 21 %) for the less performant `Mumps centralized` (resp. `Lapack sequential`) CGC modes.

PSBLAS and MLD2P4 for Alya

Activity type	WP1 support
Contributors	Ambra Abdullahi Hassan (University of Rome “Tor Vergata”, Italy), Pasqua D’Ambra (CNR, Italy), Daniela di Serafino (University of Campania “L. Vanvitelli”, Italy), Salvatore Filippone (Cranfield University, UK), Herbert Owen (BSC, Spain) for WP2

The improved versions of PSBLAS and MLD2P4 developed during the EoCoE project (see Deliverable D1.7) have been applied to a data set from Alya. The goal of this work was to provide a sound basis for the selection of solvers and preconditioners for future integration and tuning into the application code.

Data Set

The set of linear systems comes from computational fluid dynamics simulations for wind farm design and management, carried out at BSC, within WP2 (*Meteorology for Energy*), by using the HPC multi-physics simulation code Alya [7]. The systems arise from the numerical solution of Reynolds-Averaged Navier-Stokes equations coupled with a modified $k-\varepsilon$ model. The space discretization is obtained by using stabilized finite elements, while the time integration is performed by combining a backward Euler scheme with a fractional step method, which splits the computation of the velocity and pressure fields and thus requires the solution of two linear systems at each time step. Four test cases were made available by BSC, each including a pressure and a velocity system. The dimensions of the pressure systems range from 305472 to 2224476, and the corresponding number of nonzeros from 8060182 to 58897774. The dimensions and the number of nonzero entries of the velocity system range from 916416 to 6673428 and from 72541638 to 530079966, respectively. The data partitioning used by Alya on 8, 16, 32 and 64 processors was also provided for one of the test cases, having pressure and velocity systems of dimensions 1123344 and 3370032, respectively. We focused mainly on the pressure systems, whose sparsity pattern is shown in Figure 8.

In order to assess the behaviour of different MLD2P4 preconditioners on the selected test case and choose the best ones for the applications of interest, an evaluation in terms of execution time, strong and weak scalability, and linear solver iterations was carried out.

Results on the Data Set

We applied to the linear systems from Alya the PSBLAS Conjugate Gradient (CG) solver with algebraic multilevel preconditioners implemented in MLD2P4, using several choices of smoothers and coarsest-level solvers. We also used the one-level block-Jacobi precondi-

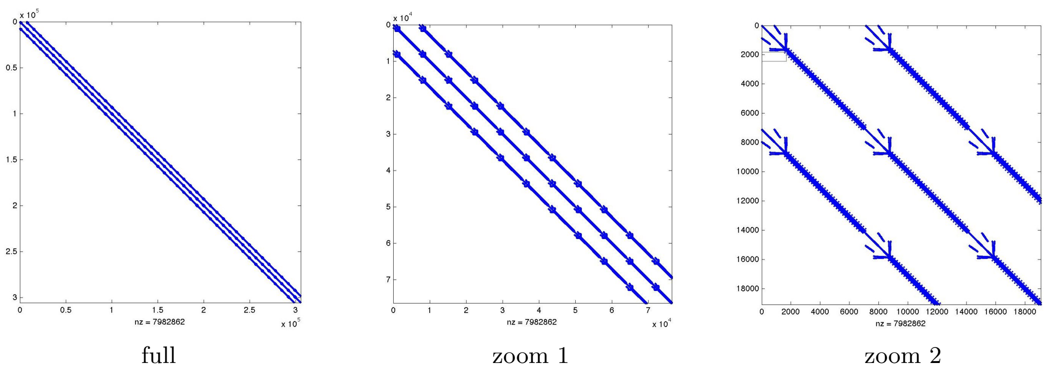


Figure 8: Pressure matrices from wind farm simulations: sparsity pattern (full matrix and details).

tioner, applying ILU(0) to the blocks. The zero vector was chosen as starting guess; the reduction of the 2-norm of the residual by a factor of 10^{-6} was used as stopping criterion. The experiments were run on a linux cluster, named yoda, operated by the Naples Branch of the CNR Institute for High-Performance Computing and Networking.

The results obtained with the multilevel preconditioners implemented in MLD2P4 are not satisfactory. These preconditioners result very sensitive to the matrix distribution, and the performance quickly deteriorates as the number of parallel processes increases, often leading to stagnation of CG. Furthermore, a strong load imbalance arises when the data partitioning from Alya is used. A crucial role is played by the aggregation threshold used by the smoothed aggregation algorithm at each level, but determining this threshold is not an easy task. Conversely, the block-Jacobi preconditioner has a reasonable performance, although the number of iterations increases significantly with the number of cores. As an example of this behaviour, in Table 5 we report the number of CG iterations, the times (in seconds) for building the preconditioners and applying the preconditioned CG, and total time obtained with the block-Jacobi preconditioner on the pressure matrix with dimension 1123344, using the Alya data partitioning. Further experiments showed that

Table 5: Pressure matrix from wind farm simulation (dim. 1123344): number of iterations and execution time (in seconds) on yoda.

BLOCK-JACOBI				
procs	iters	tprec	tsolve	ttot
8	285	0.2	6.4	6.6
16	323	0.1	4.4	4.5
32	354	0.1	2.9	3.0
64	430	0.0	1.8	1.8

the multilevel preconditioners implemented in MLD2P4 are not effective with the velocity systems too. In particular, the smoothed aggregation algorithm implemented in MLD2P4 is not effective with linear systems coming from discretization grids where each node is associated with multiple variables, as is the case with the velocity systems.

The results obtained on the pressure and velocity matrices motivated our interest toward new coarsening algorithms, aimed at improving the convergence behaviour and the parallel performance of the smoothed-aggregation based multilevel preconditioners implemented in MLD2P4. This is part of a longer-term research directed toward a more general revision of the package, which may include the design and the developments of new methods. First results in this direction appear very promising and are reported in Deliverable D1.11 (Applied research activities outcome).

AGMG for Alya

Activity type	WP1 support
Contributors	Herbert Owen (BSC, WP2), Yvan Notay (ULB, WP1)

As concluded in earlier performance reports, Alya can possibly be improved by using external, well optimized, software packages for the solver part of the code, instead of the so far implemented in house method.

Here we consider in particular CFD algorithms that use the solution of a discrete Poisson problem to compute a pressure correction (at each time step and within each non linear step). Such solves represent a significant part of the total compute time. Because of the nature of the linear systems to solve, the AGMG solver [1] is a potential candidate to substitute the native solver.

Comparison between Alya's native solver and AGMG (standard version)

For CFD problems, two different approaches can be used: RANS and LES. It turns out that the numerical treatment and the kind of meshes used significantly affect the behaviour of AGMG.

RANS (Reynolds average Navier-Stokes): RANS is cheaper and more robust than LES. All the turbulence scales are modelled. It is the approach typically used by most of the industry. The meshes used are highly anisotropic and both the velocity and pressure are solved implicitly. AGMG does not work well for this kind of problems. The anisotropic meshes discretised with finite elements are not well suited for AGMG. For this kind of problems Alya's own solvers (GMRES, CG or deflated CG [7]) typically provide smaller solution times.

LES (Large eddy simulation): When LES is used the small scales are modelled but the large scales are resolved. It is significantly more expensive than RANS but it can provide much more accurate results for problems with important separation as those that occur in flow over complex terrain. With the constant increase in computational resources, pioneering companies are moving towards LES. For incompressible LES problems the velocity is treated explicitly and the pressure implicitly. It means that, for a complete simulation, several hundred thousands of pressure corrections have to be computed; i.e., several hundred thousands linear systems have to be solved. With the native (in house) solver in Alya, these solves represent typically 30% of the total computing time.

[1] AGgregation-based algebraic MultiGrid; `http://www.agmg.eu`

Possibly because LES meshes are much less anisotropic than RANS meshes, AGMG has proved to be very successful for these kind of problems. In Table 6 we present results for the calculation of the pressure for a flow with complex geometry. The mesh is isotropic and has 8 M nodes. The number of cores used vary from 72 to 288. Thus, the average number of nodes per sub-domain varies from 111 k nodes to 27 k nodes respectively. The native solver in Alya is a based on a deflated conjugate gradient method [4] (with the number of groups set to 1000).

Table 6: Elapsed time per time step (AGMG setup time can be neglected, cf. text).

number of cores	Alya (native) solution[s]	AGMG solution[s]	AGMG setup [s]
72	1.67	0.33	0.37
144	0.86	0.16	0.19
288	0.46	0.12	0.12

AGMG needs 2 steps to be performed; first setup (time reported in last column) and then solution (time reported in third column). For the LES problems being solved, the setup can be performed only once at the beginning of the simulation and it can be reused for all time steps since the pressure matrix remains constant. Thus, this time can be neglected and *only the "AGMG solution" time matters* for the comparison. In the cases with 72 and 144 cores, one sees then that AGMG is more than 5 times faster than Alya's own solver. Further tests on other problems report similar gains; sometimes AGMG is even one order of magnitude faster.

Hence the integration of AGMG in Alya (realized at the state of prototype) is a very fruitful outcome. If the time spent in solving linear systems is cut by a factor of five, it means that, from 30%, the amount of time needed by this part of the algorithm is reduced to less than 8%.

However, the above table suggests some scalability issue with AGMG, as the gain is significantly reduced when using 288 cores. Our efforts to solve this issue are reported below.

Improving the scalability of AGMG in Alya

The standard version of AGMG scales very well in the context it is typically used; that is, to solve linear systems with relatively large load per core in a minute or so. Within Alya, due to the number of solves (above 10^5) for one complete simulation, one needs that each solve lasts the order of a second, which requires to use less than 1 M unknowns per core. Then, maintaining scalability is much more challenging. Indeed, it means that, for a complete solve, only a few tens of milliseconds can be spent in communications. Given the latency of inter-node communications, we are clearly at the limit of the technology. Note here that the fastest the solver, the bigger the challenge is.

To improve AGMG performance in this context, we considered the following. Firstly, we used the massively parallel version of AGMG, as described in [6]. Normally, this specific version is useful when using at least 10 k cores, but, in the context of Alya, it turned out to be faster even with much less cores. Next, we performed some tuning of internal parameters. Normally, AGMG is black box and internal parameters need not to be adapted, performance being little sensitive to them. However, in the specific context of Alya, the change of some default values brought significant behavior improvements.

These enhancements were tested on a sample of 30 linear systems (10 time steps with 3 solves each). The initial problem is defined on 24 cores, with about 3.3×10^5 unknowns each. Then, the tool in Alya that allows to subdivide the elements into elements with half the size is used [3]; each time the divisor is applied the number of nodes of the problem is thus multiplied by 8, and the number of cores is multiplied by 8 as well, so as to maintain roughly constant the load per core. This defines a weak scalability test in which, ideally, the time per linear system solve should be constant. The results are illustrated on Figure 9.

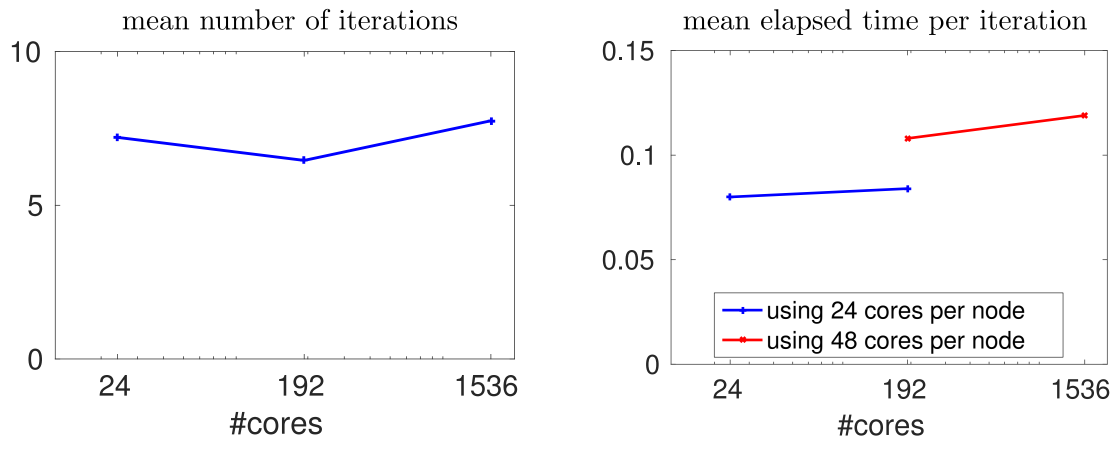


Figure 9: AGMG: weak scalability results with the enhanced version.

The number of iterations is subject to some variations, but there is not significant trend when using more cores. Regarding the time spent for each iteration, we have to distinguish whether one uses half loaded nodes (24 active cores per node) or fully loaded nodes (48 active cores per node, which is the maximum). In the latter case, one uses less nodes and therefore less inter-node communications, but the computation is slower because it is essentially memory bound, hence more active cores on one node means more bottleneck to access node memory. (This is not peculiar to AGMG, but inherent to any computation associated with finite element codes like Alya.) For both curves, one sees some increase of the time as more cores are used. However, this increase is too slight to affect the terms of the comparison with Alya native solver, even if this latter would scale perfectly.

Finally, the mean elapsed time per linear system is reported in Table 7, where a comparison is provided with the standard version of AGMG used for the results reported in Table 6. One sees that improvements are more significant when more cores are used (as expected since now the basis is AGMG massively parallel version [6]). Hence AGMG seems ready to substitute Alya native solver in the LES variant, even when many cores are used. Extensive tests on production runs are planned to confirm this conclusion, with as possible/probable outcome the incorporation of AGMG in the production version of Alya for LES simulations.

Table 7: Mean elapsed time per linear system for AGMG standard and enhanced version.

number of cores	number of nodes per core	standard version	enhanced version
24	24	0.71	0.58
192	24	—	0.54
192	48	0.92	0.70
1536	48	1.39	0.92

References

[1] M. Avila et al. "A Parallel CFD Model for Wind Farms". In: *Procedia Computer Science* 18 (2013). 2013 International Conference on Computational Science, pp. 2157–2166. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.05.386. URL: http://www.sciencedirect.com/science/article/pii/S1877050913005292.

[2] Hadrien Calmet et al. "Large-scale CFD simulations of the transitional and turbulent regime for the large human airways during rapid inhalation". In: *Computers in Biology and Medicine* 69 (2016), pp. 166–180. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2015.12.003. URL: http://www.sciencedirect.com/science/article/pii/S0010482515003881.

[3] Guillaume Houzeaux et al. "Parallel uniform mesh multiplication applied to a Navier–Stokes solver". In: *Computers & Fluids* 80 (2013). Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011, pp. 142–151. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2012.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0045793012001569.

[4] Rainald Löhner et al. "Deflated preconditioned conjugate gradient solvers for the pressure-Poisson equation: Extensions and improvements". In: *International Journal for Numerical Methods in Engineering* 87.1-5 (2011), pp. 2–14. ISSN: 1097-0207. DOI: 10.1002/nme.2932. URL: http://dx.doi.org/10.1002/nme.2932.

[5] *Massively Parallel Hybrid Solver (Maphys)*. URL: https://gitlab.inria.fr/solverstack/maphys.

[6] Y. Notay and A. Napov. "A massively parallel solver for discrete Poisson-like problems". In: *Journal of Computational Physics* 281 (2015), pp. 237–250.

[7] Mariano Vázquez et al. "Alya: Multiphysics engineering simulation toward exascale". In: *Journal of Computational Science* 14 (2016). The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills, pp. 15–27. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2015.12.007. URL: http://www.sciencedirect.com/science/article/pii/S1877750315300521.

3. Eirene

Overview

EIRENE is a classical Monte Carlo transport code that simulates neutral particle (and to a certain extend also charged particle) linear transport, mostly in nuclear fusion applications. It is often coupled iteratively to many integrated fusion plasma transport code systems (e.g. in all EU edge transport code systems), both in 2D and in 3D magnetic configurations. Non-linear particle (and photon) transport problems are also dealt with by iteration. EIRENE is written in Fortran (∼170.000 lines of code) and parallelised using MPI.

Code team:

- Petra Börner (FZJ) (WP5)
- Tamás Fehér (MPG) (WP5)
- Thomas Breuer (FZJ) (WP1)

Benchmark characteristics:

One example of coupling EIRENE to other codes is used in the SOLPS-ITER code package. Here EIRENE is linked together with the B2 code and is called by B2 in each iteration. The following test cases are inspired by a SOLPS-ITER test case, but they use the stand-alone version of EIRENE.
The physical parameters for the test case correspond to the so-called ASDEX Upgrade Plasma with 5 bulk ion species and 59 reactions. The particle number and iteration number is set in a way to have one minute execution time using one node of JURECA. In both test cases six strata are used.

Test case 1, fixed number of particles: In this test the number of particles is fixed in each stratum. A total 630 k particles are calculated in every iteration. There are 30 iterations. The six strata have different numbers of particles, and the CPU time to simulate a stratum also depends on the physical parameters inside the stratum.

Test case 2, fixed execution time: This test specifies that in each iteration 5 seconds of wall-clock time is allocated to following the particles. Each MPI task is assigned to one of the strata and then calculates as many particles as possible in 5 seconds. Twelve iterations are used to reach 1 minute of execution time. The actual execution time is longer because of post processing and communication.
Since the execution time is fixed for test case 2, time is not a good measure of performance. Instead, we should consider the number of particles that are calculated. Some of the metrics in the performance tables will hence use the unit kP/s (thousand particles per second).

Application support

Activity type	WP1 support during performance workshop, Consultancy
Contributors	T. Fehér (WP5, MPG, Germany), T. Breuer (WP1, JSC, Germany), D. Brömmel (WP1, JSC, Germany)

The main support action so far was support during the second benchmarking workshop in Saclay. The results from this workshop have been reported before and are partly mentioned below. This also triggered improvements to EIRENE that were performed via the EUROfusion High Level Support Team (HLST) in Garching, explained later in this section.

A new support activity for EIRENE has been started at the end of 2017. The main developers expressed interest in running large parameter studies that would require a much more scalable code and a new, more scalable architecture to run on for which they were previously targeting JUQUEEN. Since JUQUEEN will be decommissioned, the focus shifted to the JURECA booster system consisting of 1640 compute nodes each with one Intel Xeon Phi Knights Landing (KNL).

Initial tests have been performed investigating the performance of EIRENE on KNL vs. standard Intel CPUs as well as the scalability on the JURECA booster. Along with this, a master student started working on EIRENE in January 2018, focussing on methods employing large parameter studies using EIRENE. His work complements a general restructuring and the continued improvement that is carried out in collaboration with support from EoCoE WP1.

EoCoE benchmark tables

Tables 8 and 9 show the summary of the benchmarks that have been run during the project so far. The two tables each present one of the test cases. The individual columns represent results from the performance evaluation workshop and later measurements to evaluate EIRENE on the booster (KNL) architecture.

Findings from the workshop: Test case 2 has 5.8 seconds of MPI communication time. According to the Scalasca trace analysis, around half of this time is spent in distributing the input data. Some part of the input data does not change between the iterations, and it would be possible to reduce the communication costs of distributing the input data.

The other half of the communication time is summing up the data within the stratum and over all strata. There is a large number of collective communication calls with an average message size of 25 kB. This is due to looping through the columns of large arrays, and doing reductions column-wise. It would be trivial to replace this with a single reduction over 2D arrays.

Even though an Allinea Performance Report suggests that the code is memory bound, the 'Memory vs Compute Bound' metric does not show significant improvements using higher memory bandwidth.

If the vectorization is turned off, then the code performance remains the same, this is in agreement with the Allinea Report. Without FMA operations the code seems to be slightly faster. The difference is very small and is likely only a measurement fluctuation.

Repeated measurements: With the renewed support activity, the metrics have been updated again from previous runs, c.f. the second column in tables 8 and 9. Since the metrics have been changed during the lifetime of the project, the JUBE scripts had to be adapted. This also means not all metrics have a 1:1 correspondence and direct comparisons are made difficult (e.g. the extraction of MPI metrics has changed, some measurements have been removed and others added.). The total runtime nevertheless allows for testing whether

Table 8: Performance metrics for test case 1 on the JURECA cluster and booster system. The first column is the original measurement after the workshop, the last two columns are re-runs to start investigating the KNL performance and latest software stage. Metrics that have been added (removed) during the project are marked with n.m. (o.m.).

		05'2016	01'2018	
	Metric name	Cluster	Cluster	Booster
Global	Total Time (s)	64.4	68	241
	Time IO (s)	0.2	0.24	5.87
	Time MPI (s)	36.0	28.58	134.30
	Memory vs Compute Bound	1.1	1.82	1.01
	Load Imbalance (%)	n.m.	11.01	8.74
IO	IO Volume (MB)	498.3	498.33	502.11
	Calls (nb)	650279	652903	775805
	Throughput (MB/s)	2142.3	2043.87	85.51
	Individual IO Access (kB)	0.9	0.73	0.77
MPI	P2P Calls (nb)	37	37	14
	P2P Calls (s)	0.0	0.42	0.60
	P2P Calls Message Size (kB)	n.m.	3080	1332
	Collective Calls (nb)	191543	191543	131165
	Collective Calls (s)	2.2	13.00	45.13
	Coll. Calls Message Size (kB)	n.m.	22	25
	Synchro / Wait MPI (s)	10.8	11.15	37.67
	Ratio Synchro / Wait MPI (%)	29.9	40.56	28.09
	Message Size (kB)	24.7	o.m.	o.m.
	Load Imbalance MPI	13.4	o.m.	o.m.
Mem	Memory Footprint	151552 kB	166936kB	196584kB
	Cache Usage Intensity	N.A.	0.84	0.63
	RAM Avg Throughput (GB/s)	N.A.	o.m.	o.m.
Core	IPC	N.A.	2.30	1.44
	Runtime without vectorisation (s)	65.8	73	248
	Vectorisation speedup factor	1.0	1.07	1.03
	Runtime without FMA (s)	63.8	73	239
	FMA speedup factor	1.0	1.07	0.99

compiler improvements or updates to the software stack on JURECA had an effect on EIRENE's performance: this seems not to be the case. The small variations in runtime are more likely attributed to variability of different jobs on JURECA. The test cases are likely too short to hide this variability. Two options are possible: change the test cases (with the drawback of loosing continuity in measurements) or extend the JUBE scripts to account for statistics. The latter is a rather involved step and has not been taken by EoCoE.

Investigating the JURECA booster: Tables 8 and 9 also contain a first execution of the code on KNL. Those runs have not been tuned to the architecture, so a performance drop (e.g. for the kP/s metric) is expected at first. We currently observe a slowdown of a ≈ 4 for both test cases when using 24 MPI ranks on the JURECA cluster and 64 MPI ranks on

Table 9: Performance metrics for test case 2 on the JURECA cluster and booster system. For test case 2, the figure of merit is the number of particles calculated per second, given as kP/s (thousand particles/sec). The first column is the original measurement after the workshop, the last two columns are re-runs to start investigating the KNL performance and latest software stage. Metrics that have been added (removed) during the project are marked with n.m. (o.m.).

	Metric name	05'2016 Cluster	01'2018 Cluster	01'2018 Booster
Global	Total Time (s) / Throughput (kP/s)	69.0 / 666	67 / 702	98 / 166
	Time IO (s)	0.1	45.00	6.14
	Time MPI (s)	5.8	7.46	21.09
	Memory vs Compute Bound	1.0	1.81	1.00
	Load Imbalance (%)	n.m.	4.74	12.88
IO	IO Volume (MB)	202.7	202.72	204.29
	Calls (nb)	260403	261404	311524
	Throughput (MB/s)	1842.7	4.50	33.28
	Individual IO Access (kB)	0.9	0.74	0.78
MPI	P2P Calls (nb)	15	15	5
	P2P Calls (s)	0.0	0.18	0.24
	P2P Calls Message Size (kB)	n.m.	4264	2843
	Collective Calls (nb)	76590	76568	52621
	Collective Calls (s)	0.9	7.03	18.76
	Coll. Calls Message Size (kB)	n.m.	21	25
	Synchro / Wait MPI (s)	4.4	6.25	15.70
	Ratio Synchro / Wait MPI (%)	76.5	76.23	74.34
	Message Size (kB)	24.6	o.m.	o.m.
	Load Imbalance MPI	4.5	o.m.	o.m.
Mem	Memory Footprint	154624 kB	339544kB	202232kB
	Cache Usage Intensity	N.A.	0.85	0.66
	RAM Avg Throughput (GB/s)	N.A.	o.m.	o.m.
Core	IPC	N.A.	2.39	1.40
	Runtime without vectorisation (s)	68.0	67	93
	Vectorisation speedup factor	0.99	1.00	0.95
	Runtime without FMA (s)	68.8	75	95
	FMA speedup factor	1.00	1.12	0.97

the JURECA booster, both roughly corresponding to a single node (68 physical cores on a booster node). Note that the 'Memory vs Compute Bound' metric is estimated as on the cluster nodes (i.e. the same number of MPI ranks is distributed over two nodes instead of one, using every other code, in theory freeing memory bandwidth and enlarging L2 cache available to the ranks). One obvious difference is the achieved I/O bandwidth on the cluster and booster part. This is attributed to the early production phase of the booster. But investigating I/O performance revealed inefficient I/O within EIRENE where data is written multiple times (the difference in measure I/O volume and actual data on disk is $\mathcal{O}(10-30)$). While this has no dramatic effect on overall runtime it certainly leaves room

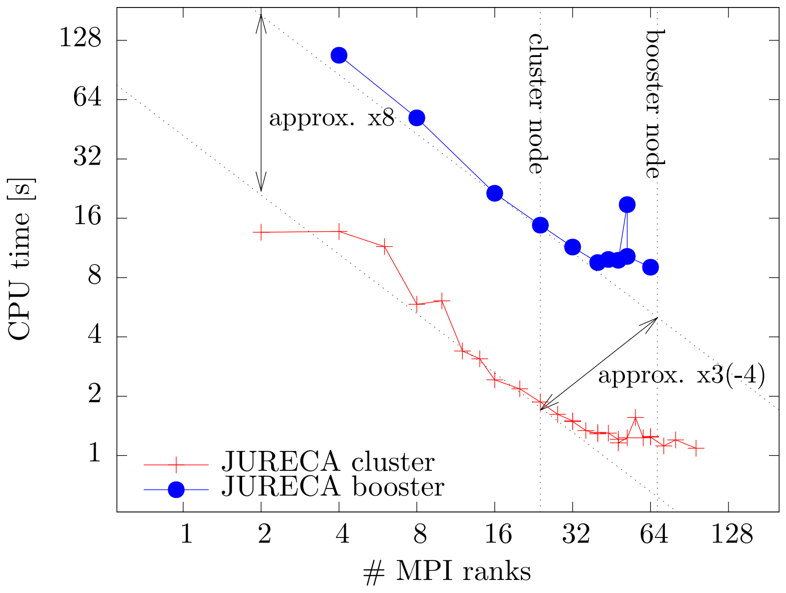


Figure 10: Scalability of EIRENE on the JURECA cluster and booster parts. Shown is the CPU time per step, also indicating the initial (untuned) slowdown between a single core (node) on either part shown.

for improvement.

Please note that the metrics, without statistics, are not to be taken too serious for EIRENE since the observed variations between repeated runs is on the order of the 'observed effects'. In case of I/O, bandwidths between 4 MB/s and 2 GB/s can be seen in the tables. Hence both test cases need to be extended in their runtime.

Since scalability was of special interest, a test case similar to test case 1 was used to test the scaling of EIRENE on the cluster and booster parts, shown in Figure 10. With the used number of strata and overall particle number per rank it was not possible to go beyond the number of MPI ranks shown. We did try and relate performance per core and node of the cluster and booster parts, showing an untuned (i.e. no optimisations for KNL or changes to the source code) slowdown of approximately a factor of 8 and 3–4, respectively. During the new support action we hope to close the gap between the two.

External support

The EUROfusion High Level Support Team (HLST) at Garching was present at the workshop in Saclay and continued supporting the development of Eirene. They have concentrated on the load-balancing issues present in some setups for running EIRENE (with a fixed particle number, similar to test case 1). Here, we quote their main results and refer to their write-up [2, 1] for more details, in particular for the implementation steps.

EIRENE distributes work (in form of strata and particles therein) over the available processing elements (PEs). Strata with a larger particle number will be assigned to more PEs and this balancing will be optimised over time steps according to the work done.
The HLST team discovered and fixed a bug that lead to bad load-balancing in case where

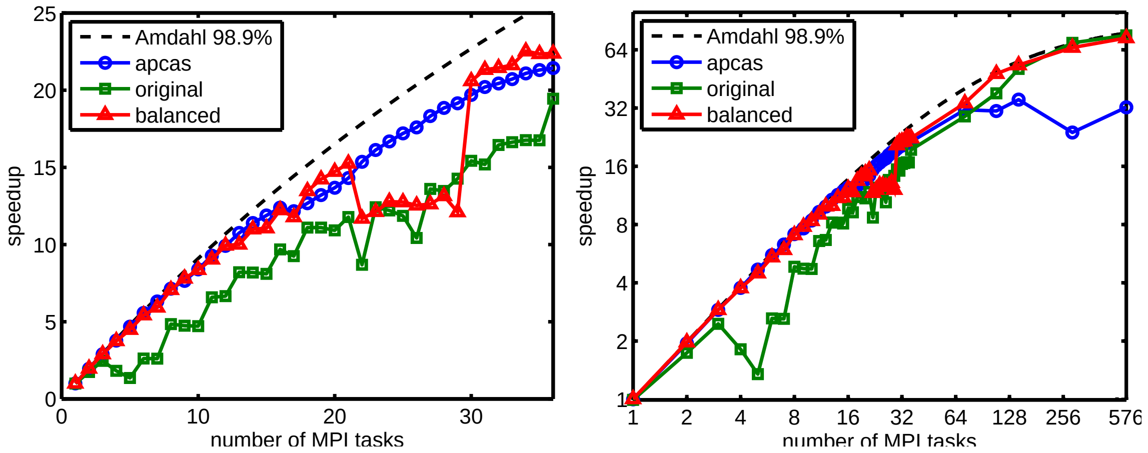


Figure 11: Speed-up of the MCARLO subroutine for different load-balancing strategies for the AUG test case, taken from [1]

the number of PEs was comparable to the number of strata.
Apart from the bugfix, the team also introduced new load-balancing strategies:

- **APCAS**: ALL PEs Calculate All Strata. Here work is distributed evenly among PEs. This also introduces blocking reduction operations. Since all PEs process parts of all strata, overall more communication is necessary which may deteriorate performance with a larger number of PEs.
- **Balanced**: Here non-blocking reductions are used while summing results within strata, thus work does not have to be divided evenly between PEs. Instead work can be divided freely based on measurements of the processing time and overheads. In this case also, some PEs may work on multiple strata.

The performance impact has been studied on MARCONI where each node has 2x18 cores. Test cases called 'AUG3' and 'ITER2' used a fixed particle and iterations number. Thus keeping the problem size constant, the number of MPI tasks was varied and the execution time measured. The HLST noted several performance drops with certain numbers of tasks for some of the load-balancing strategies and suggested this may be related to the MPI library. Figure 11 shows this is the case for the 'AUG3' test case and the 'balanced' load-balancing, while Figure 12 has such an effect for 'APCAS' and the ITER case.

Any achieved improvements will depend on the exact setup of the runs performed with EIRENE. However, ITER users report a routine usage of these improvements for their scientific production of SOLPS-ITER without quantifying the overall impact in runtime.

References

[1] Tamás Fehér. *Intermediate report for July–September 2016. HLST project SOLPSOPT*. 2016.

[2] R. Hatzky et al. "HLST Core Team Report 2016". In: *EUROFUSION WPISA-REP* (16 2016). URL: `http://www.euro-fusionscipub.org/wp-content/uploads/eurofusion/WPISAREP17_18574_submitted.pdf`.

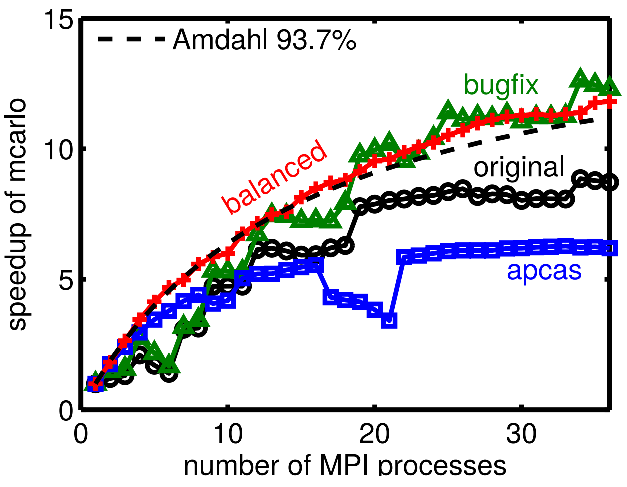


Figure 12: Speed-up of the MCARLO subroutine for different load-balancing strategies for the ITER test case, taken from [1]

4. Esias

Overview

ESIAS stands for Ensemble for Stochastic Integration of Atmospheric Simulations.

It is a coupled ensemble implementation of Weather Research and Forecasting Model (WRF) and European Air Pollution and Dispersion Inverse Model (EURAD-IM) for short to medium range probabilistic forecasts and emission parameter estimation using Monte Carlo and Variational Data assimilation techniques.

WRF is a state-of-the-art mesoscale numerical weather prediction system which is used extensively for research and operational real-time forecasting at numerous public research organizations and the private sector throughout the world and is open to the public. It offers various sophisticated physics and dynamics options.

EURAD-IM is a fully adjoint chemistry transport model on the regional scale for chemical species and aerosols which is used for both, operational air quality forecasts and research applications. A main feature is the joint initial value and emission factor optimization using four dimensional variational data assimilation.

Code team:

- Sebastian Lührs (FZJ) for WP1
- Jonas Berndt (FZJ) for WP2

Performance metrics

The benchmark setup contains a random simulation period of 6 hours with 240x240x24 grid points as a typical size. For benchmarking, solely 2 ensemble members run in parallel (instead of the order 1000 for production runs, would be too computational intensive for benchmarking). No particle filtering is performed due to the small ensemble size. 1024 processors on JUQUEEN are used. Parallel NetCDF is used. In Table 10 the direct comparison between the initial code state on JUQUEEN for this particular benchmarking case together with a second run done after the improving the node level performance is shown.

For a second setup the same benchmark case was chosen, but the number of ensemble members was increased together with the number of used processors to validate the weak scaling behavior. For this run 256 processors per ensemble member were selected and three individual runs for 2, 4 and 16 members were performed on JUQUEEN. Table 11 shows the extracted metrics.

Support

Activity type	Consultancy
Contributors	Sebastian Lührs (WP1, JSC, Germany), Jonas Berndt (WP2, IEK8, Germany)

The main support activity for ESIAS within the framework of EoCoE took place in the

Table 10: Performance metrics for Esias on the JUQUEEN HPC system.

	Metric name	metrics_O2.json	metrics_O3.json
Global	Total Time (s)	259.46	199.71
	Time IO (s)	28.53	27.42
	Time MPI (s)	150.01	132.33
	Memory vs Compute Bound	N.A.	N.A.
	Load Imbalance (%)	31.03	31.36
IO	IO Volume (MB)	3570.93	3570.93
	Calls (nb)	63594	63594
	Throughput (MB/s)	125.16	130.24
	Individual IO Access (kB)	118.42	118.45
MPI	P2P Calls (nb)	135267	135267
	P2P Calls (s)	70.25	57.07
	P2P Calls Message Size (kB)	15	15
	Collective Calls (nb)	6170	6170
	Collective Calls (s)	21.93	18.35
	Coll. Calls Message Size (kB)	14	14
	Synchro / Wait MPI (s)	85.89	68.73
	Ratio Synchro / Wait MPI (%)	48.05	42.20

context of WP2 with additional guidance by WP1 especially in context of the usage of the performance evaluation tools.

The main outcome of this particular support activity is presented within the separate deliverable D2.2 "Ultra large meteorological ensemble" [1] as well in the performance evaluation deliverable D1.17 "Application Performance Evaluation" [2].

The overall computing performance could be improved with the help of adapted compiler options. Here mainly the WRF part of ESIAS was adapted to the JUQUEEN system capabilities. This allows a overall runtime improvement in the sense of a a factor 1.5 to 2. A comparison of the performance metrics for the non optimized ESIAS execution and the optimized version is shown in Table 10.

In addition within WP2 the internal stochastic pattern computation was parallelised. This was needed to finally reach a acceptable execution time on JUQUEEN due to its specific architecture with its low per core performance. The routine itself could be improved by a factor of 5 up to 10 in the sense of computing time.

The overhead of the ensemble approach was evaluated as well. For a small number of ensemble members the scaling behavior was analyzed with the help of the performance metrics, as shown in Table 11. Larger number of ensemble members were also tested in the context of D2.2. Overall there is a growing datasize for additional ensemble members, which is compensated by the growing I/O bandwidth when using more computing elements of JUQUEEN. On the other site the overall MPI time increased, while the total time kept quite stable. This behaviour is mainly influenced due to the metric calculation and the selected benchmark case, which was just copied multiple times. While the number of computing elements increased, there is one process defining the critical path of the application. This path stays mostly unchanged even if there are new members involved.

Table 11: Performance metrics for increasing number of Esias ensemble members on the JUQUEEN HPC system.

	Metric name	2 member	4 member	16 member
Global	Total Time (s)	268	270	279
	Time IO (s)	20.70	21.23	22.13
	Time MPI (s)	111.92	119.21	174.22
	Memory vs Compute Bound	N.A.	N.A.	N.A.
	Load Imbalance (%)	24.39	26.23	36.40
IO	IO Volume (MB)	3244.71	6489.41	25957.65
	Calls (nb)	38506	77002	307978
	Throughput (MB/s)	156.73	305.61	1172.94
	Individual IO Access (kB)	147.82	147.83	147.85
MPI	P2P Calls (nb)	133050	133050	133050
	P2P Calls (s)	57.52	58.88	99.64
	P2P Calls Message Size (kB)	19	19	19
	Collective Calls (nb)	6170	6170	6170
	Collective Calls (s)	10.91	10.77	16.67
	Coll. Calls Message Size (kB)	16	30	117
	Synchro / Wait MPI (s)	62.80	68.95	118.29
	Ratio Synchro / Wait MPI (%)	47.13	48.91	60.20

Due to the metric average calculation the overall MPI delay increases, as more members have to wait for the critical path.

References

[1] Hendrik Elbern, ed. *D2.2 Ultra large meteorological example.* URL: `http://www.eocoe.eu/sites/default/files/results_files/d2.2-m18-676629_ultralarge_meteorological_ensemble.pdf`.

[2] Matthieu Haefele, ed. *D1.17 Application Performance Evaluation.* URL: `http://www.eocoe.eu/sites/default/files/results_files/d1.17.pdf`.

5. Gysela

Code team:

- Matthieu Haefele (MdlS) for WP1
- Guillaume Latu (CEA) for WP5

Metrics and Performance Report

Tiny case characteristics:

Domain size	64 x 128 x 64 x 31 x 1
Resources	part of 1 node on JURECA (16 cores)
IO details	Checkpoint written every 8 steps instead of 100, more often than production
Type of run	development run

Small case characteristics:

Domain size	512 x 256 x 128 x 60 x 2
Resources	4 nodes on JURECA (96 cores)
IO details	Checkpoint written every 16 steps instead of 100, more often than production
Type of run	development run

Medium case characteristics:

Domain size	512 x 256 x 128 x 60 x 32
Resources	43 nodes on JURECA (1024 cores)
IO details	Checkpoint written every 16 steps instead of 100, more often than production
Type of run	production run

Table 12: Performance metrics for Gysela on the JURECA HPC system (Tiny case) - status at the beginning of the project.

	Version	2015-11-24
Global	Total Time (s)	31.2
	Time MPI (s)	2.37
	Memory Footprint (B)	1.8 GB
Node	Time Spent in OpenMP (s)	27.41
	Time spent in OpenMP barriers (s)	0.40
	Time spent outside OpenMP regions (s)	1.76
Core	IPC	1.7
	Runtime without vectorisation (s)	68.0
	Runtime without FMA (s)	36.9

During the course of the project, the code has been extensively upgraded in order to shorten restitution time (in parallel to the development of new physics modules). This is quite clear in the Table 13 that summarizes on a small test case that the execution time decreases with successive versions of the code. We will go through the various improve-

Table 13: Performance measures for Gysela on the JURECA HPC system (Small case) throughout the course of the project.

Gysela Version	2015-11-24 splines	2016-02-25 splines	2017-03-14 splines	2018-08-07 splines	2018-08-07 lagrange $8th$
threads per node	32	32	48	48	48
threads per process	8	8	12	12	12
Total Time (s)	1588	1461	1053	727	525
Memory Footprint (B)	80 GB	80 GB	84 GB	85 GB	85 GB
Runtime without vectorisation (s)	1564	1529	1110	801	635

ments that have been performed in the remainder of this section. GYSELA is a 5D gyrokinetic global code for simulating flux-driven plasma turbulence in a tokamak. The benchmark test case is based on a semi-Lagrangian scheme solving 5D gyrokinetic ion turbulence in tokamak plasmas. The GYSELA code is mainly written in Fortran90 and parallelised using both MPI and OpenMP. The code was built and run on the JURECA cluster with Scalasca/Score-P (profile and trace) measurements provided for examination. To achieve an initial performance analysis, the code was built using Intel MPI 5.1 and Intel 15.0.3 compilers, and instrumented with Score-P 1.4.2 as part of Scalasca 2.2.2. Part of the information contained in this paragraph have been extracted from a report written by the PoP center of excellence (https://pop-coe.eu/).

Two execution traces were collected on JURECA each running 128 MPI processes with 8 OpenMP threads per process considering the `Medium case`. One execution on 43 compute nodes had 3 MPI processes per node and therefore a dedicated core for each thread, whereas the other for comparison used hyper-threading with 6 MPI processes per node on 22 compute nodes. Program spent most of its time in two routines 80% in `blz_predcorr`, 15% in `diagnostics_compute`. Main equations (Vlasov and Poisson) are solved in `blz_predcorr` and post-processing of physical values and export on disk are done in `diagnostics_compute`. Most of the computations are tackled within OpenMP regions. MPI communications represents less than 2% of execution time inside `blz_predcorr`. For conventional production runs (number of cores is below 16 000 cores) the MPI overheads and MPI parallel imbalance are not an issue. We will not investigate here large configurations with high number of cores (32k and more) and will assume that MPI communication costs and parallel domain decomposition are not a major bottleneck.

80% of GYSELA total time in `blz_predcorr` is computation, 71% of which is in three OpenMP parallel regions with significant load imbalance. Work should be done to improve this, especially whenever hyper-threading is activated because it reinforces the imbalance. Furthermore, within `blz_predcorr`, 2D advection operator located in `advec2d_bsl.F90` shows specific problems: it is notable that the OpenMP synchronisation cost is particularly high for half of the OpenMP threads for the MPI rank straddling the two processors on each compute node. This is due to the number of threads per MPI process chosen (8) that does not fit very well on a node that has 2 sockets of 12 cores. Something has to be done to avoid MPI processes straddling the 2 sockets (this issue will be tackled in support activities).

Efficiency of vectorisation should be investigated. One can expect better speedup than a factor 2 with (31.2s) or without vectorisation (68s). The vectorization issue will also be tackled in support activities.

On large production runs, IO becomes an issue because checkpoint file size represents 100 GB up to 1 TB to be written down several times per run. HDF5 format is used up to now, but other strategy can be looked at in order to improve performance.

We have investigated the most intensive computation parts of the code with Paraver set of tools (www.bsc.es/paraver). These tools are based on traces capturing the detailed behavior of the different MPI processes and threads along time. Calls to the MPI and OpenMP runtime can be enriched with hardware counters, so we were able to measure the instructions and cycles for each computation region. In the next section we will show how the use of the Paraver tool helped to efficiently put into place simultaneous multi-threading in Gysela.

Application support

Activity type	Consultancy or WP1 support
Contributors	Brian Wylie (WP1, Germany), Judit Gimenez (WP1, Spain), Guillaume Latu (WP5, France), Julien Bigot (WP1, France), Corentin Roussel (WP1, France), Benedikt Steinbusch (WP1, Germany), Chantal Passeron (WP5, France), Wolfgang Frings (WP1, Germany), Sebastian Lührs (WP1, Germany)

Direct benefits of SMT

To evaluate SMT, we choose a domain size of $N_r \times N_\theta \times N_\varphi \times N_{v_\parallel} \times N_\mu = 512 \times 256 \times 128 \times 60 \times 32$ in this section. Due to GYSELA internal implementation choices, we are constrained to choose, inside each MPI process, a number of threads as a power of two. Let us remark, that the application performance increases by avoiding very small power of two (*i.e.* 1, 2). Haswell node that we target are made of 24 cores. That is the reason why we choose to set 8 threads per MPI process for the runs shown hereafter. This configuration will allow us to compare easily an execution with or without SMT activated.

In the following, the deployment with 3 MPI processes per node (one compute node, 24 threads, 1 thread per core) is checked against a deployment with 6 MPI processes per node (one compute node, 48 threads, 2 threads per core, SMT used). Strong scaling experiments are conducted with or without SMT, timing measurements are shown in Table 14. Let us assume that processes inside each node is numbered with an index n going from 0 to 2 without SMT, and $n = 0$ to 5 whenever SMT is activated. For process n, threads are pinned to cores in this way: logical cores id from $8\,n$ up to $8\,n + 7$.

The different lines show successive doubling of the number of cores used. The first column gives the CPU resources involved. The second and third columns highlight the execution time of mini runs comprising 8 time steps (excluding initialization and output writings): using 1 thread per core (without SMT), or using 2 threads per core (with SMT support). The last column points out the reduction of the run time due to SMT comparing the two previous columns. As a result, the simultaneous multi-threading with 2 threads per core gives a benefit of 21% up to 28% over the standard execution time (deployment with one thread per core). While an improvement is expected with SMT, as already reported for other applications this speedup is quite high for a HPC application.

Table 14: Time measurements for a strong scaling experiment with SMT activated or deactivated, and gains due to SMT.

Number of nodes/cores	**Exec. time (1 th/core)**	**Exec. time (2 th/core)**	**Benefit of SMT**
22/ 512	1369s	1035s	**-24%**
43/1024	706s	528s	**-25%**
86/2048	365s	287s	**-21%**
172/4096	198s	143s	**-28%**

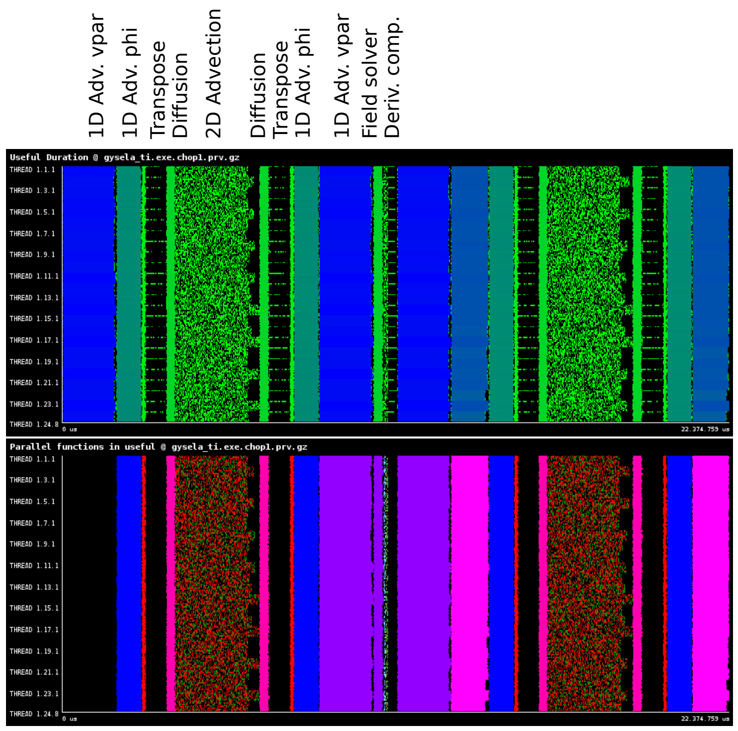


Figure 13: Snippet of a run with 2 threads per core (SMT), Top: Paraver useful duration plot, Bottom: Parallel functions plot.

Within Paraver, we observe that for each intensive computation kernel the number of instructions per cycle (IPC) cumulated over the 2 threads on one core with SMT is always higher than the IPC obtained with one thread per core without SMT. For these kernels, the cumulated IPC is comprised between 1.4 and 4 for two threads per core with SMT, whereas it is in the range of 0.9 up to 2.8 with one thread per core without SMT. These IPC numbers should be compared to the number of micro-operations achievable per cycle, 4 on Haswell. Thus, we use a quite large fraction of available micro-operation slots. Two factors explain the boost in performance with SMT. First, SMT hides some cycles wastes due to data dependencies and long latency operation (*e.g* memory accesses). Second, SMT enables to better fill available execution units. It provides a remedy against the fact that, within a cycle, some issue slots are often unused.

Optimizations to increase SMT gain

The Paraver tool gives us the opportunity to have a view of OpenMP and MPI behaviors at a very fine scale. The visual rendering informs rapidly the user of an unusual layout and therefore hints to look on some regions with unexpected patterns. On the Fig. 13 is plotted a snippet of the timeline of a small run with SMT (2 threads per core, 24 MPI processes, 8 threads per MPI process, meaning 4 nodes hosting 48 threads within each node). We can extract the following information:

1. The 2D advection kernel (first computationally intensive part of the code) is surprisingly full of small black holes.

2. There are several synchronizations during this timeline between MPI processes that are noticeable. As several moderate load imbalances are also visible, a performance penalty can be induced by these synchronizations. See for example 2D advection and Transpose steps (Useful duration plots), there is much black color at the end of these steps. This is due to final MPI barriers. Nevertheless the impact is relatively low in this reduced test case because the tool reported a parallel efficiency of 97% over the entire application indicating that only 3% of the iteration time is spent on the MPI and OpenMP parallel runtimes. The impact is stronger on larger cases, because load imbalance is larger.

3. The transpose steps show a lot of black regions (threads remaining idle). At the end of the phase, all the ranks are synchronized by the MPI_Barrier. Checking the hardware counters indicate the problem is related with a different IPC where the fast processes are getting twice the IPC of the delayed ones. This behavior illustrates well that SMT introduces heterogeneity of the hardware that should be handled by the application even if the load is well balanced between threads.

4. At the end of 2D advection step, a serrated form is noticeable. All the processes that straddle two different sockets are slowed down a little bit.

These inputs from the Paraver visualization helped us to determine some code transformations to make better use of unoccupied computational resources. The key point was to point out the cause of the problem, the improvements were not so difficult to put into place. The upgrade are described in the following list. The Table 15 and Fig. 14 exhibits associated measurements.

1. The 2D advection kernel is composed of OpenMP regions. There is mainly an alternation of two distinct OpenMP kernels. The first one fills the input buffers to prepare the computation of 2D spline coefficients for a set of N poloidal planes (corresponding to different $\varphi, v_{\|}$ couples). The second kernel computes the spline coefficients for the same N poloidal planes and performs the advection itself that encompasses an interpolation operator. Yet, there is no reason for having two separate OpenMP regions encapsulated in two different routines, apart from historical ones. Thus, we decided to merge these OpenMP regions in a single large one. This modification avoids the overheads due to entering and leaving the OpenMP regions multiple times. Also the implicit synchronization at the beginning and end of each parallel region are removed. Thus, avoiding synchronization leads to a better load balance by counteracting the imbalance originating mainly from the SMT effects.

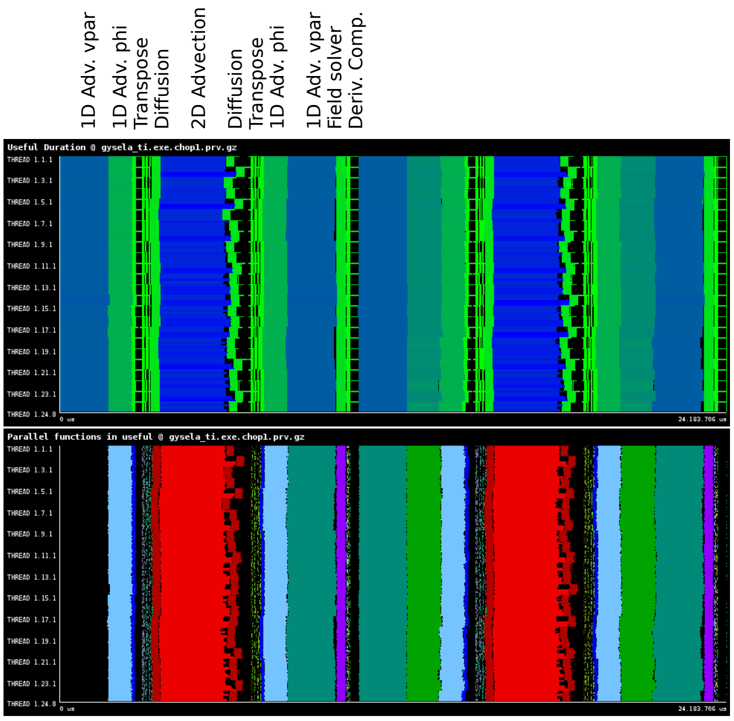


Figure 14: Snippet of a run with 2 threads per core (SMT), after optimizations are done, Top: Paraver useful duration plot, Bottom: Parallel functions plot.

2. Some years ago, with homogeneous computing units and resources, the workload in GYSELA was very balanced between MPI processes and inside them, between threads. Thus, even if some global MPI barriers were present within several routines, they induced negligible extra costs because every task was executed synchronously with the others. In latest hardware, there is heterogeneity coming from cache hierarchy, SMT, NUMA effects or even Turbo boost. The penalty due to MPI barriers is now a key issue, and thread idle time is visible on the plot. We removed several useless MPI barriers. As a result, we see for example in Fig. 14 that, now, diffusion is sandwiched between the transpose step and the 2D advection, without global synchronization.

3. The transpose step is compounded of three sub-steps: copy of data into send buffers, MPI non-blocking send/receive calls with a final wait on pending communications, copy of receive buffers into target distribution function. On the Fig. 13, it is worth noticing that only the first thread of each MPI process is working, *i.e.* only the master thread is performing a useful work. To improve this, we added OpenMP directives to parallelise all the buffers' copies. This modification increases the extracted memory bandwidth and the thread occupancy. On the Fig. 14, the bottom plot shows that the transpose step is now partly parallelised with OpenMP.

Thanks to these upgrades, there is much less black (idle time) in Fig. 14 compared to Fig. 13. Still, MPI communications induce idle time for some threads in the transpose step and in the field solver. This can not be avoided within the current assumptions done in GYSELA. Table 15 also illustrates the achieved gain in term of elapsed time. If one compares to Table 14, the timings are reduced with one or two threads per core. Comparing one against two threads per core, the SMT gain is still greater than 20% (almost the same statement as before optimization). Now, if we cumulate the gain resulting from SMT and from the optimizations, we end up with a net benefit on execution time of 32% up to 38% depending on the number of nodes.

Table 15: Time measurements and gains achieved after optimizations that remove some synchronizations and some OpenMP overheads.

Number of nodes/cores	Exec. time (1 th/core)	Exec. time (2 th/core)	Benefit of SMT	Benefit vs. Table 1
22/ 512	1266s	931s	**-26%**	**-32%**
43/1024	631s	474s	**-25%**	**-33%**
86/2048	320s	239s	**-25%**	**-34%**
172/4096	164s	124s	**-25%**	**-38%**

Much of this work has been published in a paper `https://doi.org/10.1145/2929908.2929912` entitled [*Benefits of SMT and of Parallel Transpose Algorithm for the Large-Scale GYSELA Application*, Latu & al., 2016]

In addition to these upgrades, we have modified the code to remove the constraint of having a power of two concerning the number of threads within an MPI process. This has been a bit of work to modify some algorithms, but the reward is that we can now avoid MPI processes that straddle two different sockets. Typically we now put 2 MPI processes per socket and 2 threads per core for production runs. Avoiding the straddling allows us for an extra saving of 5% on the total execution time.

To conclude, the use of SMT has decrease run times by 24%, whereas additional optimization done in the framework of this application support brought an additional 16% of reduction. Then, this optimization work consitutes a strong benefit for the user of GYSELA application. Modifications has been included in the production code in the Q1 of 2016. The positive impact is illustrated by the third column (2017-03-14) of Table 13. Compared to the first column (2015-11-24), restitution time has been reduced by 30% on this small case (this also the case on larger configurations). This work on SMT and load-balancing contribute largely to this improvement. The amount of core-hours consumed in 2016 on machines that have Hyper-threading/SMT activated was 40 millions for GYSELA code. One can estimate that this application support has saved at least **6.4** millions of core-hours in year 2016 and direct SMT use has saved **9.6** millions of core-hours within the same year.

Auto-tuning

Portability of performance with static auto-tuning. Within a single HPC application, multiple aims concerning the source code should be targeted at once: performance, portability (including portability of performance), maintainability and readability. These are very difficult to handle simultaneously. A solution is to overhaul some computation intensive parts of the code in introducing well defined kernels. BOAST (Bringing Optimization Through Automatic Source-to-Source Transformations, developed by INRIA project-team CORSE, part of WP1) is a metaprogramming framework to produce portable and efficient computing kernels. It offers an embedded domain specific language (using ruby language) to describe the kernels and their possible optimization. It also supplies a complete runtime to compile, run, benchmark, and check the validity of the generated kernels. BOAST has been applied to some of the most computation intensive kernels of GYSELA: 1D and 2D advection kernels. It permitted to gain speedup from 1.9× up to 5.7× (depending on the machine) on the 2D advection kernel which is a computation intensive of the code. Furthermore, BOAST is able to generate AVX-512 instructions on INTEL Xeon Phi KNL in order to get high performance on this architecture. A specific point to take into account with this approach is to handle the integration of Ruby code within the production/legacy code. This optimization work saves in average 8% of computation time over the total execution time, integration into production code has been carried out in 2017. This activity was part of the CEMRACS school where WP1 and WP5 people have met (`http://smai.emath.fr/cemracs/cemracs16`). A proceeding paper will be published in 2018 that describes this auto-tuning approach for the GYSELA application [*Building and auto-tuning computing kernels: experimenting with BOAST and StarPU in the GYSELA code*, Bigot & al, 2018, ESAIM proc., to appear]. This work contributed to the improvements in restitution time brought by Lagrange interpolators of order 8^{th} - shown in the two last columns of Table 13.

Portability of performance with dynamic auto-tuning. Another option for performance portability is to use auto-tuning at runtime. Compared with static auto-tuning, this dynamic approach incurs more overhead at runtime but it is able to leverage information that only becomes available at execution. The result of these different compromises is that the dynamic approach makes sense at a coarser grain than the static approach and that is therefore interesting to combine both. In the case of GYSELA, we have implemented this approach based on the StarPU runtime developed in Inria project-team STORM (related to WP1 contribution). The whole 2D advection of which the kernel optimized with BOAST is a part has been ported to use the native StarPU API for parallelisation instead of OpenMP. This new approach makes it possible to express parallelism at a grain that would be complex to express in the previously used OpenMP fork-join model and thus improve cache usage. StarPU should also improve performance portability by letting execution choices be made in the StarPU scheduling plug-in rather than in the application code. The scheduling plug-in can take into account informations about the available hardware and can even be changed for different executions. Early evaluations on the Poincare cluster have demonstrated a 15% speedup on a realistically sized case compared to the version using OpenMP fork-join. This work has also been described in the proceeding paper of the CEMRACS school [*Building and auto-tuning computing kernels: experimenting with BOAST and StarPU in the GYSELA code*, Bigot & al, 2018, ESAIM proc., to appear]. It has not been integrated in the production code.

Extending Input/Ouput capabilities for checkpoint

On large configurations, one significant cost is caused by writing and reading checkpoint/restart files. To access new physics as kinetic electrons, computational domain is increased a lot and it requires much more resources and I/O capabilities. A specific support has been asked to EoCoE to improve GYSELA on this matter. The main goals were to reduce these Input/Output expenses and add new capabilities in order to: have access to several libraries to handle different checkpoint file format to perform comparisons, improve the maintenance of the code dedicated to the coupling with I/O libraries, add a new feature to GYSELA in order to be able to restart from files generated with another domain decomposition and/or grid size.

This activity has began in Q4 2016 and ended in Q4 2017. The following tasks were done: 1) Incorporate calls to SIONlib to write/read restart files in GYSELA, 2) Allow for GYSELA restarting with a different MPI domain decomposition that was used for checkpointing (with HDF5), it implies that the code should handle the reading of 5D distribution function whatever the number of files used for storing the distribution function, 3) PDI is a Parallel Data Interface that decouples parallel code from I/O libraries, some features available in PDI have been integrated in GYSELA.

Vectorizing and increasing numerical intensity

We also focused on the vectorization of the main computation kernels, enhancing the compiler ability to generate efficient code, and upgrading interpolation schemes in order to increase numerical intensity and shorten restitution time. The current generation of the Xeon Phi Knight Landing (KNL) and recent INTEL chips (Skylake) provides a highly multi-threaded environment on which regular programming models such as MPI/OpenMP can be used as we do. These hardware offer both large memory bandwidth and computing resources and are available on some computing facilities we have access to. Efficiently exploiting SIMD vector units is one of the most important aspects in achieving high performance for application codes that are not strictly memory bounded on these architectures. Hereafter, we describe a set of different techniques that improved performance of GYSELA code: loop splitting, inlining, grouping set of LU solve, removing conditionals and some loop nests, changing a key numerical scheme (Lagrange interpolation instead cubic splines). As a result, KNL and Skylake execution times have been largely reduced. But good news is that this effort has also permitted to shorten resitution time on older architectures as well. This achievement is presented in the proceeding paper of the WAMCA 2018 workshop [*Scaling and optimizing the GYSELA code on a cluster of many-core processors*, Latu & al, 2018, IEEE proc., to appear].

SIMD & KNL One option to improve single core performance is based on vector registers and SIMD instructions. SIMD operations exhibit parallelism proportional to the length of vector registers. Increasing vector length thus offers the opportunity to achieve speedups in codes through more SIMD parallelism. Current vector size in Intel KNL and Skylake is 512-bit. One possible way is for the developer to transform the source code for the compiler to generate a proper executable with respect to vectorization. Most of the optimizations targeting vectorization improve performance both on KNL and on general-purpose multi-core architectures (as Haswell, Broadwell, Skylake). Intel KNL is a standalone many-core processor. It has many features: a large number of threads, large vector units, multiple memory tiers, large memory bandwidth (MCDRAM). The chip provides up to 72 cores grouped in tiles, four threads per core, two levels of cache.

Helping the compiler The first steps in adapting our code were to minimize the cost of function calls using inlining, to avoid or move away conditionals from the innermost loops and to reformulate some mathematical operations to ease the compiler's work. For example, we added inline functions through the `!$dir force inline` directive and by moving the function declarations in header files. As a metter of fact, some algorithms and/or programming styles inhibit vectorization. Some of the requirements to help the vectorization process are: (i) the vectorized loop should be the innermost loop of a nest, (ii) there should be no I/O nor function calls (apart from math functions) inside those

loops, (iii) loop-carried and complex data dependencies should be avoided, (iv) the control flow should be uniform (exceptions exist but one should consider removing branches at first) and array notations should be promoted instead of pointers. A conditional branch is a control hazard, it introduces additional instructions, it can lead to pipeline stalls that can compromise the efficiency of the Vector Processing Unit.
The SIMD instruction sets of processors tends to be less general than the scalar ones. Specialized domain-specific operations are included, many operations are available only for some data types, and a high-level understanding of the computation is often required in order to take advantage of them. In order to avoid going to assembly, the developer has to transform the code so that the auto-vectorization of the compiler achieve good optimizations. Some standard techniques we have used include: 1) precompute and store reciprocals (to avoid divisions), 2) reformulate some mathematical expressions and remove temporary variables for simpler data dependencies analysis, 3) introduce small vectors as local variables (typically of the size of register width or a bit larger) together with strip-mining 4) add explicit vectorization directives as `!$dir simd`.

Alternative high-order interpolation High-order methods require more floating point operations per degree of freedom than low-order methods. One could expect high-order to slow down applications, but execution time is not directly proportional to computational cost. Increased operation efficiency (*e.g* through good vectorization) can compensate the increase of computational cost. In addition, high-order schemes usually increase the achievable accuracy. Furthermore, the computation intensity for data in cache is large for high-order methods and this fits well with the idea that "FLOPS are almost free" in the Exascale landscape while costs associated to data accesses should increase. In this context, we evaluated the benefits of 1D high-order Lagrange interpolations instead of cubic splines. Lagrange polynomials of degree 5 and 7 were selected. These Lagrange polynomials provide close accuracy compared to cubic splines within GYSELA runs. Practically, splines require a set of coefficients that are computed prior to the interpolations. This step involves additional data moves, storage for the coefficients, but also extra operations (*i.e.* small LU systems to be solved) that the compiler have difficulties to vectorize. With the Lagrange approach, the compiler is able to well vectorize the simple mathematical formulas. The number of multiplications and additions is larger for Lagrange than for splines, but as the vectorization is effective with Lagrange, the computational overhead is cleared.

Cache-friendly one-strided advections Contiguous memory access patterns fit well with the SIMD approach. Many SIMD operations can reference aligned unit-stride vectors in-memory as part of the instruction, thus avoiding separate load/stores. In other words, contiguous accesses permit to save extra and possibly inefficient gather/scatter operations or strided load/store. To access memory efficiently, one has also to minimize indirect addressing, and to align data to 64-byte boundaries on both KNL and Skylake. Some data layout transformations may help in that regard. In several algorithms, we have avoided long-strided accesses along the last contiguous dimension of multi-dimensional arrays. Copies are performed to switch the storage within the muti-D array before and after computation intensive kernels. During the copy we mix the slowest varying index with the fastest varying one in order to benefit from fast reads from the main memory. Computations are then performed on the 2D tile, typically in L2 cache, with good cache locality ensured. These modifications improve the quality of auto-vectorization and ensure cache-friendliness (use of L2 cache is improved and the TLB is less stressed by long-strided access).

Vectorized LU solver Several routines of LAPACK can work with multiple right-hand-sides, and `dpttrs` is one of them. It solves a tridiagonal system $AX = B$ where X and B are general matrices and A is positive definite real symmetric. The computations performed by such routine is conceptually easy to transform into a set of SIMD instructions as the same steps are applied at the same time to different right-hand-sides stored into small vectors. This can be achieved organizing the storage of the right-hand-sides in memory. The developer has to carry out a data layout transform that may introduce a minor overhead, but it allows for a very efficient and vectorized implementation of the solve in the `dpttrs` routine. We modified our code to benefit from a vectorized LU solver. This solution has been incorporated into two operators: heat source, diffusion.

Additional optimizations *Loop fission* (also known as loop distribution) consists in splitting a single loop into more than one, generally to remove or simplify dependencies. It attempts to build simpler loop bodies (part of the original one) while keeping the same index range. This simplifies dependency analysis for the compiler and isolates the parts of the loop that inhibit vectorization (in addition this reduces the pressure on the vector registers). This technique has been applied in several innermost loops of the code.
We introduced *strip-mining* technique within some GYSELA's operators. We also introduced small vectors declared as local variables. Their *size* was set accordingly with the strip-mining segment, it enables us to get a better SIMD-encoding from the compiler.

Benchmark The best configuration that we have identified on KNL node is 4 MPI processes of 32 threads within a node of 68 cores (roughly two threads per core with the hyperthreading). The memory mode on KNL was set to cache and cluster mode to quadrant. On Marconi Broadwell and Skylake nodes, the hyper-threading support was unavailable, thus we imposed one thread per core and one process per processor (*i.e.* two processes per node). Let us summarize all benefits arising from the contribution of the current subsection entitled "Vectorizing and increasing numerical intensity". Table 16 gives a wrap-up of the gains provided by techniques described above. The lines of the table focus on different operators of GYSELA. Execution times are shown in seconds for a case that fits into a single node (all cores are used). In percentage, the gain over the previous version is displayed. The global execution times are reduced. It turned out that all architectures took advantages of the changes.

Table 16: Breakdown of timing (in s) for a run that fits into a node (all cores used). In parentheses, improvement compared to previous version. Domain size $256 \times 128 \times 64 \times 64 \times 1$

Steps \ Hardware	Broadwell	KNL	Skylake
advec1D in vpar	12.0 (-79%)	11.1 (-86%)	6.2 (-86%)
advec2D (r,theta)	7.2 (-82%)	6.9 (-84%)	4.1 (-86%)
comm. transpose	30.9 (-26%)	13.2 (-46%)	15.5 (-53%)
heat source	4.3 (-62%)	3.6 (-84%)	2.3 (-72%)
diffusion in theta	4.1 (-65%)	3.3 (-70%)	2.6 (-68%)
diagnostics	12.8 (-65%)	12.0 (-48%)	7.8 (-78%)
...			
Total	111 (-56%)	89 (-68%)	65 (-68%)

In this subsection, we have shown manual transformations that can be applied to overcome compiler limitations and that allow for speedup through automatic vectorization. Namely, strip-mining, loop fission, inlining, transforming conditional branches and loops, SIMD directives are the techniques we employed. We also designed higher level approaches to reduce costs and shorten execution time. These include cache-friendly algorithms, high-order interpolations, transforming data layouts to use an efficient multiple right-hand side vectorized solver. Applying all these transformations, we achieved a speedup of 7× on the advection operators on all three architectures: KNL, Broadwell, Skylake. Furthermore, a speedup of 2× to 3× were observed on the restitution times.

Execution times have been reduced in the production code by 15% in March 2017 and 15% in September of 2017 on old architectures such as Sandy Bridge. Better improvements (up to two times 30%) have been observed on KNL, Skylake and Broadwell architectures. These gains are well illustrated by the two last columns of Table 13. Thanks to these code transformations and those of the previous subsections, users have saved a large fraction of their time allocations both in 2017 and 2018. This means **41** millions of core-hours not spent in 2017 and **90** millions of core-hours not spent in 2018.

6. MDFT

The molecular density functional theory and its associated code MDFT are a disruptive way of tackling the problem of the embedding medium at the molecular scale. It computes *fast* the solvation structure (where are solvant molecules) and solvation free energy (how much does it cost to embed) of any object in water.

The code is originally serial.

Code team:

- Yacine Ould Rouis & Matthieu Haefele (MdlS) for WP1
- Cedric Gageat & M. Levesque (MdlS) for WP3

Performance metrics

The representative benchmark: "benchmark_mid":

Domain size	128*128*128*84
Solute size	1960 sites
Resources	1 node
IO details	All the output are written at the end (1 GB)
Comments	typical targeted production run, initially takes few tens of minutes, and uses 10 GB memory. It was chosen as a reference case for the performance evaluation

Table 17: Performance metrics for MDFT on the JURECA HPC system - Compiler: gfortran - case: benchmark_mid.

	Metric name	Initial (01'2017)	After app support (03'2017)			
				threads		
		w/o OpenMP	w/o OpenMP	4	8	24
Global	Total Time (s)	1529	843	287	190	120
	Time IO (s)	2.42	1.61	1.63	1.97	1.91
	Time MPI (s)	-	-			
	Memory vs Compute Bound	-	-			
	Load Imbalance (%)	-	-	15.07	17.70	44.21
IO	IO Volume (MB)	1094.79	1070.23			
	Calls (nb)	278102	271851			
	Throughput (MB/s)	452.24	665.92	655.83	543.99	560.67
	Individual IO Access (kB)	4.03	4.03			
Node	Time OpenMP (s)	-	-	262.51	157.03	80.99
	Ratio OpenMP (%)	-	-	89.7	81.6	67.4
	Synchro / Wait OpenMP (s)	-	-	10.33	12.77	18.45
	Ratio Synchro / Wait OpenMP (%)	-	-	3.97	8.31	23.15
Mem	Memory Footprint (GB)	N.A.	10.28	10.25	10.25	11.01
	Cache Usage Intensity	0.50	0.50	0.45	0.48	0.59
Core	IPC	2.15	2.08	1.99	1.72	1.35
	Runtime without vectorisation (s)	1530	844	288	194	121
	Vectorisation efficiency	1.00	1.00			

Application support

The application support on MDFT focuses on the introduction of OpenMP multithreading, and a general improvement of the code's quality and its core-level performance. The work has been conducted based on the conclusions of the performance evaluation. Its success is due to the strong implication and the close collaboration of the MDFT team.

In the following, I describe point-by-point the different actions and steps taken in this work, and expose the results at the end.

Activity type	WP1 support
Contributors	Y. Ould-Rouis (WP1), M. Levesque (WP3), C. Gageat (WP3)

Main obstacles

- Defining a relevant benchmark: It is a major step in every project, to design a test case close to the production use in respect to memory and bandwidth usage, the type of calculations, the IO volumes and frequency, and the calculations reproducibility. In this case, we first had to determine a big case that does not exceed the available memory. We also had to replace the L-BFGS minimizer with a fix number of simulations, in order to reproduce the same number of iterations independently from rounding errors (*next point) introduced by different compiling options, or changes in the code. We later redefined the input, adding a non trivial molecule in order to reveal the real behavior of the lennard Jones forces calculation.
- Rounding precision: If the simple precision was validated as suitable for small grids, the tests we conducted on benchmark_mid, and especially when changing compiler, or changing the order of a reduction operation (sum), revealed very big disparities. It was then decided to return to double precision for such big cases.
- The introduction of openmp caused some bugs (catastrophic errors) related to the use of some "block" statements (introduced in Fortran 2008).

Work description

1. **energy_cproj_mrso** handles a four dimentional array, *delta_rho(angles, x, y, z)*, through five main parts. Each step computes calculations, accessing this array in a different order. Each of these parts was distributed separately:
   The first and fifth parts call, for each space cartesian coordinate (x, y, z), the subroutines *angl2proj* and *proj2angl* on the (already contiguous) angles array. These functions had to be adapted to an omp distribution on the space coordinates (z), by making them reentrant (thread safe): the global variables used in these functions in order to save allocation time in, up to now, exclusively serial executions, had to be replaced with inner variables, respecting the stack memory size (allocation on the heap when needed), and calls to *fftw_execute* routines had to be replaced by thread safe forms of these routines[2]. The cost of these necessary modifications was +4.2% on angl2proj time and +15% on proj2angl time, mainly due to the allocs and frees. The memory overhead was negligible.

[2] http://www.fftw.org/fftw3_doc/Thread-safety.html

The second and fourth parts of *energy_cproj_mrso* compute, for each angle, three dimensional dfts using fftw. This operation requires, for each angle, a copy of the values of *delta_rho* for all space coordinates into a contiguous 3 dimensional array, and a copy back to the initial array after the dft calculation. In our case, this array represents 128^3 double precision values, and has to be stored on the heap. The *fftw_execute* routines were already in their thread safe form.
Finally, the third and costliest part of *energy_cproj_mrso* was distributed without notable difficulties, as the calculations made on every orientation and cartesian coordinate were totally independant.
This OpenMP distribution results in a speedup of *energy_cproj_mrso* of roughly 6 times on 8 threads, and 11 times on 24 threads.

2. The first profiling results with a big Lysozyme solute molecule (1960 sites) showed an unexpected amount of time spent calculating the Lennard Jones forces (in **calcul_lennardjones**). In the light of these measurements, the code holders identified a big optimization potential consisting in the implementation of the water model case separately from the general case. This particular case, that represents the big majority of the target use, only takes into consideration the interaction between the oxygen atoms. The result is more than 70% improvement on this routine, which makes the total run 36% faster.
This step also solved an overhead problem caused by the log function calls when introducing OpenMP on JURECA: *ieee754_log_avx* alone slowed down calcul lennardjones from 750 to 1200 seconds in the passage from no openmp to 1 openmp thread. This overhead has totally disappeared. An eye has to be kept on this issue for the general solute (non water) case.

3. About the introduction of OpenMP in **calcul_lennardjones**, the first choice was to distribute the loop on the number of sites. This allowed a good load balance, but caused calculation errors in a case where the number of sites was inferior to the number of threads. This issue still needs to be investigated, and meanwhile, it was replaced by a less advantageous distribution on an inner loop (on the z coordinates). The difference, due to the inbalance and a repetitive threads initialization (for every molecule site) is estimated to 5% on the whole run time (this translates, on 24 threads, in x4 more time spent in the OpenMP barriers of this loop). After this step, *calcul_lennardjones* has a speedup of 4.2 on 8 threads, and 8.5 on 24 threads compared to a serial run.

4. A serial optimization has been identified in the calculation of rotation matrices between spherical harmonics. This routine is called in the third part of *energy_cproj_mrso*, for every angular and cartesian coordinate. It contains a part of invariant small arrays initialization and calculation that become expensive with the redundance after hundreds of millions of calls. The factorization of these small calculations at the beginning of the simulation allowed a gain of 55% on **rotation_matrix_between_spherical_harmonics_lu**, resulting in 9% improvement of *energy_cproj_mrso*

5. After the distribution of the largest parts of the code, the Amdahl limit could be lowered by the distribution of smaller hotspots, like **Energy_ideal_and_external** and **chargedensityandmolecularpolarizationofasolventmoleculeatorigin**, identified using Scalasca and Vampir trace visualization.

6. As the code is still in development phase, the code holders decided not to resort to "agressive" optimizations that would make some parts of the code harder to read or modify, for little improvement. Therefore, few parts with improvement potential were left serial, for the moment.

7. The profiles and traces at this stage also uncovered useless post-processing operations. Quoting Maximilien:
"At first I did not understand why you were calling *histogram_3d* so many times. I would have expected less than 10 calls. In fact, for a solute containing more than few sites, it makes no sense to call *output_rdf* and thus *histogram_3d*."
These operations, that represent 70 seconds, about 7% of the serial time at this stage, could be totally removed for solutes above a defined amount of sites.

Table 18: MDFT Improvements and scalability for each modified routine from VTune profiling, benchmark_mid, JURECA, gfortran. inclusive time expressed.

subroutine	Original	Improved (March 2017)			
	serial (s)	serial (s)	serial improvement	parallel improvement: Speedup	
				8 threads	24 threads
calcul_lennardjones	756	204	-73%	4.5	8.5
energy_cproj_mrso	586	535	-8.7%	5.5	11
rotation_matrix_..._lu	100	45	-55%	6.5	17
angl2proj	95	99	+4.2%	5.5	8
histogram_3d	71	0	-100%		
proj2angl	63	73	+15%	6	11.5
energy_ideal_and_external	36	36		10	27.5
chargedensityandmolecular...	46	47		4	5.5
libc_malloc,int_free	15	21	+40%	2.7	3
All OpenMP regions		805		5	10
Total run	1530	850	-44%	4.5	7.5

Results

- The optimisation work results in 44% improvement in the serial code execution time, bringing it from 1530 to 850 seconds.

- The introduction of OpenMP multithreading allows a better exploitation of the calculation resources, with x3 speedup on 4 cores. The parallel efficiency drops to x4.5 speedup on 8 cores, and roughly x8 on 24 cores. The Table 18 details the improvements and scalability for each hotspot.

- With a load inbalance rising to 44% on 24 threads (Table 17), and 40 seconds still serial (30% of the execution time on 24 threads), there is still room for improvement. Few regions in the code were identified as good candidates for OpenMP, but the quantity of complexity introduced in the code compared to the potential gains was not accepted in a code that is still in development and maturation phase.

- More efforts can also be put in the quality of the distribution in the different regions of the code. We talked in this chapter (Work description, point 3) about the limitations met in *calcul_lennardjones*, where a closer look at the bug could unlock a better distribution, therefore a better scalability.

- In *energy_cproj_mrso*, we can see, Figure 17, the load unbalance due to the division

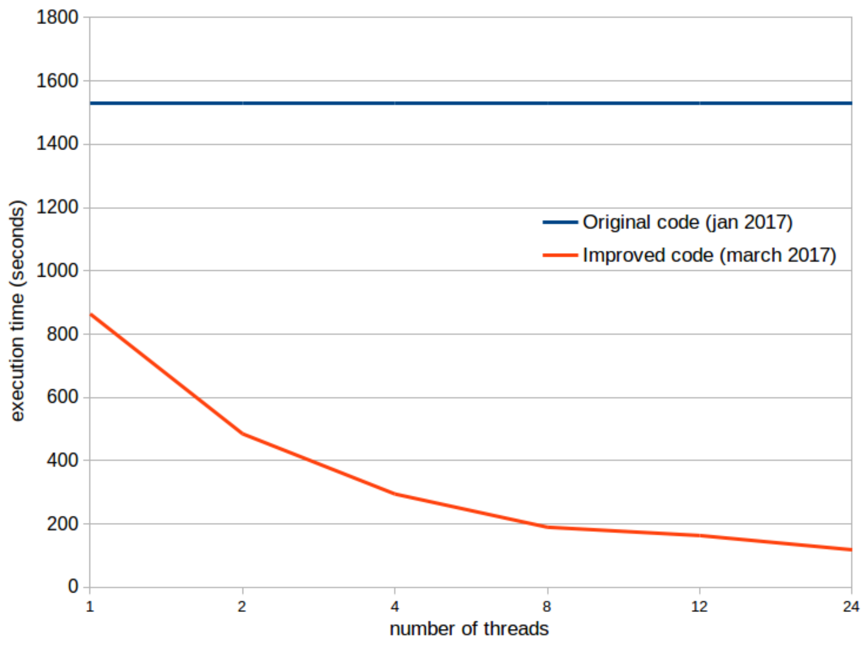
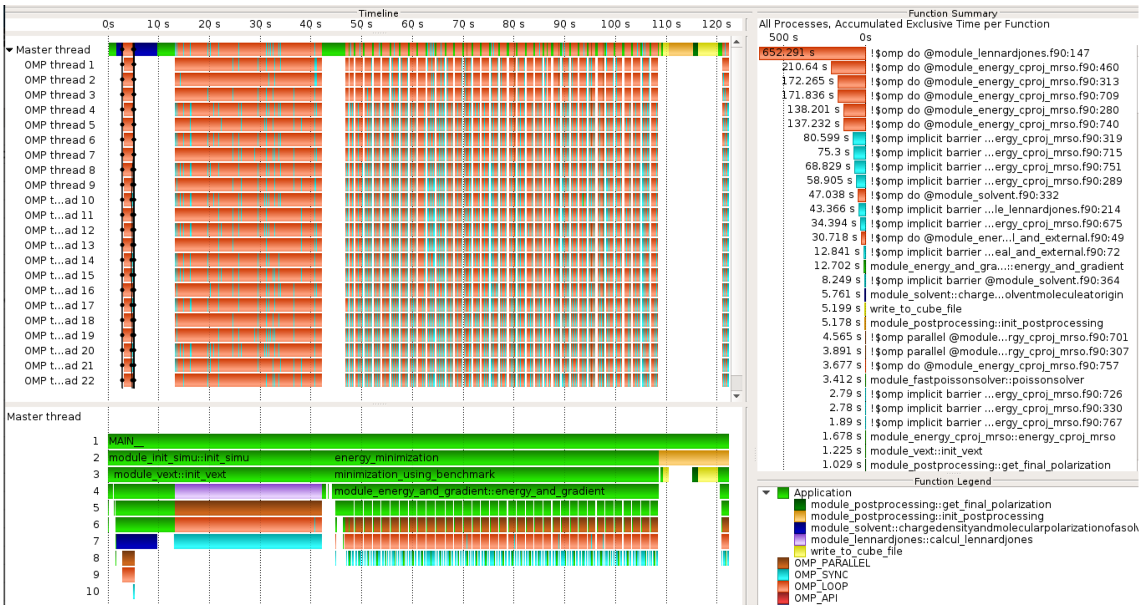


Figure 15: MDFT scalability on one JURECA node, from 1 to 24 threads, before and after application support - case: benchmark_mid.

remainder of the iterations number (here 128) on the number of cores (here 24). This can be reduced by introducing different levels of distribution on the different imbricated loops. The serial time in between successive calls totalizes 10 seconds, and can be distributed.

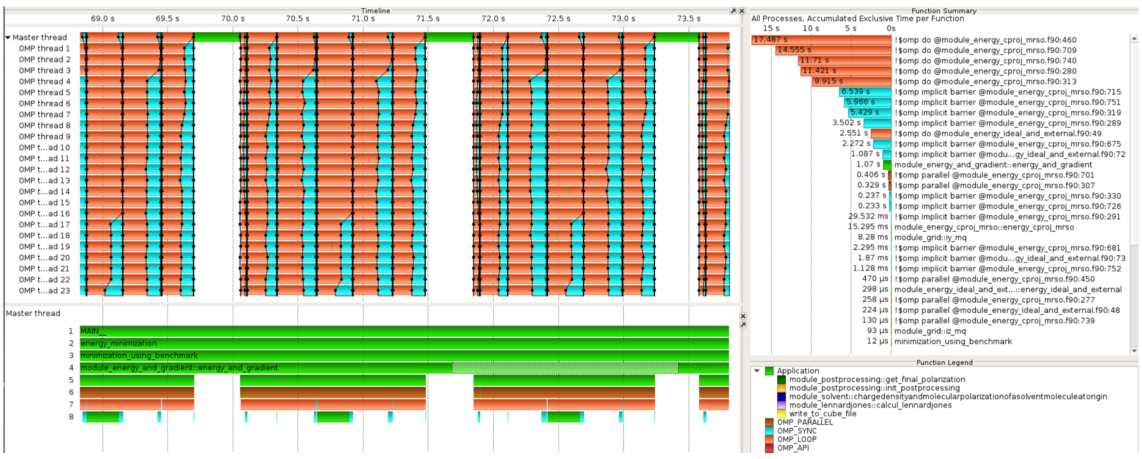
- We see no significant rise in memory usage (less than 10%). However, the multithreading and necessary adaptations introduced 40% increase in the allocation and deallocation time. It could be the object of advanced optimization, using preallocations for every thread.
- Following the serial optimization, the IPC rate dropped from 2.15 to 2.08 ipc. There was no improvement in terms of SIMD.
- The cache usage intensity, or in other words, the L3 cache hit rate, is still at 5O%. It rises to 59% when using 2 sockets. The memory vs compute bound test gives a very small advantage to the scatter mode showing a memory-bound behavior.



```
All Processes, Accumulated Exclusive Time per Function
652.291 s  !$omp do @module_lennardjones.f90:147
 210.64 s  !$omp do @module_energy_cproj_mrso.f90:460
172.265 s  !$omp do @module_energy_cproj_mrso.f90:313
171.836 s  !$omp do @module_energy_cproj_mrso.f90:709
138.201 s  !$omp do @module_energy_cproj_mrso.f90:280
137.232 s  !$omp do @module_energy_cproj_mrso.f90:740
 80.599 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:319
   75.3 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:715
 68.829 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:751
 58.905 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:289
 47.038 s  !$omp do @module_solvent.f90:332
 43.366 s  !$omp implicit barrier ...le_lennardjones.f90:214
 34.394 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:675
 30.718 s  !$omp do @module_ener...l_and_external.f90:49
 12.841 s  !$omp implicit barrier ...eal_and_external.f90:72
 12.702 s  module_energy_and_gra...::energy_and_gradient
  8.249 s  !$omp implicit barrier @module_solvent.f90:364
  5.761 s  module_solvent::charge...olventmoleculeatorigin
  5.199 s  write_to_cube_file
  5.178 s  module_postprocessing::init_postprocessing
  4.565 s  !$omp parallel @module...rgy_cproj_mrso.f90:701
  3.891 s  !$omp parallel @module...rgy_cproj_mrso.f90:307
  3.677 s  !$omp do @module_energy_cproj_mrso.f90:757
  3.412 s  module_fastpoissonsolver::poissonsolver
   2.79 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:726
   2.78 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:330
   1.89 s  !$omp implicit barrier ...ergy_cproj_mrso.f90:767
  1.678 s  module_energy_cproj_mrso::energy_cproj_mrso
  1.225 s  module_vext::init_vext
  1.029 s  module_postprocessing::get_final_polarization

Function Legend
Application
    module_postprocessing::get_final_polarization
    module_postprocessing::init_postprocessing
    module_solvent::chargedensityandmolecularpolarizationofasolv
    module_lennardjones::calcul_lennardjones
    write_to_cube_file
OMP_PARALLEL
OMP_SYNC
OMP_LOOP
OMP_API
```

Figure 16: Vampir trace of MDFT after optimization (March), on 24 threads, JURECA. Case: benchmark_mid, compiler: gfortran.



```
All Processes, Accumulated Exclusive Time per Function
  17.487 s  !$omp do @module_energy_cproj_mrso.f90:460
  14.555 s  !$omp do @module_energy_cproj_mrso.f90:709
   11.71 s  !$omp do @module_energy_cproj_mrso.f90:740
  11.421 s  !$omp do @module_energy_cproj_mrso.f90:280
   9.915 s  !$omp do @module_energy_cproj_mrso.f90:313
   6.539 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:715
   5.966 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:751
   5.429 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:319
   3.502 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:289
   2.551 s  !$omp do @module_energy_ideal_and_external.f90:49
   2.272 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:675
   1.087 s  !$omp implicit barrier @modu...gy_ideal_and_external.f90:72
    1.07 s  module_energy_and_gradient::energy_and_gradient
   0.406 s  !$omp parallel @module_energy_cproj_mrso.f90:701
   0.329 s  !$omp parallel @module_energy_cproj_mrso.f90:307
   0.237 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:330
   0.233 s  !$omp implicit barrier @module_energy_cproj_mrso.f90:726
29.532 ms  !$omp implicit barrier @module_energy_cproj_mrso.f90:291
15.295 ms  module_energy_cproj_mrso::energy_cproj_mrso
  8.28 ms  module_grid::iy_mq
 2.295 ms  !$omp implicit barrier @module_energy_cproj_mrso.f90:681
  1.87 ms  !$omp implicit barrier @modu...gy_ideal_and_external.f90:73
 1.128 ms  !$omp implicit barrier @module_energy_cproj_mrso.f90:752
   470 µs  !$omp parallel @module_energy_cproj_mrso.f90:450
   298 µs  module_energy_ideal_and_ext...::energy_ideal_and_external
   258 µs  !$omp parallel @module_energy_cproj_mrso.f90:277
   224 µs  !$omp parallel @module_energy_ideal_and_external.f90:48
   130 µs  !$omp parallel @module_energy_cproj_mrso.f90:739
    93 µs  module_grid::iz_mq
    12 µs  minimization_using_benchmark

Function Legend
Application
    module_postprocessing::get_final_polarization
    module_postprocessing::init_postprocessing
    module_solvent::chargedensityandmolecularpolarizationofasolventmoleculeatorigin
    module_lennardjones::calcul_lennardjones
    write_to_cube_file
OMP_PARALLEL
OMP_SYNC
OMP_LOOP
OMP_API
```

Figure 17: Zoom on successive calls to *energy_cproj_mrso* (and its 5 OpenMP regions), on jureca node, 24 threads. On the right, some measurements on the sample shown.

7. Metalwalls

Activity type	Consultancy and Support
Contributors	Haefele M., Marin Lafleche A

MetalWalls is a molecular dynamics simulation code dedicated to the study of electrochemical systems. Its main originality comes from the capability to simulate electrodes held at constant potential. Currently, it is used to model nanoporous electrodes, known as super-capacitors, to better understand their behavior and improve their potential applications in the field of energy storage, energy production and desalination.

From a computer science point of view, the application follows a two steps algorithm: i) to compute the charges density within the electrodes according to the atom locations, an iterative process that requires to reach a convergence, ii) to compute the position of each atom according to the charge density. More than 90% of the total runtime is spent in the charge density computation, so this is definitely a target for the code optimization process.

In term of parallelisation, the application is pure MPI and no data is distributed across the ranks. This means that each rank has all the whole data. Only the computations are distributed such that each rank computes the interaction between a set of pairs of atoms. An MPI Allreduce sums up all these contributions and sends the result back to all MPI ranks.

Code team:

- Matthieu Haefele (MdlS) and Abel Marin-Lafleche (MdlS) for WP1
- Mathieu Salanne (MdlS) for WP3

Metrics

Case1 characteristics:

Domain size	3776 ions (walls + melt)
Resources	1 node on JURECA (24 cores)
IO details	Checkpoint written every 10 steps instead of 1000 ⇒ much larger than production
Type of run	both a development and small production run

Table 19: Performance metrics for Metalwalls on the JURECA HPC system.

	Metric name	03/01/2016	27/01/2017	15/12/2017	05/09/2018
Global	Total Time (s)	43.2	10.0	8.0	2.8
	Time IO (s)	0.3	N.A	N.A	N.A
	Time MPI (s)	12.4	7.2	0.8	2.4
	Memory vs Compute Bound	1.1	0.9	1.5	0.7
IO	IO Volume (MB)	35.8	N.A	N.A	N.A
	Calls (nb)	384000	N.A.	N.A.	N.A
	Throughput (MB/s)	105.0	N.A.	N.A.	N.A
	Individual IO Access (kB)	0.1	N.A.	N.A.	N.A
MPI	P2P Calls (nb)	0	0	0	0
	P2P Calls (s)	0.0	0.0	0.0	0.0
	Collective Calls (nb)	2721	1408	707	721
	Collective Calls (s)	0.1	7.0	0.7	1.9
	Synchro / Wait MPI (s)	11.7	6.9	0.6	1.8
	Ratio Synchro / Wait MPI	94.8	93.8	38.7	68.5
	Message Size (kB)	908.4	925.7	462.9	3612.3
	Load Imbalance MPI	24.8	17.8	20.8	10.6
Node	Ratio OpenMP	0.0	0.0	89.0	0.0
	Load Imbalance OpenMP	0.0	0.0	0.1	0.0
	Ratio Synchro / Wait OpenMP	0.0	0.0	2.1	0.0
Mem	Memory Footprint (MB)	66	134.3	165.6	19.8
	Cache Usage Intensity	N.A.	0.9	0.86	0.97
	RAM Avg Throughput (GB/s)	N.A.	N.A.	N.A.	N.A
Core	IPC	N.A.	0.60	0.57	1.34
	Runtime without vectorisation (s)	46.5	55.0	40	19.6
	Vectorisation efficiency	1.1	5.5	5.0	7.0
	Runtime without FMA (s)	44.6	11.0	9	3.0
	FMA efficiency	1.0	1.1	1.12	1.1

```
do i = ibeg_w, iend_w
   vsumzk0=0.0d0
   do j = nummove+1,num
      if (i==j) then
         vsumzk0=vsumzk0+q(j)*sqrpieta
      else
         zij=z(i)-z(j)
         zijsq=zij*zij
         rerf = erf(eta*zij)
         vsumzk0=vsumzk0+q(j)*((sqrpieta*exp(-etasq*zijsq))+&
            (pi*zij*rerf))
      end if
   enddo
   cgpot(i)=cgpot(i)-vsumzk0
enddo
```

Figure 18: Kernel not vectorized by the compiler

Memory footprint reduction

At several places in the code, the information if the interaction between two specific atoms has to be taken into account is needed. As this information depends only on the type of system simulated, in the original version, it was computed once during the code initialisation and stored. But the amount of memory required to store this information grows with N^2, N being the number of atoms in the simulation.

The optimisation that has been implemented suppresses completely the need for this N^2 memory by recomputing this information each time it is required from existing information of size N. Now the memory footprint of the application scales linearly with the number of atoms and enables to treat larger systems. From the restitution time point of view, this optimisation had only a moderate impact as the time spent in this part of the code was not that important. Unfortunately, we could not measure the impact of this single optimisation as it has been done in conjunction with the code vectorisation.

Vectorisation

As mentioned in the performance report, the vectorization of the code could be the source of potential improvements. A careful examination of the compiler log could identify the internal loops that the compiler could not vectorise.

For instance, Fig 18 shows a kernel not vectorised by the compiler. The *if* statement introduces an issue: the iteration $j = i$ executes different code than $j = i - 1$ and $j = i + 1$. The compiler can simply not transform this code into a Single Instruction Multiple Data (SIMD) version. As a consequence, the whole j loop is not vectorised. By examining the code in the *if* and *else* branches, one can notice that the purpose of this construct is to save the evaluations of an error function, an exponential function and some multiplications. This optimisation has been likely implemented at a time where scalar processors did not have vector units. Nowadays, this construct prevents the compiler from introducing vector instructions. By removing the *if* part and keeping only the *else* part, a speedup of 2.5 could be obtained on this single kernel.

Other code modifications enabled the compiler vectorization and now, thanks to Intel Advisor, we could check that all the kernels in the high computing intensity part of Metalwalls are vectorised by the compiler.

Cache blocking

During the porting of Metalwalls on Intel Xeon Phi KNL architecture, we observed larger run times than expected for some routines and especially the *cgwallrecipE* routine. After an examination with the memory analyser of VTune, it turned out that these routines were almost compute bound on Xeon architectures and became memory bound on KNL. A careful examination of the source code revealed that several large arrays were accessed within the same kernel. These large arrays were still fitting in the L3 cache of Xeon processors but, as there is no L3 cache on KNL, these arrays could not fit into L2 cache. As a consequence, the kernel triggered a large amount of memory transfer and, despite the considerably large memory bandwidth of KNL's MCDRAM, the execution time of this kernel on KNL was larger by a factor of 8.

A cache blocking mechanism has been implemented on this kernel. Instead of performing computations on the total size of the arrays, computations are performed only on a subset such that the sum of all these subsets fit into the cache. The implementation was not completely trivial as a reduction of some of these arrays was performed inside the kernel and used directly in the kernel. The kernel had to be split in two and an intermediate data structure that accumulates the partial reductions had to be introduced. The overhead in memory of this data structure is negligible and we could recover very good performance on this specific routine.

Hybrid MPI/OpenMP parallelisation

An hybrid MPI/OpenMP version of Metalwalls has been implemented by adding OpenMP directives to the existing code base. Figures 19 and 20 shows the basic additions to the code base in order to add OpenMP support.

The performance of the hybrid version was evaluated on two different machines: "JURECA", based on Intel "Haswell" 12-core E5-2680 V3 processors and "Frioul" available at CINES[3], based on Intel "Knight Landing" 68-cores 7250@1.40GHz processors. On each machine, the test case "Blue Energy" was run for 10 time steps on 2 nodes. Table 20 reports the time to solution for this test case on these two architectures. For the MPI only version, 1 MPI rank was assigned to each physical core on the nodes: 24 ranks per node on JURECA and 68 ranks per node on Frioul. For the hybrid version, the reported time correspond to the best configuration of rank per node and threads per rank assignment. On JURECA, this corresponds 8 tasks per node and 3 threads per tasks, on Frioul this corresponds to 17 tasks per node and 8 threads per tasks, where the SMT 2 mode is activated. The hybrid MPI/OpenMP version improve the time to solution by 6% on a Haswell architecture and by 20% on a KNL architecture compared to the MPI only version. On top of that, as can be seen on Figure 24, the hybrid version enables a better scalability since less MPI rank per nodes are used.

[3] `www.cines.fr`

```
!$omp parallel
!$omp do reduction(+:Potential) schedule(dynamic) &
!$omp private(j, potj, qj, i, rij, rijNormSquare, rijNorm, ercFactor, potj)
do j = 2, numParticles
    potj = 0.0d0
    qj = q(j)
    do i = 1, j-1
        rij(:) = r(:,j) - r(:,i)
        rijNormSquare = dot_product(rij(:),rij(:))
        if(rijNormSquare < rijNormSquareMax) then
            rijNorm = sqrt(rijNormSquare)
            erfcFactor = (erfc(alpha*rijNorm) - erfc(eta*rijNorm)) / rijNorm
            potj = potj + q(i) * erfcFactor
            Potential(i) = Potential(i) + q(j) * erfcFactor
        end if
    end do
    Potential(j) = Potential(j) + potj
end do
!$omp end do
!$omp end parallel
```

Figure 19: Kernel RealE with OpenMP directive

```
!$omp parallel
!$omp do reduction(+:SkCos, SkSin) &
!$omp private(particleBlock, iStart, iEnd, kMode, kNormSquare, i, rdotk)
do particleBlock = 1, numParticleBlocks
    iStart = (particleBlock -1)*blockSize + 1
    iEnd = min(particleBlock*blockSize, numParticles)
    do kMode = 1, numKModes
        kNormSquare = dot_product(k(:,kMode), k(:,kMode))
        if (kNormSquare < kNormSquareMax) then
            do i = iStart, iEnd
                rdotk = dot_product(r(:,i), k(:,kMode))
                SkCos(kMode) = SkCos(kMode) + cos(rdotk)
                SkSin(kMode) = SkSin(kMode) + sin(rdotk)
            end do
        end if
    end do
end do
!$end omp do

!$omp do private(particleBlock, iStart, iEnd, kMode, kNormSquare, i, rdotk)
do particleBlock = 1, numParticleBlocks
    iStart = (particleBlock -1)*blockSize + 1
    iEnd = min(particleBlock*blockSize, numParticles)
    do kMode = 1, numKModes
        kNormSquare = dot_product(k(:,kMode), k(:,kMode))
        if (kNormSquare < kNormSquareMax) then
            do i = iStart, iEnd
                rdotk = dot_product(r(:,i), k(:,kMode))
                Potential(i) = Potential(i) + &
                        Sk*(cos(rdotk)*SkCos(kMode) + sin(rdotk)*SkSin(kMode))
            end do
        end if
    end do
end do
!$omp end do
!$omp end parallel
```

Figure 20: Kernel RecipE with OpenMP directive

Table 20: Time to solution (in seconds) on different architectures

Machine	MPI	MPI/OpenMP
JURECA (BDW)	82.2	77.3
Frioul (KNL)	61.4	48.8

GPU offloading with OpenACC

The GPU porting of Metalwalls started in Q1 2017 with the objective of comparing the OpenPower platform with the already available Intel KNL and Broadwell platforms both from performance and energy points of view. The GPU implementation of MetalWalls was evaluated on Ouessant, an IBM OpenPower Minsky S822LC prototype available at IDRIS[4] which is based on IBM Power8+ processors and NVidia P100 GPUs (Pascal). A single node contains 2 Power8+ processors and 4 GPUs all connected with NVLink.

This porting effort took around three months of development, a relatively short amount of time given the lack of prior knowledge on GPU programming from the developers of Metalwalls. Three main factors can explain this fact. First, we decided to use OpenACC, a directive-based programming model. It is the least intrusive programming model in the source code among the available GPU programming technologies. CUDA implementation was not considered because the complexity of the resulting code would have been too high to be integrated successfully in the production version and further maintained in time. OpenMP was not considered neither because its level of maturity was clearly way below the OpenACC one at that time. Second, the most compute intensive parts of the code were clearly identified and with the help of OpenACC tutorials available online, the first trivial kernels could be successfully offloaded on the GPU. A PATC workshop on ”Performance portability for GPU application using high-level programming approaches” gave us the opportunity to meet GPU and OpenACC experts and to solve issues we encountered with more complex kernels. Finally, we relied on the ”Unified Memory” feature available on NVidia platform since CUDA 6. It relieves the programmer from a lot of the memory transfer and management between the host and the accelerator, which in turn can focus on loop optimization.

Figures 21 and 22 show the listing of two typical compute intensive kernels with OpenACC directive included. As can be seen in these figures, the impact on the code base is minimal and, except 150 lines, a single code base for the whole code could be kept for all three Broadwell, KNL and OpenPower platforms.

FPGA porting activity

Some efforts have been made to port Metalwalls, or at least a part of it, on FPGA, more specifically on the Data Flow Engine (DFE) technology developed by Maxeler. Matthieu Haefele was invited by Maxeler to spend two weeks in their office in London during summer 2017 to get started on this new technology.

The development of a DFE implementation requires a change of paradigm in code design compared to CPU or even GPU programming. Indeed, on a CPU or a GPU, data move-

[4]`http://www.idris.fr/`

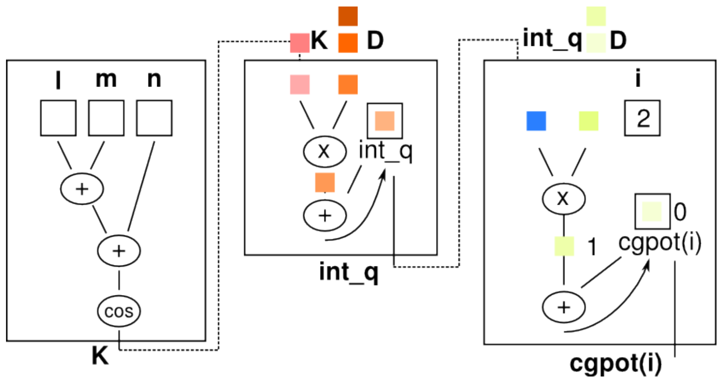
```
!$acc parallel
!$acc loop private(potj)
do j = 2, numParticles
   potj = 0.0d0
   qj = q(j)
   !$acc loop private(rijNormSquare, rijNorm, erfcFactor) reduction(+:potj)
   do i = 1, j-1
      rij(:) = r(:,j) - r(:,i)
      rijNormSquare = dot_product(rij(:),rij(:))
      if(rijNormSquare < rijNormSquareMax) then
         rijNorm = sqrt(rijNormSquare)
         erfcFactor = (erfc(alpha*rijNorm) - erfc(eta*rijNorm)) / rijNorm
         potj = potj + q(i) * erfcFactor
         !$acc atomic update
         Potential(i) = Potential(i) + q(j) * erfcFactor
      end if
   end do
   !$acc atomic update
   Potential(j) = Potential(j) + potj
end do
!$acc end parallel
```

Figure 21: Kernel RealE with OpenACC directive

```
!$acc parallel
!$acc loop gang
do kMode = 1, numKModes
   SkCosKMode = 0.0d0
   SkSinKMode = 0.0d0
   kNormSquare = dot_product(k(:,kMode), k(:,kMode))
   if (kNormSquare < kNormSquareMax) then
      !$acc loop vector reduction(+:SkCosKMode, SkSinKMode)
      do i = 1, numParticles
         rdotk = dot_product(r(:,i), k(:,kMode))
         SkCosKMode = SkCosKMode + cos(rdotk)
         SkSinKMode = SkSinKMode + sin(rdotk)
      end do
      SkCos(kMode) = SkCosKMode
      SkSin(kMode) = SkSinKMode
   end if
end do

!$acc loop gang
do kMode = 1, numKModes
   kNormSquare = dot_product(k(:,kMode), k(:,kMode))
   if (kNormSquare < kNormSquareMax) then
      !$acc loop vector
      do i = 1, numParticles
         rdotk = dot_product(r(:,i), k(:,kMode))
         !$acc atomic update
         Potential(i) = Potential(i) + &
               Sk*(cos(rdotk) * SkCos(kMode) + sin(rdotk) * SkSin(kMode))
      end do
   end if
end do
!$acc end parallel
```

Figure 22: Kernel RecipE with OpenACC directive



(a) DFE implementation

```
cgpot = 0.0
do l = 0, nb_mode_x
  do m = -nb_mode_y, nb_mode_y
    do n = -nb_mode_z, nb_mode_z
      int_q = 0.0
      K = cos(l+m+n)
      do i = 1, Nmelt + Nwalls
        int_q += K*D(i)
      end do
      do i = 1, Nwalls
        cgpot(i) += int_q*D(i)
      end do
    end do
  end do
end do
```

(b) Original Fortran kernel

Figure 23: DFE implementation of a Fortran kernel

ments are triggered by the flow of instructions of the application and are then handled by the hardware. The intrinsic data stream character of a FPGA leaves to the developer the care of moving the data explicitly, between the memory and the FPGA chip of course, but also inside the chip. Multiply, add or more complex operations like cosine or other complex functions are then put on the different data paths in order to implement the desired algorithm. Figure 23 shows the DFE implementation of kernel and the original fortran code. Three DFE kernels are necessary to increase computations concurrency due to reduction operations. Lines inside and between kernels are data paths inside the FPGA chip on which the different operations are put. The colored squares represent data flowing inside the chip along these paths. The array D is streamed in the chip from memory by two different kernels and the cgpot array, the result of the algorithm, is streamed out back into memory.

Knowing the amount of data that should go through each DFE kernel, the number of required clock ticks to perform the full kernel execution is known. As a consequence, the performance of the implementation can be predicted with a very good confidence (5-10%) with a simple spreadsheet. So the design phase of the DFE implementation is strongly driven by this spreadsheet analysis. Once the design finished, the implementation can start using MaxJ, the embedded DSL based on java developed by Maxeler. It allows to describe how data flows within each kernel and how kernels are connected together. An eclipse plug-in speeds up the development and provides a comfortable unit testing environment for each kernel.

As a compilation that would allow to run the application on the real hardware takes between 24 and 48 hours, all executions needed to develop the algorithm are performed within a FPGA simulator/emulator in order to check the algorithm correctness. A basic emulator is used to run the unit tests but cannot execute algorithms that require more than one kernel. The full emulator has then to be used in this case with the offload mechanism. Small fortran proxy applications have been developed to read input files, initialise data structures and call the right routine in order to offload the desired workload onto the FPGA. The emulator is then triggered, it executes the kernels and sends back the result to the fortran application. The full algorithm correctness can be then tested this way provided the test case is small enough to fit and to be run in a reasonable time within the emulator.

Porting Metalwalls to FPGA consists then in implementing DFE kernels to run the most computing intensive part of the code on the device. In term of code base, this represents 30% of the 20k lines of the full code. A subset of these 30% is currently targeted for this activity, namely the conjugate gradient part. If this part can be successfully and efficiently ported to FPGA, there is a good chance that the full 30% can be ported and ensure maximum performance as the communication between the CPU and the FPGA will be minimum. We have currently successfully set up a design that tells us that a single FPGA should be faster than a bi-socket Haswell node by a factor 4-5. The corresponding kernels have been implemented, tested in the emulator and they give the same result as the original application on a very small test case. We are at the point where a run on the real hardware is mandatory. This requires a compilation and then the support from Maxeler to achieve this as it is a complex task and is completely new for our team. The natural computing system to perform the runs is the PRACE PCP machine based on Maxeler technologies hosted at JSC. Accounts on the machine have been created for project members and we are eager to measure run time and energy requirement.

MW2: Refactoring the code to increase performance and usability

A new version of the code has been developed in order to improve its performance and maintanability. Building upon the acquired knowledge from the existing code base, the objectives were to increased code readability, ease further code development and improve time to solution. It order, to keep the software easily readable to the research team, the new version is written, as the original version, in Fortran and the MPI library is used for parallelisation.

One striking feature of the existing code base is that all variables are global and the subroutines do not have any arguments. This makes it hard for the code developer to (1) know the type of the variables without to constantly refer to the global declaration file and (2) to determine which variable are actually modified by a subroutine. Additionaly, all variables are declared within the same module ('commondata'). Therefore, the context in which the variables are used is not appearent in the declaration file. In order to cope with these two features, the new version uses modules and derived data types. Modules are used to create a context, they usually define a derived data type and all the functions and subroutine related to this data type. The derived data type is a structure used to store related variable, for example the MW_ewald_t datatype stores the number of reciprocal space k-points, the cut-off parameter alpha, and so on. Forcing all subroutines declaration to be done in a module, the compiler is able to check the call signature of each subroutine, preventing the developer to misuse a subroutine.

Furthermore, an effort was made to use self-descriptive variable name and to avoid name confusion as much as possible. In the original version, there was two variables named 'eta' and 'eta2' which were referring to the gaussian parameter in the Ewald summation and in the electrode charge model respectively. However, it was hard to know it without referring to the input reader routine. In the new version, the names were replaced by 'eta' and 'alpha' respectively corresponding to the notation used in the documentation. Additionally, 'eta' and 'alpha' are members of two distinct derived types, one corresponding the charge distribution definition and one corresponding to the Ewald parameters definition. We hope therefore to reduce the confusion on this particular point. Another example is the naming of the Ewald summation work array 'ckcsakk' and 'ckssakk' renamed into 'Sk_cos' and 'Sk_sin' respectively and members of the Ewald derived data type.

Most subroutines were broken down into smaller and more independent units of computation. By doing this, we hope that adding a new feature will have a smaller impact on the code base than before. As an example, in the original version, the routine named 'rattle' is responsible to perform the whole time step integration. It implements the rattle algorithm, which is used to enforce bond constraints in molecules, but also computes the effect of the thermostat, calls a subroutine to update the electrode charges, calls another subroutine to compute the forces on melt species and updates species coordinates and velocities. This routine was split into several small routines. The rattle algorithm is now decomposed into two routines, one called before the force update and the other after. The thermostat is applied using also two subroutines. The time step integration is now performed in the main loop and is simpler to read.

A workshop was help on July 6 in the PHENIX laboratory to present the new code design and performance and kickstart the development of new features by the lab's researcher. The workshop led to fruitful discussion and positive feedbacks.

Much effort was devoted on the original version to improve the performance by reducing the memory footprint, enabling compiler automatic vectorization and introducing cache blocking techniques to reduce memory access. All these optimization were kept in the new version.

Additionally, a careful study of the system of equation solved by the code let appear that a significant amount of work could be avoided. Indeed, during the charge computation process, the electrostatic potential felt by all electrode atoms due to other electrode atoms must be computed at each iteration of the conjugate gradient minimization step. In the original version, the code computed the potential due to all species and subtracted the potential due to melt species only. A procedure was derived to compute directly the desired quantity and thus avoid to compute the contribution of melt species. This can yield a significant reduction in the amount of work since in some system ,such as 'blue energy', the ratio of melt species to electrode species is greater than 4.

The performance of the code was improved by a factor 2.4 when run on 2 nodes of Jureca (Intel Haswell processor with 24 core per nodes) - 77.3 s for MW1 and 31.63 s for MW2, system: Blue Energy 10 steps. The same system runs 5.4 times faster on Irene (Intel Skylake processors with 48 cores per nodes) - 5.82 s for MW2, system: Blue Energy 10 steps. The new code base is already used in production runs on the Irene supercomputing system (Intel Skylake nodes).

The new code base is already used by researchers to introduce new features (single electrode model, thermodynamic integration) into the code in order to study different physical parameters or different models.

Future work on a very short term will consist in doing again the exercise of porting the code on GPU with MPI+OpenACC and developing the hybrid MPI+OpenMP version to leverage the SMT functionalities of current CPUs.

Results

Metwalls' performance has been improved by a factor 4.3 between the initial version and the version provided in January 2017. This improvement is mainly related to vectorisation, memory footprint reduction and cache blocking techniques. An additional 25% could be

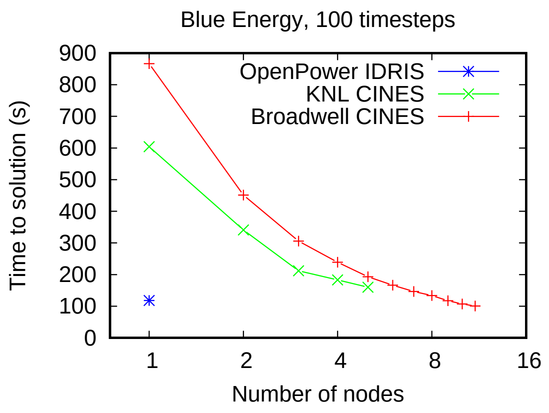


Figure 24: Time to solution for 100 iterations of the BlueEnergy test-case on different architectures

obtained by the end of 2017 thanks to the hybrid MPI/OpenMP implementation. It enables the efficient usage of the SMT features on Intel Xeon and Xeon Phi and to reduce the pressure on the interconnect with a smaller number of MPI ranks per node. The new version of the code developed in 2018 improves the performance by yet an additional factor of 2.8 and is used in production since July 2018. All these support actions have lead to a total speedup of 15.4 and have been merged in the production version of the code regularly since the beginning of the project. 28 MCPUh were used by Metalwalls on various Tier-0 and French Tier-1 machines during 2016, 2 MCPUh in 2017 and 1.4 MCPUh in 2018 (of which 0.7 MCPUh uses the new version of the code). Only for 2018, this represents a saving of 13.2 MCPUh. Overall, on the three years of the project this represents a saving of 60.1 MCPUh.

Thanks to the hybrid MPI+OpenMP and MPI+OpenACC implementations of Metalwalls, it is also possible to compare computing architectures. First, the comparison on a small scale shows a clear advantage of the GPU architecture. Offloading the code to a single NVidia P100 GPU is 2.1 (resp. 1.6) faster than running the code on a single node of Occigen with Intel "Broadwell" 2x14-core E5-2690 V4@2.6GHz (resp. a single KNL node 7250 68-cores @1.40GHz). At scalability limit, i.e. maximum scalability while keeping a parallel efficiency above 70%, the best time to solution is obtained on the Broadwell platform with the usage of 11 nodes. When 100 simulation time steps are executed in this configuration, only 62 are executed on 5 KNL nodes and 84 on a single OpenPower node during the same wall clock time. Scalability curve is presented on figure 24. However, the best energy to solution is obtained on the OpenPower platform. At the same scalability limit, the OpenPower platform requires 2.5 less energy than the KNL platform to perform the same simulation. Unfortunately, energy measurement issues were encountered on the Broadwell platform, so the energy to solution criterion is not available there.

Overall, Metalwalls has been improved by a factor 15.4 which could save 60.1 MCPUh and allow now researchers to perform the same numerical experiments with 15.4 less computing resources. Hybrid MPI+OpenMP and MPI+OpenACC are now available and researchers can target now multi-core and GPU architectures to perform their production runs. All these support activities have been conducted with the aim of keeping a single and as maintainable as possible code base. The objective seems to be achieved as the new code base is already used by researchers to introduce new features (single electrode model, thermodynamic integration) into the code in order to study different physical parameters or different models. Finally, some first attempts to port Metalwalls on FPGA have been tried. Preliminary results are promising but this is still work in progress.

8. PVnegf

Overview

Central to the prediction of material properties for solar cells is the utilization of an accurate and versatile simulation software intended to treat all of the relevant processes on equal footing and enabling an efficient exploration of the parameter space. Simulations based on NEGF provide unique physical insight, but they are also computationally demanding especially when the target are simulations of real-world heterojunctions. On the other hand, the exploration of parameter-space would require high-throughput accurate simulations. Consequently, one of the keys to successfully predict material properties is a highly efficient and optimized numerical implementation. A blueprint of such an implementation is provided by the 1D-NEGF code explained below. The initial focus during a EoCoE workshop was on the current implementation and its evaluation of OpenMP.

Code team:

- Edoardo Di Napoli (FZJ) for WP2
- Urs Aeberhard (FZJ) for WP3
- Thomas Breuer (FZJ) for WP1

Case1 characteristics:

single bias point of 40-nm GaAs p-i-n photodiode under monochromatic illumination:

Domain size	Nz=100 (spatial grid), Nk=32 (momentum grid), NE=406 (energy grid)
Resources	24 OpenMP Threads on 1 node on JURECA (24 cores)
IO details	only sequential IO (input file, physical output quantities) → no bottleneck
Type of run	reduced production run, only two SCBA self-consistency iterations, but including all the elements of full run (electron-photon interaction, electron-phonon coupling - POP+AC, evaluation of LDOS, carrier density and current, scattering rates, absorption coefficient

Application support

Current implementation:

Activity type	Assessment of OpenMP implementation and performance optimization
Contributors	Thomas Breuer, Brian Wylie, Sebastian Lührs, Urs Aeberhard

Starting from a serial version of PVnegf a parallel implementation with OpenMP has been developed. The JUBE metrics shown in Table 21 show a speed-up close to 4.

External work inspired by EoCoE:

Activity type	Hybrid MPI-OpenMP parallelisation of core modules in PVnegf
Contributors	Sebastian Achilles, Edoardo Di Napoli, Urs Aeberhard

	Metric name	w/o OpenMP	w/ OpenMP
Global	Total Time (s)	467	118
	Time IO (s)	5.1	5.19
	Time MPI (s)	N.A.	N.A.
	Memory vs Compute Bound	0.99	1.00
	Load Imbalance (%)	0.00	68.02
IO	IO Volume (MB)	169.11	169.11
	Calls (nb)	1682219	1682219
	Throughput (MB/s)	33.14	32.56
	Individual IO Access (kB)	0.10	0.10
Node	Time OpenMP (s)	N.A.	46.5
	Ratio OpenMP (%)	N.A.	38.84
	Synchro / Wait OpenMP (s)	N.A.	19.77
	Ratio Synchro / Wait OpenMP (%)	N.A.	45.59
Mem	Memory Footprint	3162676kB	2993368kB
	Cache Usage Intensity	0.75	0.75
Core	IPC	1.66	2.16
	Runtime without vectorisation (s)	456	105
	Vectorisation efficiency	0.98	0.89
	Runtime without FMA (s)	471	120
	FMA efficiency	1.01	1.02

Table 21: Performance metrics for PVnegf using Case1 on the JURECA HPC system on 24 cores (MPI metrics omitted since PVnegf is only OpenMP parallel).

Some core modules of PVnegf have been designed and implemented from scratch with MPI and OpenMP parallelisation. This code is successfully tested on the full JUQUEEN system and acts as a blueprint implementation.

1D-NEGF

The code 1D-NEGF is the result of a master thesis focused on proof-of-concept optimizations. It was developed from scratch to get familiar with the algorithm and implement an efficient code based on the underlying framework. The PVnegf code, recently developed at the IEK-5, can simulate a variety of physical properties, due to the possibility to configure the couplings between the dynamical player of the non-ballistic model. However PVnegf offers only shared memory parallelisation, and is thus limited to a single node. In the master thesis we have developed a new simplified version of the PVnegf code, termed 1D-NEGF, with the aim of obtaining an efficient and scalable implementation for the NEGF framework. Despite the fact that the 1D-NEGF currently contains only a subset of features of the original PVnegf code, the long-term goal is to blueprint the design principles of 1D-NEGF to improve the PVnegf code.

The 1D-NEGF implementation does only contain unipolar single-band simulations containing only interaction between electrons and phonons. Therefore only simulations with reduced physics can be performed. However the insights of the parallelisation can be used also to implement and parallelise other interactions and features. The underlying data structure and data distribution stays the same, so that the parallelisation strategy of the proof-of-concept code could be blueprinted and adopted to the PVnegf code. The distributed memory parallelisation allows not only to compute results much faster by using more resources, but also to compute bigger, and so more realistic, system sizes as well as increased accuracy through the possibility to study finer grid sizes. The complexity of simulations of real-world nanostructures leads to an exascale problem.

Code Performance:

The Code is portable between different architectures, since it only depends on a efficient `BLAS` and `LAPACK` library, which is available on every modern supercomputer installation. The performance and availability has been tested on quite different architectures: Intel Xeon Haswell on JURECA, Intel Xeon Phi (KNL) on JURECA booster and Blue Gene/Q on JUQUEEN. In the following these results are presented. The 1D-NEGF code is based on the use of the hybrid MPI + OpenMP parallelisation.

JURECA:

In Figure 25 we show the speedup results of the 1D-NEGF code with only one outer and one inner loop (= 1 step). This test was performed on JURECA with a system size of $N_K = 32$, $N_E = 4096$, and $N_P = 100$. Up to 1024 nodes have been used. In this scaling test, we have used 1 MPI task per node and 48 threads per task.

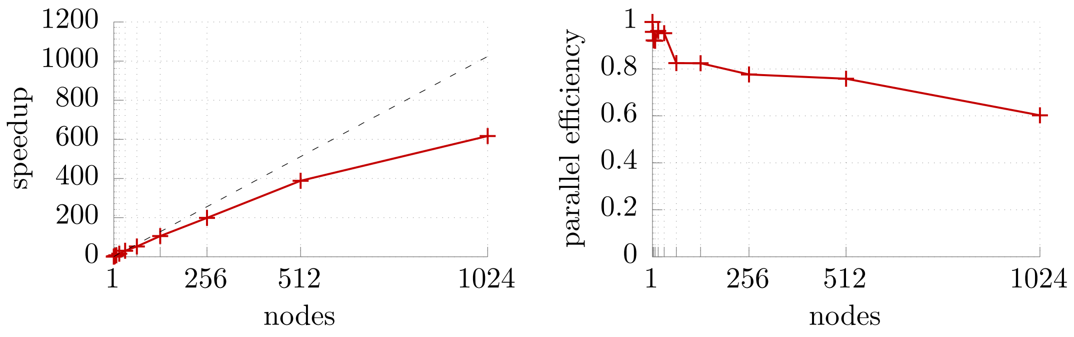


Figure 25: Scaling behaviour (left) and efficiency (right) of one outer and one inner SCF loop of 1D-NEGF on JURECA at JSC. This data was obtained with a problem size of $N_K = 32$, $N_E = 4096$, and $N_P = 100$.

After further optimization new scaling results have been measured with a system size of $N_K = 64$, $N_E = 1792$, see Figure 26. In this scaling test, we have used 1 MPI task per node and 24 threads per task. The results show that the implementation scales up to 1024 nodes reaching a parallel efficiency of 70.8% compared to 1 node.

JURECA booster:

The program also scales on the booster. This test was performed with a system size of $N_K = 32$, $N_E = 480$, and $N_P = 100$. Up to 48 nodes have been used. In this scaling test, we have used 1 MPI task per node and 64 threads per task. The current job profile contains independent jobs. Different operation points are submitted with a bash script as individual jobs.

After some optimization the scaling experiement on the booster was repeated, see Figure 28. This test was performed with a system size of $N_K = 64$, $N_E = 1792$, and

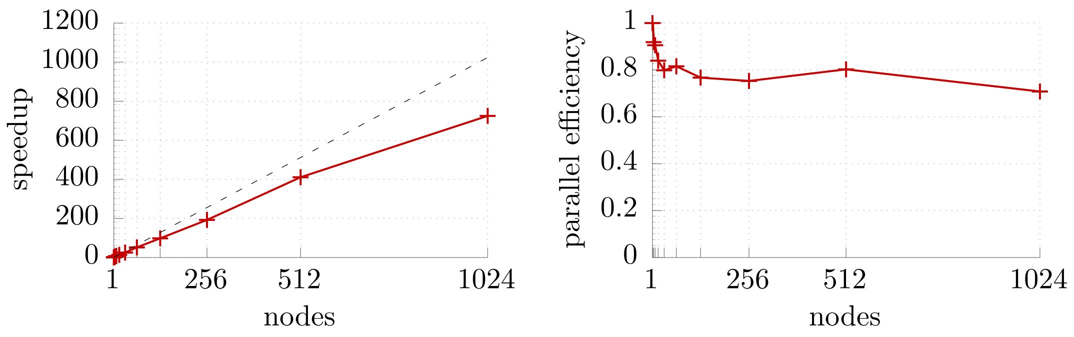


Figure 26: Scaling behaviour (left) and efficiency (right) of one outer and one inner SCF loop of 1D-NEGF on JURECA at JSC. This data was obtained with a problem size of $N_K = 32$, $N_E = 4096$, and $N_P = 100$.

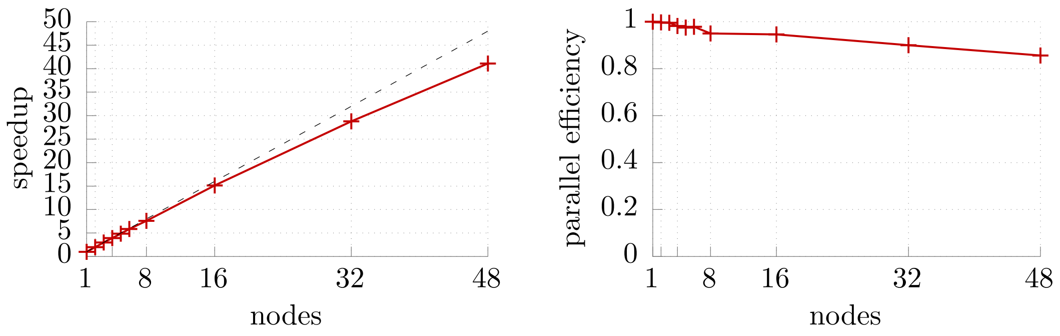


Figure 27: Scaling behaviour (left) and efficiency (right) of one outer and one inner SCF loop of 1D-NEGF on the booster at JSC. This data was obtained with a problem size of $N_K = 32$, $N_E = 480$, and $N_P = 100$.

$N_P = 100$. Up to 1024 nodes have been used. Due to memory limitations, the scaling test could only be started with at least 4 nodes. In this scaling test, we have used 1 MPI task per node and 64 threads per task.

JUQUEEN:

For large scaling results, we show in Figure 29 the speedup results of one inner loop. This test was performed on JUQUEEN with a system size of $N_K = 2048$, $N_E = 5376$, and $N_P = 100$. Up to 28 672 nodes with 16 cores and 2-way SMT have been used.

An entire simulation was performed with a system size of $N_K = 64$, $N_E = 1792$, and $N_P = 100$. The results are reported in Figure 30.

JUBE metrics:

For the JUBE metrics a small input file with $N_K = 32$, $N_E = 128$ and $N_P = 100$ was used. On JURECA 16 nodes with 1 MPI rank and 24 OpenMP threads each have been used to extract the metrics shown in Table 22. Note that the metrics for the efficiency of vectorisation and FMA appear to show no impact on 1D-NEGF. This is due to the way those metrics are obtained: different compiler switches are used to re-compile the code and asses the impact on vectorisation and FMA. Since 1D-NEGF relies on external (system) libraries for vectorisation and FMA the metrics do not show any substantial change without the libraries being re-compiled.

Table 22: Performance metrics for 1D-NEGF as measured on JURECA with 16 nodes running 1 MPI ranks and 24 OpenMP threads each. Note that the 'Core' section is misleading, see text for details.

	Metric name	1D-NEGF
Global	Total Time (s)	43
	Time IO (s)	0.00
	Time MPI (s)	0.29
	Memory vs Compute Bound	0.98
	Load Imbalance (%)	65.59
IO	IO Volume (MB)	0.02
	Calls (nb)	0
	Throughput (MB/s)	7.19
	Individual IO Access (kB)	0.00
MPI	P2P Calls (nb)	1200
	P2P Calls (s)	0.21
	P2P Calls Message Size (kB)	26
	Collective Calls (nb)	4
	Collective Calls (s)	0.02
	Coll. Calls Message Size (kB)	2
	Synchro / Wait MPI (s)	0.13
	Ratio Synchro / Wait MPI (%)	1.92
Node	Time OpenMP (s)	10.93
	Ratio OpenMP (%)	25.00
	Synchro / Wait OpenMP (s)	1.36
	Ratio Synchro / Wait OpenMP (%)	13.71
Mem	Memory Footprint	1459488kB
	Cache Usage Intensity	0.98
Core	IPC	0.64
	Runtime without vectorisation (s)	42
	Vectorisation speedup factor	0.98
	Runtime without FMA (s)	44
	FMA speedup factor	1.02

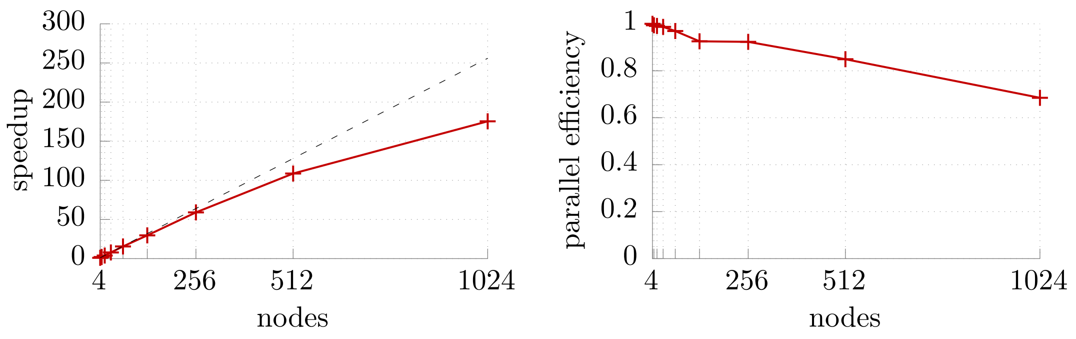


Figure 28: Scaling behaviour (left) and efficiency (right) of one outer and one inner SCF loop of 1D-NEGF on the booster at JSC. This data was obtained with a problem size of $N_K = 64$, $N_E = 1792$, and $N_P = 100$.

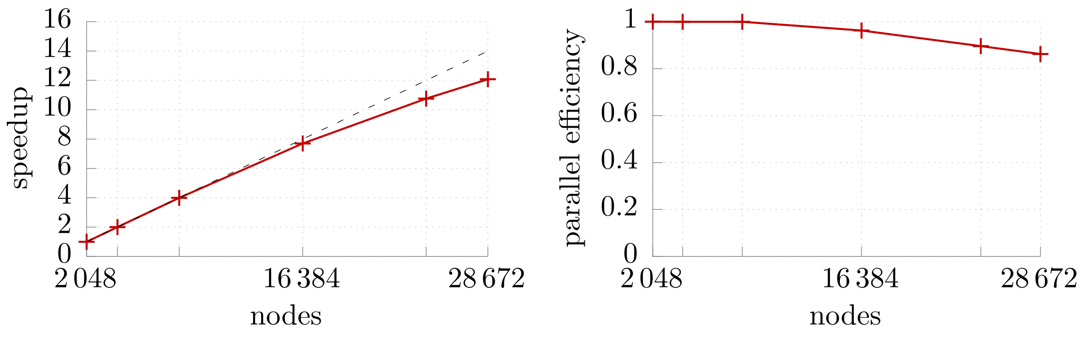


Figure 29: Scaling behaviour (left) and efficiency (right) of the inner SCF loop of 1D-NEGF on JUQUEEN at JSC. This data was obtained with a problem size of $N_K = 2048$, $N_E = 5376$, and $N_P = 100$.

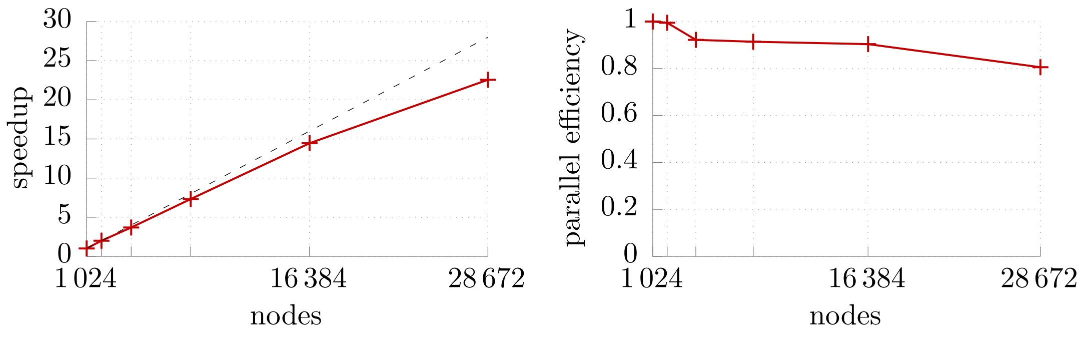


Figure 30: Scaling behaviour (left) and efficiency (right) of one entire simulation with multiple outer and inner SCF loops of 1D-NEGF on JUQUEEN at JSC. This data was obtained with a problem size of $N_K = 64$, $N_E = 1792$, and $N_P = 100$.

9. Parflow

Overview

ParFlow is a parallel physics-based integrated watershed model, which simulates fully coupled, dynamic 2D/3D hydrological, groundwater and land-surface processes suitable for large scale problems. ParFlow is used extensively in research on the water cycle in idealized and real data setups as part of process studies, forecasts, data assimilation experiments, hind-casts as well as regional climate change studies from the plot-scale to the continent, ranging from days to years. Saturated and variably saturated subsurface flow in heterogeneous porous media are simulated in three spatial dimensions using a Newton-Krylov nonlinear solver [3, 8, 17] and multigrid preconditioners, where the three-dimensional Richards equation is discretised based on cell-centered finite differences. ParFlow also features coupled surface-subsurface flow which allows for hillslope runoff and channel routing [12]. Because it is fully coupled to the Common Land Model (CLM), a land surface model, ParFlow can incorporate exchange processes at the land surface including the effects of vegetation [18, 11]. Other features include a parallel data assimilation scheme using the Parallel Data Assimilation Framework (PDAF) from [20], with an ensemble Kalman filter, allowing observations to be ingested into the model to improve forecasts [14]. An octree space partitioning algorithm is used to depict complex structures in three-dimensional space, such as topography, different hydrologic facies, and watershed boundaries. ParFlow parallel I/O is via task-local and shared files in a binary format for each time step. ParFlow is also part of fully coupled model systems such as the Terrestrial Systems Modeling Platform (TerrSysMP) [24] or PF.WRF [19], which can reproduce the water cycle from deep aquifers into the atmosphere. ParFlow is written in C and CLM is written in Fortran 90 (117000 lines of C code and 20000 lines of Fortran code) and parallelised using MPI. The solvers currently used in ParFlow are Hypre (preconditioner) and KINSOL non-linear solver (SUNDIALS).

Code team:

- Stefan Kollet (FZJ) (WP4)
- Ketan Kulkarni (FZJ) (WP4 associated)
- Slavko Brdar (FZJ) (WP4 associated)
- Klaus Görgen (FZJ) (WP4)
- Wendy Sharples (FZJ) (WP1)

Benchmark characteristics:

A three-dimensional sinusoidal topography as shown in Figure 31 was used as the computational domain with a lateral spatial discretization of $\Delta x = \Delta y = 1$m and a vertical grid spacing of $\Delta z = 0.5$m; the grid size, n, was set to $nx = ny = 50$ and $nz = 40$ resulting in 100 000 unknowns per CPU core, with one MPI task per core. In order to simulate surface runoff from the high to the low topographic regions with subsequent water pooling and infiltration, a constant precipitation flux of 10 mm/hour was applied. This results in realistic non-linear physical processes and thus compute times. The water table was implemented as a constant head boundary condition at the bottom of the domain with an unsaturated zone above, 10m below the land surface. The heterogeneous subsurface was simulated as a spatially uncorrelated, log-transformed Gaussian random field of the saturated hydraulic conductivity with a variance ranging over one order of magnitude. The soil porosity and permeability were set to 0.25 m/day. This idealized setup was used for the profiling case study as opposed to a real world set up due to the symmetry inherent in the setup. In contrast, a real world experiment has asymmetry in both the meteorological forcing and also the model topography which naturally lead to load imbalances. These asymmetries could therefore obscure whether there are actually load imbalances due to poor software design.

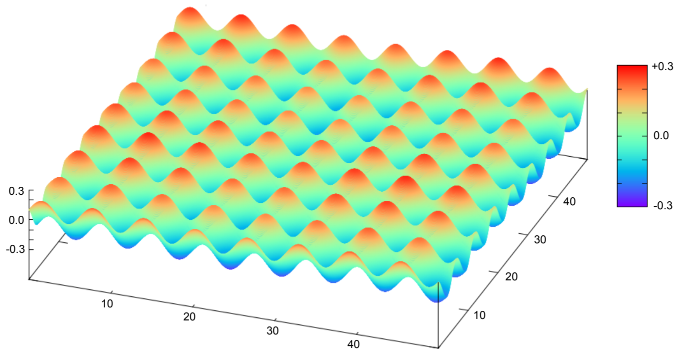


Figure 31: Model setup, showing cross-sectional domain and sinusoidal topography variation from the top of the model (z=20) for each processor.

Test case 1, ParFlow v320: This test uses the model described above with the current stable release. Weak scaling tests with 24 MPI ranks and 48 MPI ranks were run and the performance metrics were gathered.

Test case 2, ParFlow + p4est v320: This test uses the model described above with the current stable release integrated with p4est. p4est is integrated with ParFlow such that p4est becomes the parallel mesh manager. Weak scaling tests with 24 MPI ranks and 48 MPI ranks were run and the performance metrics were gathered.

EoCoE benchmark tables

Tables 23 and 24 show the summary of the benchmarks that have been run during the project so far. The two tables each present one of the test cases on 24 and 28 processors. The individual columns represent results from the latest stable version of ParFlow, v320 and that latest stable version integrated with p4est, using p4est as the parallel mesh manager (see Section 9).

Findings from the comparison: ParFlow integrated with p4est spends less time in communication than the stable version of ParFlow amounting to a speed up of 30%. However caution should be taken in interpreting these results as most of this reduction could be in the setup phase. Test cases with more time steps and more in depth profiling should be run in the future to confirm which routines account for the biggest speedups.

Both versions of ParFlow are compute bound, in agreement with the performance report from PoP.

If the vectorization is turned off, then the code performance is worsened, indicating that there is a potential for serial performance improvement via vectorization. Without FMA operations the code seems to be slightly faster only in the ParFlow case and the difference could just be due to measurement fluctuations.

IO does not appear to be a bottleneck, however a more realistic test case should be used in the future to confirm this.

General comments on overall performance

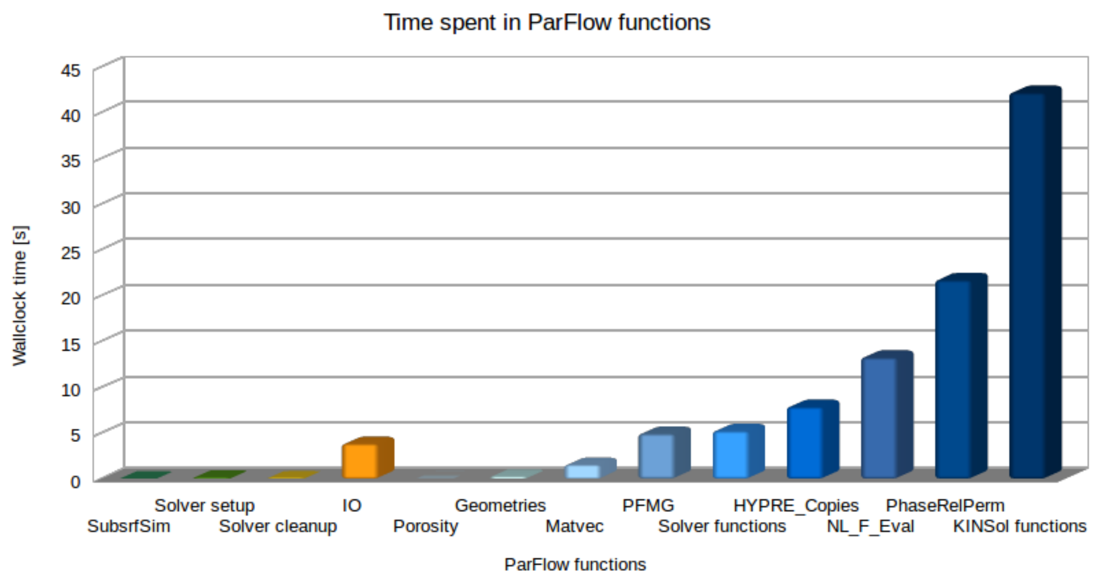


Figure 32: Time spent in ParFlow functions or routines, where the functions/routines can be divided into four categories, set up, clean up, I/O, and solve. The functions in the category "set up" are depicted in green: SubsrfSim—setting up the domain, Solver setup—initializing the solver. The functions in the category "clean up" are depicted in yellow: Solver cleanup—finalizing the solver. The functions in the category "I/O" are depicted in orange: PFB I/O—ParFlow binary I/O. The functions in the category "solve" are depicted in blue: Porosity—calculation of the porosity matrix, Geometries—calculation of the simulation domain, MatVec—matrix and vector operations, PFMG—Geometric Multigrid Preconditioner from HYPRE, Solver functions—miscellaneous functions, HYPRE_Copies—copying data within HYPRE, NL_F_Eval—setting up the physics and field variables for the next iteration, PhaseRelPerm—setting up the permeability matrix, KINSol functions—non-linear solver functions from SUNDIALS

We can see that in running test case 1 ParFlow spends large part of the execution time in the non linear solve step (see Figure 32. The performance metrics gathered show that ParFlow is compute bound and up to a certain amount of processors, ParFlow scales extremely well [13]. However from extensive profiling, it was determined that both time spent in communication and memory use become a problem at scale (see POP_AR_17.pdf). To mitigate this problem, ParFlow was integrated with the p4est, where p4est is now the parallel mesh manager. The performance metrics were gathered comparing these two versions of ParFlow to see what potential gains could be made in future simulations.

Overall, the gains made in using ParFlow + p4est are in a reduction of communication (time spent in MPI) and memory footprint. On average the time spent in communication from the performance metric tables is roughly 30%, and the reduction in time spent in communication with the ParFlow + p4est version means that potentially there could be an overall reduction of run time by up to 10%. However this could be due to the fact that setup time is a large percentage of these runs. Also in running a real case, the load imbalances are large due to inactive regions in the model and heterogeneity so this might counteract any gains. A more realistic test case needs to be run for the next support activity. The memory footprint has been reduced by at least 20% on average and this value does not increase at scale which means that ParFlow + p4est can now scale to the whole JURECA machine.

For both versions of ParFlow, the code is highly compute bound. This means that performance of ParFlow could be further improved via use of different solver libraries and configurations.

If the vectorization is turned off, then the code performance decreases, especially in the case of the ParFlow + p4est version, which indicates a potential for improving serial performance with more vectorization. Without FMA operations the code seems to be slightly faster only in the ParFlow case. However the difference is not significant.

Application support

Activity type	Consultancy (WP1 support request on-going)
Contributors	Sharples W. (Stefan Kollet, Ketan Kulkarni, Lukas Poorthuis, Ilya Zhukov, Damian Kaliszan, Michael Knobloch, Slavko Brdar, Thomas Breuer, Pasqua D'Ambra, Phillippe Leleux, Klaus Goergen, Bibi Naz)

In the following section, the application support activities are explained in chronological order.

Development of a run control framework to aid with porting and tuning, profiling and provenance tracking

In order to streamline development and support, ParFlow needed to be extensively profiled first to determine bottlenecks, scalability breakers and inefficiency. To enable efficient profiling and running of ParFlow, a run control framework (RCF) integrated with a workflow engine was developed by WP1 (Wendy Sharples) [23], as a best practice approach to automate profiling, porting, provenance tracking and simulation runs. The RCF encompasses all stages of the modeling chain:

- preprocess input,
- compilation of code (including code instrumentation with performance analysis tools),
- simulation run,
- postprocess and analysis,

to address these issues. Within this RCF, the workflow engine is used to create and manage benchmark or simulation parameter combinations and performs the documentation and data organization for reproducibility. This approach automates the process of porting and tuning, profiling, testing, and running a geoscientific model. profiling, a run control framework was developed. In order to analyze ParFlow's runtime behavior, determine optimal runtime settings, as well as identify performance bottlenecks during model development, we used several complementary performance analysis tools. Setup, compilation wrappers, and analysis profiling steps were built into our RCF with support for the following tools: Score-P v3.1 [10] and Scalasca v2.3.1 [6, 25], where results collected with Score-P and Scalasca can be examined using the interactive analysis report explorer Cube v4.3.5 [22], Allinea Performance Reports v7.0.4 [7], Extrae v3.4.3 [2], Paraver v4.6.3 [15], Intel Advisor 2015 [21], and Darshan v3.0.0 [5]. In addition, automated metric calculation benchmark scripts whose results are shown in this report, were implemented into the RCF (Wendy Sharples with Thomas Breuer). In the future, the RCF will be updated take advantage of the new cycle feature coming up in JUBE to allow for job resubmission.

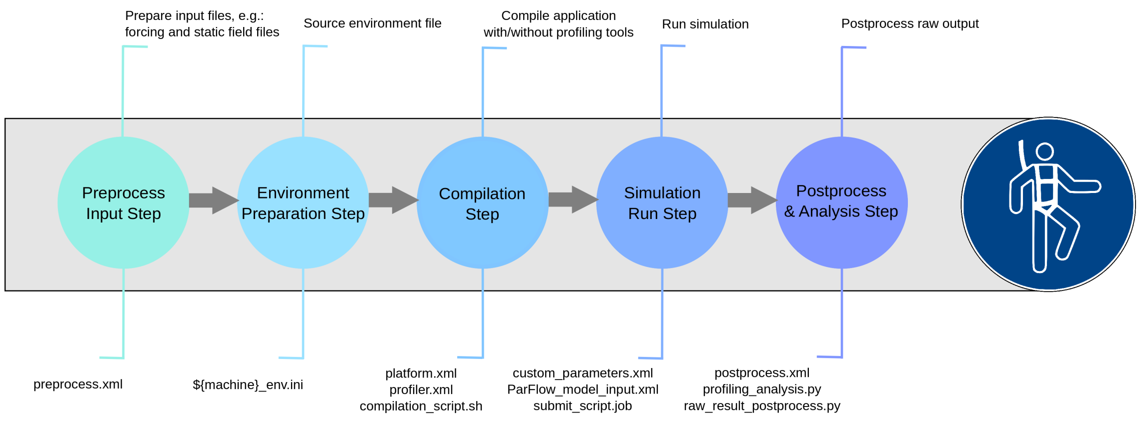


Figure 33: Schematic overview of the modeling chain as supported by our JUBE-based run harness. Each step is annotated with a brief description (top) as well as the respective RCF infrastructure (XML files and scripts, bottom).

Profiling study

Using the RCF, the specialists from the Performance Optimisation and Productivity Centre of Excellence in Computing Applications (PoP - Ilya Zhukov) performed an initial health check of ParFlow (see PoP report, POP_AR_17.pdf). The results from the profiling study show that a significant amount of time is spent in the non-linear solve step, so this was the target area for the profiling study. The profiling study showed that the main barriers to exascale were time spent in communication and memory use which informed the subsequent developments (see Section 9). This is discussed further in Section 9.

Updating ParFlow for next generation hardware

Due to the arrival of the new KNL booster for JURECA and the availability of GPUs at JSC, the application support team took a three pronged approach to getting ParFlow ready for many core, heterogeneous architecture.

(i) Development of a MiniApp:

Since ParFlow will take considerable efforts in refactoring, a MiniApp has been developed to test i) Accelerator enabled solver libraries and ii) different accelerators (KNLs, GPUs against CPUs). The MiniApp a simplified python version of ParFlow which generates a random permeability matrix and an input pressure matrix with a source and a sink and performs one linear solve. The permeability values are based on a log normal distribution where the standard deviation is set from 1 to 3. 1 meaning the least amount of heterogeneity and 3 being the most (see Figure 34, LHS showing log of the permeability). The permeability matrix is then tridiagonalized to reduce the number of arithmetic operations to become the A matrix used in $Ax = b$ solve, where the solution is shown in Figure 34 (see RHS, pressure plots) showing a solution which is not completely smooth due to the heterogeneity in A. This work is undertaken in conjunction with WP1 (Wendy Sharples, Slavko Brdar), PSNC (Damian Kaliszan), and PoP specialists (Michael Knobloch, Ilya Zhukov). Metrics such as scalability, wallclock time and serial performance is considered along with the energy delay product in assessment of each architecture's energy efficiency.

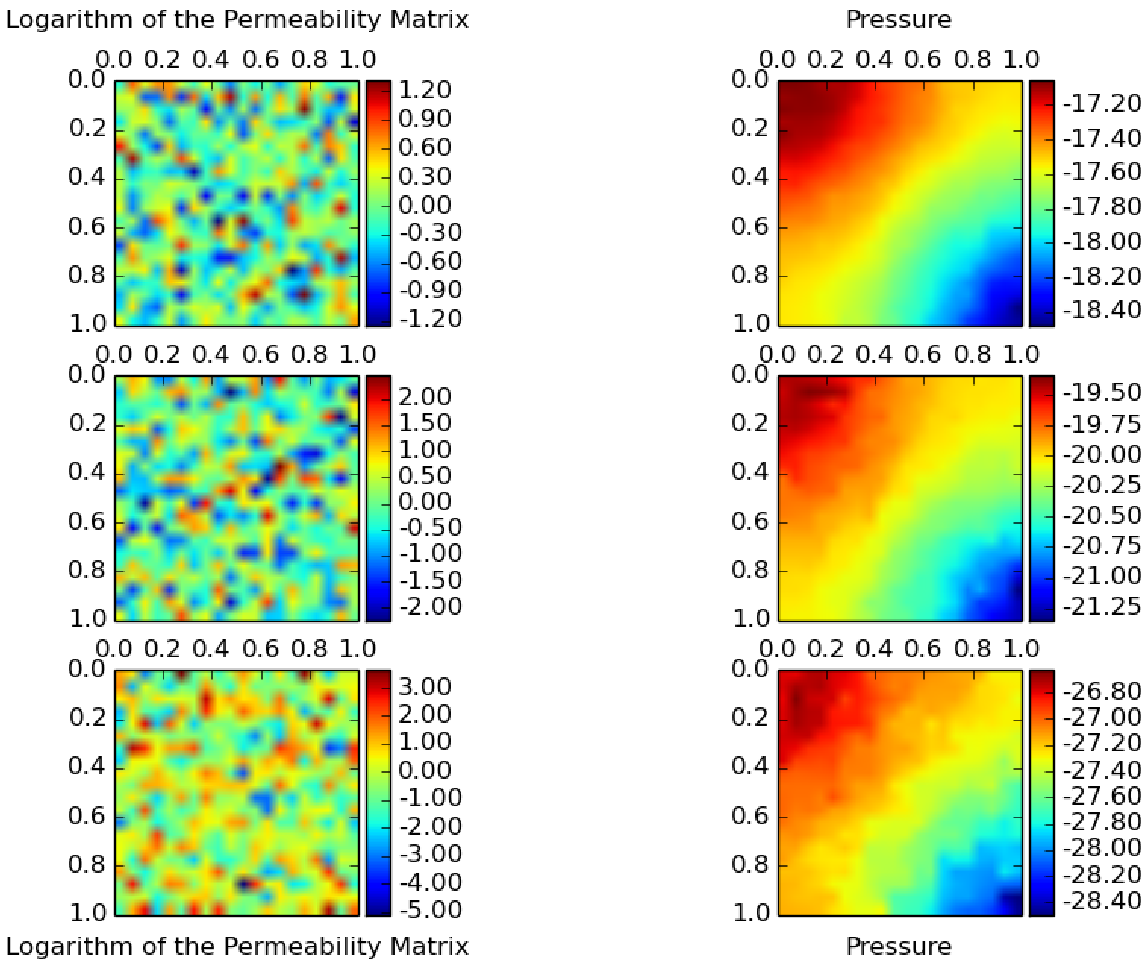


Figure 34: Plot of logarithm of permeability and pressure solution for a 40x40x40 problem, using the PETSc library. The standard deviation varies from 1 to 3 from top to bottom.

(ii) Investigation of state of the art linear solvers:

The A and b matrices set up by the python MiniApp were given to the linear algebra experts in WP1 at the IAC (Pasqua D'Ambra) and University of Brussels (Phillippe Leleux) where they performed a linear solve using different linear solver configurations using in-house and open source solver libraries. The results are promising but in order to be able to use these libraries further support from WP1 will be needed to create an interface in ParFlow in order to plug in and plug out different solvers and preconditioners.

More details on these results can be found in Sections 9 and 9.

(iii) Investigation of the vectorization potential of ParFlow:

ParFlow has many loops that have been implemented as macros. This means that it is difficult to determine dependencies and to determine whether these individual loops can be vectorised. Loops within two frequently used routines have been identified by the ParFlow developers as being able to be vectorized without causing inconsistencies as they have no inherent dependencies under normal running conditions. Work has started with WP1 support (Wendy Sharples, Slavko Brdar) and PoP experts (Ilya Zhukov) to redesign these loops for vectorization. This work is still ongoing. Additionally a preliminary PoP report detailing the road-map to vectorization for ParFlow and initial results from vectorization of some simple loops (which have no dependencies) is underway.

Aiding ParFlow's interoperability

To reduce time spent in preprocessing model input and post-processing binary ParFlow model output, a NetCDF reader and writer is under development, with testing of this new feature integrated into the RCF. To date, NetCDF Meteorological forcing can be read in and all output can be written in NetCDF. Future work will include integrating the NetCDF reader/writer with ParFlow tools.

External support

To reduce the memory usage and to reduce the time spent in communication, an Adaptive Mesh Refinement (AMR) library, p4est, has been implemented into ParFlow to function as the parallel mesh manager. This work was undertaken by Jose Fonseca and Carsten Burstedde at the Institute for Numerical Simulations at the University of Bonn. The approach was minimally invasive and preserves most of ParFlow's data structures, the configuration system, and the setup and solver pipeline. The current mesh manager is a barrier to scalability as it requires that all cells store information about every other cell. This is reduced to neighboring cells under p4est, which results in a decrease in memory use (storage reduction) and a decrease in time spent in communication (communication reduced to neighboring cells only), allowing ParFlow to scale over all 458,752 cores on JUQUEEN [4]. Using p4est as the parallel mesh manager has the additional potential benefit of integrating the adaptive mesh refinement functionality into ParFlow in order to address inactive regions (due to heterogeneous forcing, permeability, etc.) causing load imbalances in the real world models.

MUMPS solver for ParFlow

Activity type	WP1 support
Contributors	I. Duff (CERFACS, WP1), P. Leleux (CERFACS, WP1), D. Ruiz (IRIT, WP1), F. S. Torun (IRIT-CNRS, WP1), W. Sharples (JSC, WP4)

Overview

The target of this support activity is the optimization of the linear solver which is involved in the code ParFlow. This code is designed for the solution of large elliptic and parabolic

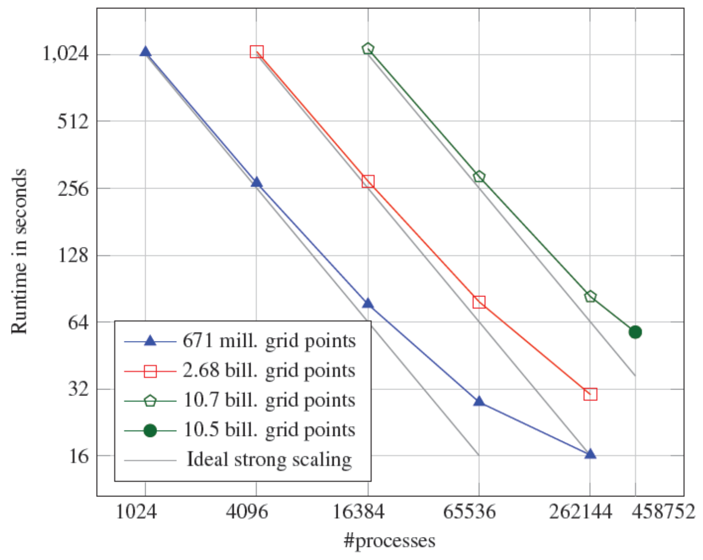


Figure 35: Scaling plot of the latest version ParFlow + p4est implementation demonstrating use of whole JUQUEEN machine. Total runtime is shown for different domain sizes shown [4].

equations on heterogeneous platforms (GPUs, KNLs, and CPUs). The linear systems are symmetric definite positive and arise from 3D finite difference discretization.

The current solution relies on a Krylov solver with a simple preconditioner (ex: Conjugate Gradient preconditioned with Block-Jacobi + Incomplete LU level 0). The largest problem solved is composed of 3.7×10^8 unknowns; the target for EoCoE2 is 3x bigger.

Method and Preliminary Results

We will not give extensive details about the use of MUMPS in this section but will focus on the use of this solver to tackle ParFlow linear systems. For more details on MUMPS principle and particularly on its latest feature: Block Low Rank Approximation (BLR), see MUMPS Section from deliverable 1.7 Software technology improvement.

Our tests were run on CERFACS cluster Nemo and CALMIP cluster EOS. Nemo consists of 1872 nodes with 128GB of memory and 2 sockets of Intel Xeon E5-2680 v3 Haswell CPUs 12-cores clocked at 2.5GHz. EOS consists of 612 nodes with 64GB of memory and 2 sockets of Intel IvyBridge processors 10-cores clocked at 2.8 GHz. In the following "P x N cores" stands for P processes with N threads each. The test case *parflow-def* used for preliminary results was the one in Table 25.

As a first approach, we tried to apply MUMPS with default parameters directly on ParFlow global system. On *parflow-def* test case, we observed that MUMPS is capable of scaling, see Figure 36). However for most applications, and ParFlow is no exception, there is no need to get a solution to machine accuracy. We can thus approximate MUMPS LU factorisation using BLR feature.

On a theoretical point of view, BLR has an asymptotic complexity of $O(n^{4/3})$ with n being the first dimension of the 3D problem. Geometric multigrid will always be a better choice on purely elliptic problems, however on different problems (ex: Helmholtz or structural mechanics) the gains compared to using an iterative scheme could be greater. For more

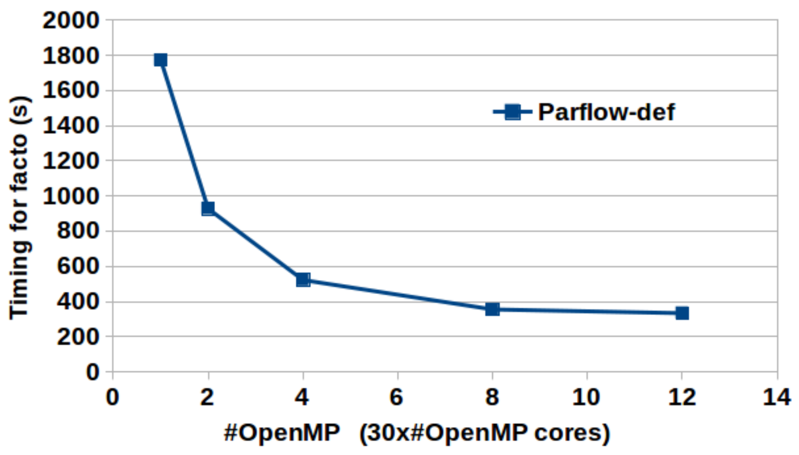


Figure 36: MUMPS factorisation phase depending on #OpenMP for ParFlow matrix *parflow-def* run on Nemo.

results, see [16].

BLR applied to ParFlow data gives particularly good results as a slight compression (ϵ=1 x 10^{-16}) is enough to already decrease flops by a factor 5 and timing by a factor 2 with no loss of accuracy on the test-case (see Figure 37). Still, with 5 minutes for a complete solve with BLR on the global system, MUMPS timings stay high compared to iterative schemes on particular problems (see Table 26). Also memory can be prohibitive: with 30x12 cores MUMPS needs 13.5GB memory on most consuming process and 12GB on average.

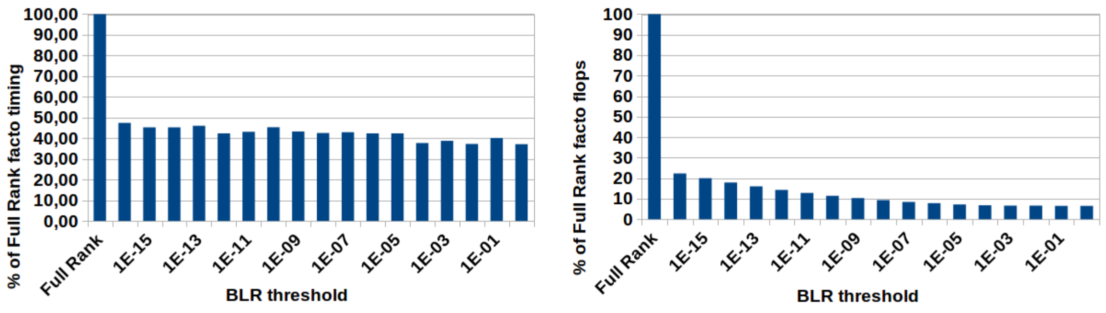
As a new approach to solving the linear systems in ParFlow, we decided to make use of MUMPS+BLR as a preconditioner for a simple iterative scheme.

Results
In the following, we focus on an iteration of ParFlow simulation (*ex10.c*) with parameters:

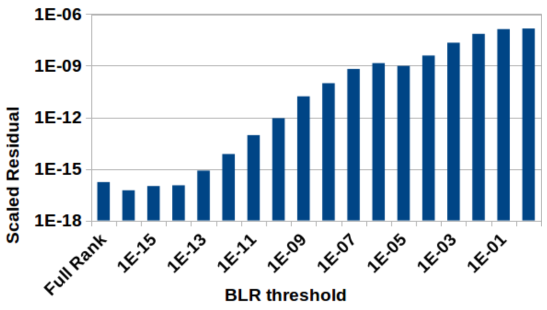
- #MPI=100,
- H=1.0; Nx=Ny=160; Nz=120; num=3.0; atol=1 x 10^{-15}; rtol=1 x 10^{-6}; divtol=1 x 10^{+5},
- Length of domain: H=1.0,
- #Cells in x,y,z: Nx=Ny=160 and Nz=120,
- Std deviation in the log-normal random number generation: num=3,
- Absolute/relative/diversion tolerance: atol=1 x 10^{-15}, rtol=1 x 10^{-6} and divtol=1 x 10^{+5},

The method can be tested directly through PETSc using command-line options. We tried several configurations:

1. Conjugate Gradient (CG) preconditioned with Block-Jacobi (BJ) and incomplete LU level 0 (ILU0): *-ksp_type cg -pc_type bjacobi -sub_pc_type ilu*,
2. CG preconditioned with BJ and incomplete Cholesky (ICC): *-ksp_type cg -pc_type*



(a) Factorisation timing and flops



(b) Accuracy of solution

Figure 37: MUMPS used with Block Low Rank Approximation on ParFlow matrix *parflow-def* with 30x12 cores on Nemo.

bjacobi -sub_pc_type icc,

3. CG preconditioned with MUMPS(+BLR) on complete system: *-ksp_type cg -pc_type cholesky -pc_factor_mat_solver_package mumps*
4. CG preconditioned with MUMPS(+BLR) as solver for BJ subsystems:
   - *-ksp_type cg -pc_type bjacobi -pc_bjacobi_blocks 100/X -sub_pc_type cholesky -sub_pc_factor_mat_solver_package mumps*,
   - We have tried several size of BJ subdomains (see Table 27) and selected the trade-off with X MPI in MUMPS per BJ block (the default is 1 block per MPI).

To activate BLR, we only need to add the option *-mat_mumps_icntl_35 1 -mat_mumps_cntl_7 1 x* 10^{-16} where 1 x 10^{-16} is the epsilon value.

The results are gathered in Table 28.

PSBLAS and MLD2P4 for ParFlow

Activity type	WP1 support
Contributors	Ambra Abdullahi Hassan (University of Rome “Tor Vergata”, Italy), Pasqua D’Ambra (CNR, Italy), Daniela di Serafino (University of Campania “L. Vanvitelli”, Italy), Salvatore Filippone (Cranfield University, UK), Wendy Sharples (JSC, Germany) for WP4

The improved versions of PSBLAS and MLD2P4 developed during the EoCoE project (see Deliverable D1.7) have been applied to a data set from ParFlow. The goal of this work was to provide a sound basis for the selection of solvers and preconditioners for future integration and tuning into the application code.

Data Set

The set of linear systems comes from the numerical simulation of the filtration of 3D incompressible single-phase flows through anisotropic porous media, carried out at the Jülich Supercomputing Centre (JSC) within WP4 (*Water for Energy*). The linear systems arise from the discretization of an elliptic equation with no-flow boundary conditions, modelling the pressure field, which is obtained by combining the continuity equation with Darcy’s law [1]. The discretization is performed by a cell-centered finite volume scheme (two-point flux approximation) on a Cartesian grid. The anisotropic permeability tensor in the elliptic equation is randomly computed from a lognormal distribution with mean 1 and three standard deviation values, 1, 2 and 3, corresponding to three types of systems with symmetric positive definite matrices and a classical seven-diagonal sparsity pattern, denoted by MAT1, MAT2 and MAT3. These systems can be regarded as simplified samples of systems arising in ParFlow.

First experiments were performed with matrices of dimension 10^6 and a 6940000 nonzero entries, generated by using a Matlab mini-app provided by JSC. In order to perform a weak scalability analysis, a Fortran code reproducing the mini-app was developed within WP1 to obtain larger sample matrices of the same type.

In order to assess the behaviour of different MLD2P4 preconditioners on the selected test case and choose the best ones for the application of interest, an evaluation in terms of execution time, strong and weak scalability, and linear solver iterations was carried out.

Results on the Data Set

A first set of esperiments, aimed at identifying the best preconditioners available from MLD2P4 for the selected linear systems, was performed on a linux cluster, named yoda, operated by the Naples Branch of the CNR Institute for High-Performance Computing and Networking. Its compute nodes consist of 2 Intel Sandy Bridge E5-2670 8-core processors and 192 GB of RAM, connected via Infiniband. The tests were carried out on the matrices of dimension 10^6, using 1, 2, 4, 8, 16, 32, and 64 cores, running as many parallel processes.

Figure 38 shows the execution times (in seconds) and the speedups obtained by applying the Conjugate Gradient (CG) solver available in PSBLAS with four multilevel preconditioners and a one-level block-Jacobi preconditioner from MLD2P4 (the times include the preconditioner setup). The corresponding numbers of preconditioned CG iterations are reported in Table 29. The multilevel preconditioners performed a V-cycle, using different

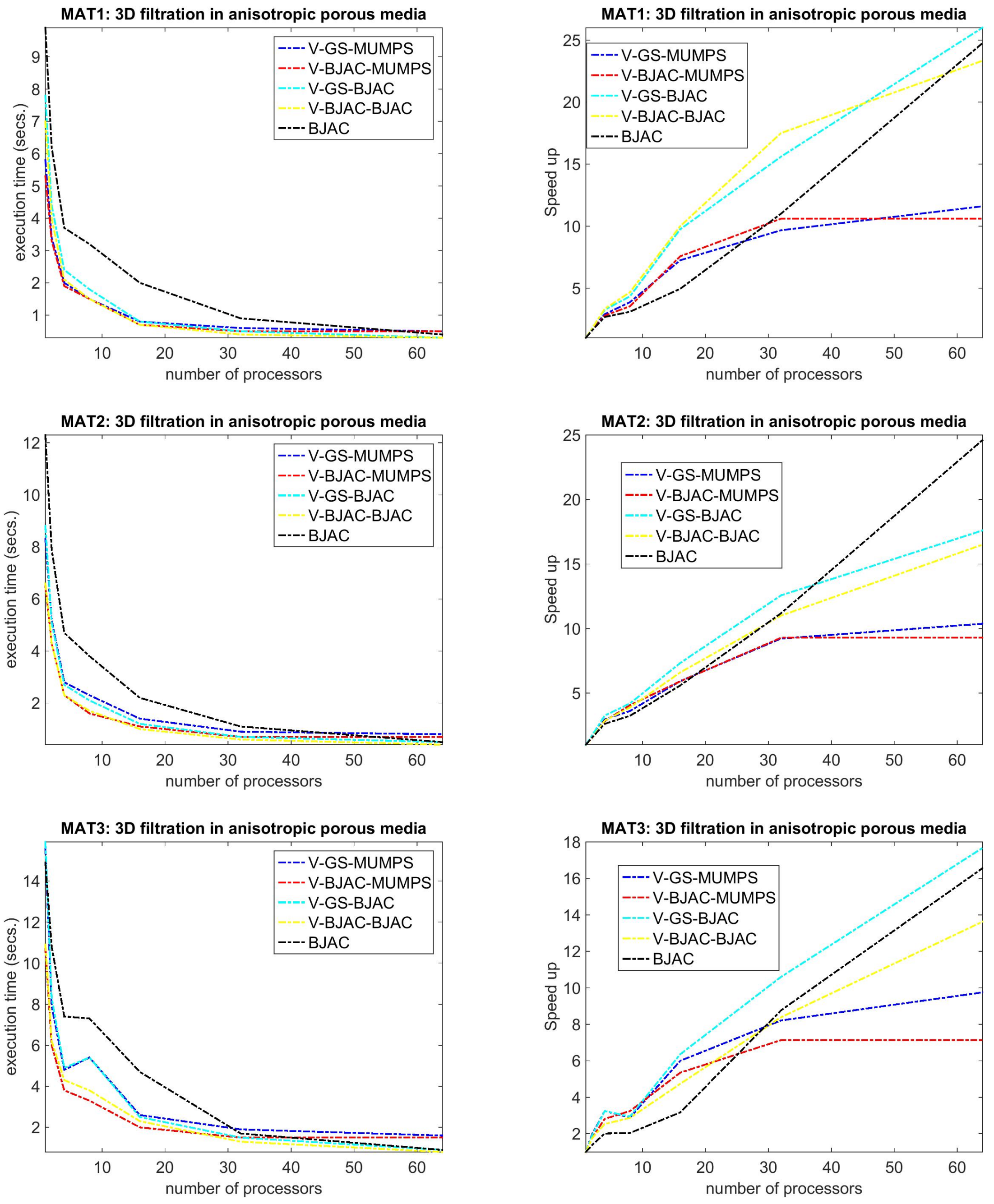


Figure 38: Linear systems from groundwater modelling: execution time and speedup on yoda. Top: MAT1, middle: MAT2, bottom: MAT3.

smoothers and coarsest-level solvers. Specifically, V-GS-MUMPS and V-BJAC-MUMPS used 1 forward/backward hybrid Gauss-Seidel sweep and 1 block-Jacobi sweep as pre/post-smoother, respectively, and applied the sparse LU factorization from MUMPS on the coarsest-level system, replicated in all the processes; V-GS-BJAC and V-BJAC-BJAC used 1 forward/backward Gauss-Seidel sweep and 1 block-Jacobi sweep as pre/post-smoother, respectively, and 10 block-Jacobi sweeps on the coarsest matrix, distributed among the processes. The ILU(0) factorization was applied to the blocks in the block-Jacobi sweeps, in both the multilevel and the one-level preconditioners. The zero vector was used as starting guess and the preconditioned CG iterations were stopped when the 2-norm of the residual achieved a reduction by a factor of 10^{-6}. A generalized row-block distribution of the matrices, obtained by using the METIS graph partitioner [9], was chosen.

Further experiments were carried out on the IBM MareNostrum 4 supercomputer, operated by BSC. We tested the preconditioners mentioned above, with the only difference that the sparse LU factorization from UMFPACK was applied when the coarsest matrix was replicated in all the processes. The execution times and speedups obtained on MAT3 are shown in Figure 39.

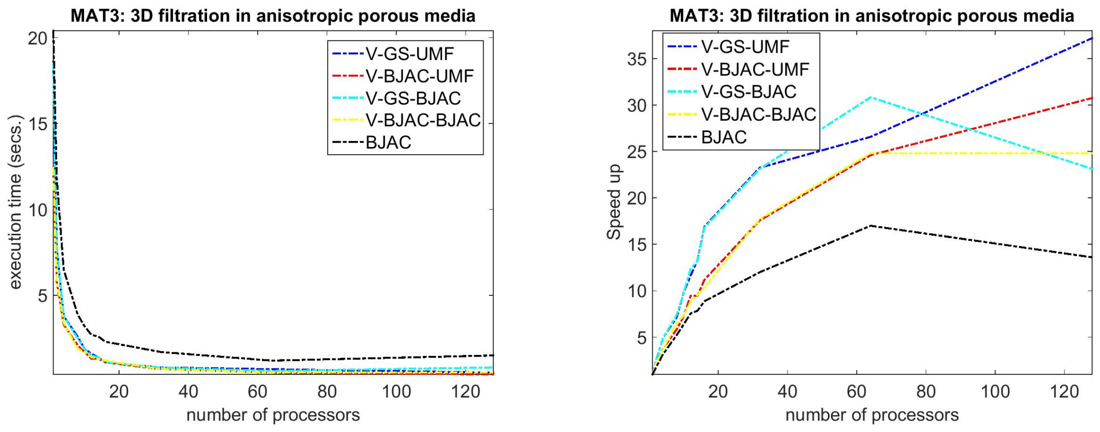


Figure 39: Linear system MAT3 from groundwater modelling: execution time and speedup on MareNostrum.

For these linear systems, all the multilevel preconditioners are generally superior than the one-level preconditioner. However, when the number of cores increases, the performance of the multilevel preconditioners using the exact coarsest-level solver deteriorates, while the multilevel preconditioners applying the distributed iterative solver to the coarsest system still appear efficient. When the overall size of the matrix is kept constant, the execution time of the block-Jacobi preconditioner becomes comparable with that of the multilevel preconditioners using the distributed iterative coarsest-level solver, although the former requires a much larger number of iterations. For all the preconditioners, the execution time increases and the speedup decreases as the anisotropy of the problem grows; this agrees with the larger number of CG iterations that are required, and with the well-known memory-bound nature of this computation. Nevertheless, the preconditioners implemented in MLD2P4 show reliability and robustness with respect to anisotropy.

On the basis of the previous experiments, the V-cycle preconditioners using the smoothed aggregation, forward/backward hybrid Gauss-Seidel (FBGS) or the block-Jacobi (BJAC) smoother, and 10 BJAC sweeps as coarsest-level solver were selected as the best preconditioners to be used with the CG solver on the selected matrices. A weak scaling analysis was performed with these preconditioners on the CRESCO cluster operated by ENEA (40 nodes, each consisting of 2 sockets with 8 Intel Xeon E5-2630 v3 processors, 2.4 GHz and

64 GB RAM, connected by Infiniband). Like in the previous tests, a generalized row-block distribution of the matrix was obtained via METIS; 15500 matrix rows per core were considered, achieving a system dimension of about 16 million on 1024 cores.

The scalability of the preconditioner build phase and of the solve phase is reported in Figure 40 for MAT1 and MAT2. Taking into account the well-known memory-bound nature of the computation, a satisfactory weak scaling is achieved in the solve phase. The preconditioner build phase is less scalable, but in many applications the preconditioner can be reused, thus reducing the impact of this phase on the overall performance.

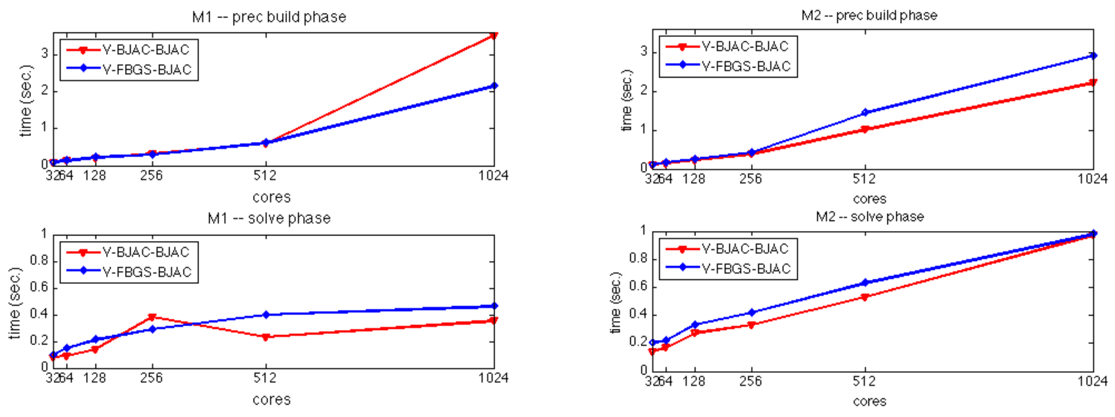


Figure 40: Linear systems MAT1 and MAT2 from groundwater modelling: weak scaling on CRESCO.

In conclusion, PSBLAS and MLD2P4 appear good candidates for the exploitation in parallel simulations of flows in heterogeneous porous media such as Parflow. Furthermore, the possibility of combining various smoothers and coarsest-level solvers provides some flexibility for achieving a good tradeoff between efficiency and robustness, depending on the problem and the parallel computer.

Table 23: Performance metrics for test case 1 on the JURECA cluster for ParFlow and ParFlow + p4est on 24 processors. The first column is the measurement from ParFlow v320, the last column is the measurement from ParFlow + p4est. Metrics that have been added (removed) during the project are marked with n.m. (o.m.).

	Metric name	01'2018 Parflow.v320	01'2018 ParFlow.v320 + p4est
Global	Total Time (s)	5	2
	Time IO (s)	0.1	0.1
	Time MPI (s)	0.6	0.4
	Memory vs Compute Bound	0.8	0.7
	Load Imbalance (%)	12.0	9.6
IO	IO Volume (MB)	183.1	183.1
	Calls (nb)	0	0
	Throughput (MB/s)	1570.4	1551.2
	Individual IO Access (kB)	0.0	0.0
MPI	P2P Calls (nb)	11808	11537
	P2P Calls (s)	0.2	0.2
	P2P Calls Message Size (kB)	4.0	4.1
	Collective Calls (nb)	1008	1022
	Collective Calls (s)	0.2	0.2
	Coll. Calls Message Size (kB)	0.4	0.4
	Synchro / Wait MPI (s)	0.3	0.2
	Ratio Synchro / Wait MPI (%)	56.3	57.5
	Message Size (kB)	o.m.	o.m.
	Load Imbalance MPI	o.m.	o.m.
Mem	Memory Footprint	45964 kB	32168 kB
	Cache Usage Intensity	0.77	0.77
	RAM Avg Throughput (GB/s)	o.m.	o.m.
Core	IPC	1.84	1.68
	Runtime without vectorisation (s)	3	3
	Vectorisation speedup factor	0.6	1.50
	Runtime without FMA (s)	2	2
	FMA speedup factor	0.4	1.00

Table 24: Performance metrics for test case 1 on the JURECA cluster for ParFlow and ParFlow + p4est on 24 processors. The first column is the measurement from ParFlow v320, the last column is the measurement from ParFlow + p4est. Metrics that have been added (removed) during the project are marked with n.m. (o.m.).

	Metric name	01'2018 Parflow.v320	01'2018 ParFlow.v320 + p4est
Global	Total Time (s)	8	5
	Time IO (s)	0.3	0.7
	Time MPI (s)	2.4	1.4
	Memory vs Compute Bound	0.9	1.2
	Load Imbalance (%)	23.2	25.2
IO	IO Volume (MB)	366.3	366.3
	Calls (nb)	0	0
	Throughput (MB/s)	1122.4	549.3
	Individual IO Access (kB)	0.0	0.0
MPI	P2P Calls (nb)	12603	12327
	P2P Calls (s)	0.9	0.5
	P2P Calls Message Size (kB)	4.0	4.1
	Collective Calls (nb)	1028	1042
	Collective Calls (s)	0.8	1.0
	Coll. Calls Message Size (kB)	0.9	0.9
	Synchro / Wait MPI (s)	0.9	1.2
	Ratio Synchro / Wait MPI (%)	37.0	46.1
	Message Size (kB)	o.m.	o.m.
	Load Imbalance MPI	o.m.	o.m.
Mem	Memory Footprint	118176 kB	26180 kB
	Cache Usage Intensity	0.81	0.80
	RAM Avg Throughput (GB/s)	o.m.	o.m.
Core	IPC	1.87	1.81
	Runtime without vectorisation (s)	7	4
	Vectorisation speedup factor	0.88	0.80
	Runtime without FMA (s)	7	5
	FMA speedup factor	0.88	1.00

Table 25: Characteristics of a test matrix generated from ParFlow MATLAB code *FD_3D_TwoPointFluxApproximation_NoLoop_modifYN.m* with default parameters: H=1.0; Nx=Ny=Nz=200; num=3.0; rtol=1 x 10^{-7}; maxiter=100.

Matrix	**unknowns**	**non-zeros per line**	**conditioning**
parflow-def	8.00×10^6	6.97	6.62×10^8

Table 26: Comparison of MUMPS solver with a Conjugate Gradient (CG) preconditioned with Block-Jacobi (BJ) and incomplete LU level 0 (ILU0) in an iteration of ParFlow simulation (*ex10.c*) with parameters: number of MPI processes=100; H=1.0; Nx=Ny=160; Nz=120; num=3; atol=1 x 10^{-15}; rtol=1 x 10^{-6}; divtol=1 x 10^{+5}.

Solver	**MUMPS**	**MUMPS+BLR ϵ=1 x 10^{-16}**	**CG/BJ+ILU0**
Timing	183.60	129.70	4.16
Residual	3.99 x 10^{-10}	1.60 x 10^{-10}	7.49 x 10^{-06}
#Iterations	1	1	1588

Table 27: Comparison of several #Blocks for Block-Jacobi, run with 100 MPI on EOS. By default, there is 1 block per MPI, so 100 blocks. Analysis/Facto/Solve timings are specific to MUMPS.

#Jacobi blocks	**#MPI per blk.**	**Timing**				**Residual**	**#Iters.**
		Total	Analysis	Facto.	Solve		
100	1	122.6	0.30	0.47	0.06	8.61 x 10^{-06}	1514
50	2	255.1	0.51	0.34	0.07	2.45 x 10^{-05}	957
25	4	205.3	1.42	1.85	0.09	3.50 x 10^{-06}	723
10	10	122.3	3.53	2.04	0.10	4.18 x 10^{-06}	538

Table 28: Comparison of several approaches to solve an iteration of ParFlow simulation (*ex10.c*) with previously defined parameters, run with 100 MPI on EOS.

Solver	**Total time**	**Residual**	**#Iterations**
CG/BJ+ILU0	4.16	7.49 x 10^{-06}	1588
CG/BJ+ICC	6.18	8.09 x 10^{-06}	1588
CG/MUMPS	177.10	3.88 x 10^{-10}	1
CG/MUMPS+BLR ϵ=1 x 10^{-16}	135.20	1.38 x 10^{-10}	1
CG/MUMPS+BLR ϵ=1 x 10^{-1}	85.18	2.20 x 10^{-02}	divergence
CG/BJ+MUMPS	118.60	8.61 x 10^{-06}	1514
CG/BJ+MUMPS+BLR ϵ=1 x 10^{-16}	116.80	8.61 x 10^{-06}	1514
CG/BJ+MUMPS+BLR ϵ=1 x 10^{-1}	112.90	8.61 x 10^{-06}	1514

Table 29: Linear systems from groundwater modelling: number of CG iterations.

procs	BJAC	V-GS-MUMPS	V-BJAC-MUMPS	V-GS-BJAC	V-BJAC-BJAC
			MAT1		
1	288	15	13	29	26
2	331	15	14	30	27
4	341	15	14	27	25
8	331	15	15	28	25
16	343	18	15	19	17
32	356	17	15	19	17
64	351	17	15	20	18
			MAT2		
1	358	32	19	33	20
2	420	38	29	38	29
4	429	34	26	34	27
8	391	33	27	35	28
16	424	35	29	37	31
32	435	34	29	36	30
64	482	38	28	40	30
			MAT3		
1	469	72	43	74	44
2	607	68	46	70	47
4	618	64	54	65	56
8	625	80	64	82	60
16	680	63	60	65	62
32	716	78	71	81	72
64	700	77	72	81	72

References

[1] Jørg E. Aarnes, Tore Gimse, and Knut-Andreas Lie. “An Introduction to the Numerics of Flow in Porous Media using Matlab”. In: *Geometric Modelling, Numerical Simulation, and Optimization: Applied Mathematics at SINTEF*. Ed. by Geir Hasle, Knut-Andreas Lie, and Ewald Quak. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 265–306. ISBN: 978-3-540-68783-2. DOI: 10.1007/978-3-540-68783-2_9. URL: https://doi.org/10.1007/978-3-540-68783-2_9.

[2] Pedro Alonso et al. “Tools for Power-Energy Modelling and Analysis of Parallel Scientific Applications”. In: *2012 41st International Conference on Parallel Processing*. IEEE, Sept. 2012, pp. 420–429. ISBN: 978-1-4673-2508-0. DOI: 10.1109/ICPP.2012.57. URL: http://ieeexplore.ieee.org/document/6337603/.

[3] Steven F Ashby and Robert D Falgout. “A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations”. In: 124.1 (1996), pp. 145–159.

[4] Carsten Burstedde, Jose A Fonseca, and Stefan Kollet. “Enhancing speed and scalability of the ParFlow simulation code”. In: *arxiv.org* (Feb. 2017). arXiv: 1702.06898. URL: http://arxiv.org/abs/1702.06898.

[5] Philip Carns et al. “Understanding and Improving Computational Science Storage Access through Continuous Characterization”. In: *ACM Transactions on Storage* 7.3 (Oct. 2011), pp. 1–26. ISSN: 15533077. DOI: 10.1145/2027066.2027068. URL: http://dl.acm.org/citation.cfm?doid=2027066.2027068.

[6] Markus Geimer et al. “The Scalasca performance toolset architecture”. In: *Concurrency and Computation: Practice and Experience* 22.6 (Apr. 2010), pp. 702–719. ISSN: 15320626. DOI: 10.1002/cpe.1556. URL: http://doi.wiley.com/10.1002/cpe.1556.

[7] Christopher January et al. “Allinea MAP: Adding Energy and OpenMP Profiling Without Increasing Overhead”. In: *Tools for High Performance Computing 2014*. Cham: Springer International Publishing, 2015, pp. 25–35. DOI: 10.1007/978-3-319-16012-2_2. URL: http://link.springer.com/10.1007/978-3-319-16012-2%7B%5C_%7D2.

[8] Jim E. Jones and Carol S. Woodward. “Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems”. In: *Advances in Water Resources* 24.7 (July 2001), pp. 763–774. ISSN: 03091708. DOI: 10.1016/S0309-1708(00)00075-0. URL: http://linkinghub.elsevier.com/retrieve/pii/S0309170800000750.

[9] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM J. Sci. Comput.* 20.1 (Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: 10.1137/S1064827595287997. URL: http://dx.doi.org/10.1137/S1064827595287997.

[10] Andreas Knüpfer et al. “Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”. In: *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*. Ed. by Holger Brunst et al. Springer, 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7.

[11] Stefan J. Kollet and Reed M. Maxwell. “Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model”. In: *Water Resources Research* 44.2 (Feb. 2008), n/a–n/a. ISSN: 00431397. DOI: 10.1029/2007WR006004. URL: http://doi.wiley.com/10.1029/2007WR006004.

[12] Stefan J. Kollet and Reed M. Maxwell. “Integrated surface–groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model”. In: *Advances in Water Resources* 29.7 (July 2006), pp. 945–958. ISSN: 03091708. DOI: 10.1016/j.advwatres.2005.08.006. URL: http://www.sciencedirect.com/science/article/pii/S0309170805002101.

[13] Stefan J. Kollet et al. “Proof of concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources”. In: *Water Resources Research* 46.4 (2010), pp. 1–7. ISSN: 00431397. DOI: 10.1029/2009WR008730.

[14] Wolfgang Kurtz et al. “TerrSysMP–PDAF (version 1.0): a modular high-performance data assimilation framework for an integrated land surface–subsurface model”. In: *Geoscientific Model Development* 9.4 (Apr. 2016), pp. 1341–1360. ISSN: 1991-9603. DOI: 10.5194/gmd-9-1341-2016. URL: http://www.geosci-model-dev-discuss.net/8/9617/2015/%20http://www.geosci-model-dev.net/9/1341/2016/.

[15] J Labarta et al. “Scalability of Visualization and Tracing Tools”. In: 33 (2006), pp. 3–. URL: http://www.fz-juelich.de/nic-series/volume33.

[16] Théo Mary. “Block Low-Rank multifrontal solvers: complexity, performance, and scalability”. PhD thesis. UT3, 2017.

[17] Reed M. Maxwell. “A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling”. In: *Advances in Water Resources* 53 (2013), pp. 109–117. ISSN: 03091708. DOI: 10.1016/j.advwatres.2012.10.001.

[18] Reed M. Maxwell and Norman L. Miller. “Development of a Coupled Land Surface and Groundwater Model”. In: *Journal of Hydrometeorology* 6.3 (June 2005), pp. 233–247. ISSN: 1525-755X. DOI: 10.1175/JHM422.1. URL: http://journals.ametsoc.org/doi/abs/10.1175/JHM422.1.

[19] Reed M Maxwell et al. “Development of a Coupled Groundwater–Atmosphere Model”. In: *Monthly Weather Review* 139.1 (Jan. 2011), pp. 96–116. ISSN: 0027-0644. DOI: 10.1175/2010MWR3392.1. URL: http://journals.ametsoc.org/doi/abs/10.1175/2010MWR3392.1.

[20] Lars Nerger and Wolfgang Hiller. “Software for ensemble-based data assimilation systems—Implementation strategies and scalability”. In: *Computers {&} Geosciences* 55 (June 2013), pp. 110–118. ISSN: 00983004. DOI: 10.1016/j.cageo.2012.03.026. URL: http://www.sciencedirect.com/science/article/pii/S0098300412001215.

[21] Ashay Rane et al. “Unification of Static and Dynamic Analyses to Enable Vectorization”. In: Springer, Cham, 2015, pp. 367–381. DOI: 10.1007/978-3-319-17473-0_24. URL: http://link.springer.com/10.1007/978-3-319-17473-0%5C_24.

[22] Pavel Saviankou et al. “Cube v4: From Performance Report Explorer to Performance Analysis Tool”. In: *Procedia Computer Science* 51 (June 2015), pp. 1343–1352. ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.320.

[23] W Sharples et al. “Best practice regarding the three P’s: profiling, portability and provenance when running HPC geoscientific applications”. In: *Geoscientific Model Development Discussions* 2017 (2017), pp. 1–39. DOI: 10.5194/gmd-2017-242. URL: https://www.geosci-model-dev-discuss.net/gmd-2017-242/.

[24] P Shrestha et al. “A scale-consistent Terrestrial Systems Modeling Platform based on COSMO, CLM and ParFlow.” In: *Monthly Weather Review* 142.9 (Apr. 2014), pp. 3466–3483. ISSN: 0027-0644. DOI: 10.1175/MWR-D-14-00029.1. URL: http://dx.doi.org/10.1175/MWR-D-14-00029.1.

[25] Ilya Zhukov et al. “Scalasca v2: Back to the Future”. In: *Tools for High Performance Computing 2014*. Cham: Springer International Publishing, 2015, pp. 1–24. DOI: 10.1007/978-3-319-16012-2_1. URL: http://link.springer.com/10.1007/978-3-319-16012-2%5C_1.

Acknowledgement

In case of PSNC the scientific/academic work is co-financed from financial resources for science in the years 2016-2018 granted for the realization of the international project financed by Polish Ministry of Science and Higher Education (agreement number 3543/H2020/2016/2)

10. Shemat

Overview

SHEMAT-Suite simulates flow, heat and species transport in porous media, as well as geochemical rock reactions, for applications regarding geothermal energy. It has inverse capabilities (Monte-Carlo, EnKF, Bayes Inversion) as well as functionalities for two-phase flow to simulate CO_2 sequestration. The code is developed at RWTH (University of Aachen) and is written in Fortran, using MPI and OpenMP for parallelisation.

Code team:

- Rene Halver (JUELICH) for WP1
- Johanna Bruckmann (RWTH) for WP4

Application Support

Activity type	WP1 support
Contributors	Sebastian Lührs (JUELICH, WP1), Rene Halver (JUELICH, WP1), Johanna Bruckmann (RWTH, WP4), Henrik Büsing (RWTH, WP4), Jan Niederau (RWTH, WP4)

Implementation of HDF5 as a parallel I/O input format for SHEMAT-Suite

The target of this support activity was the optimization of the I/O behavior of SHEMAT-Suite by adding new HDF5[5] capabilities for the input parsing process.

The existing established input format of SHEMAT-Suite allows the usage of a mix of ASCII and HDF5 (in a preliminary version) input files. The HDF5 files are referenced in the ASCII file. Large datasets can either be provided directly in the ASCII file or via the separate HDF5 file. The existing HDF5 capabilities were not used quite often and could only be used for a subset of input parameters. So far the HDF5 input files are generated by SHEMAT-Suite itself. So these files could only be used to allow re-usage of datasets in a secondary run (e.g. to restart with checkpoint data). New datasets from scratch could only be defined via the ASCII format.

The handling of the ASCII file format could be rather slow if all different input variables are written to a single input file (which is a common input case). Instead it is also possible to distribute the data over multiple files. In the case of using a single input file and a file size larger then 100 MB the input file read duration of SHEMAT-Suite could take multiple minutes.

To allow a more flexible HDF5 input format (without using SHEMAT-Suite for conversion), parallel I/O capabilities and to avoid the long input file handling of large ASCII input files, the code developers asked for application support by WP1 to implement a new HDF5 based input strategy.

[5]https://support.hdfgroup.org/HDF5

Structure

To allow an easy implementation of the new input format, two new parts within SHEMAT-Suite were implemented as part of this support activity:

- A conversion script which efficiently converts the existing input format to the new input format. By using this intermediate script solution, all preprocessing steps can stay unchanged and are not affected by this implementation.
- Adding new HDF5 capabilities to SHEMAT-Suite to allow parsing of the new input files.

The output behavior of SHEMAT-Suite was not changed.

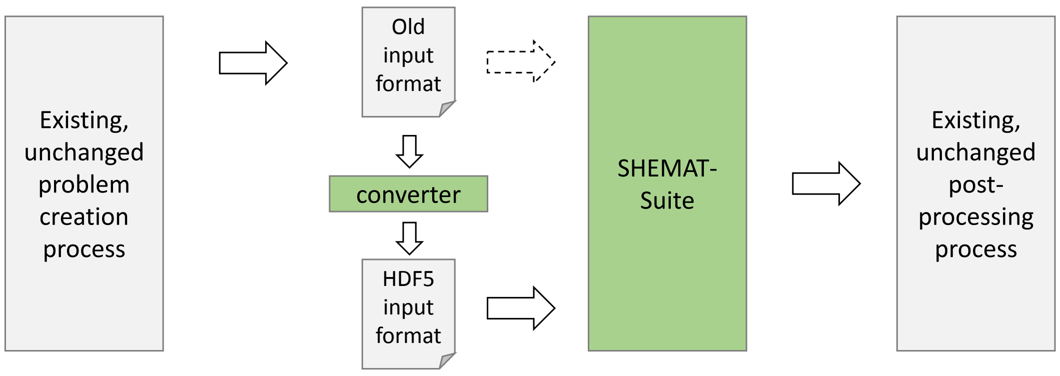

Existing, unchanged problem creation process
Old input format
converter
HDF5 input format
SHEMAT-Suite
Existing, unchanged post-processing process

Figure 41: Updated SHEMAT-Suite I/O workflow layout. The parts which were added/changed in the activity are marked in green.

The old ASCII based input format contains several different variables to parameterize SHEMAT-Suite. In addition this format also supports different ways to represent these data (e. g. by using Fortran based repeated values like `10 * 2.3` to reuse the same value ten times). The different variables and the different formats must be interpreted by the conversion script. The new implementation focuses on the larger variables in context of data size, smaller scalar values are currently not converted and stay unchanged in the old input file. The parsing process automatically change between the old and the new input format if a variable is found which is not converted so far. This also allows easy addition of new variables in the future, because these could be added to the old input format and do not need to be directly added into the conversion process. On the other site additional variables can be converted one after each other. Not all variables together has to be converted to still allow program execution. Until now nearly thirty variables were already moved from the old to the new format.

Implementation

The existing ASCII format uses header lines to mark the different variables in the input file.

As an example the following lines describe the general grid structure:

```
# grid
362 287 72

# delx
362*10.

# dely
287*10.

# delz
72*10.
```

These examples are rather short entries because they use the repeat feature of Fortran to avoid repeating multiple values. Other entries might contain several MB of data. Due to the existing parsing implementation of SHEMAT-Suite, the input file is searched for each individual header line starting at the beginning of the file. This process can extremely slow down the parsing process if header entries do not exist (e.g. if they are optional), because the whole file has to be scanned multiple times.

The new conversion script is written in Python using the h5py[6] HDF5 Python bindings. Because it reads the original input file, the existing parsing process of SHEMAT-Suite has to be reimplemented to support as many input features as possible. To avoid the same header searching bottleneck, the file is only scanned once to mark all header lines. After this, all existing headers can easily be reached by jumping directly to the specific file position.

HDF5 does not support some of the features which were present in the old ASCII format (like the Fortran repeat syntax). Such values are now automatically converted to a general format using a fixed structure. Repeated values are now also stored directly in the file, which increases the file size, but also allows to use a file more easily within other applications.

The conversion script will produce the following HDF5 structure to store the grid layout:

```
GROUP "grid" {
        DATASET "delx" {
            DATATYPE  H5T_IEEE_F64LE
            DATASPACE  SIMPLE { ( 362 ) / ( 362 ) }
        }
        DATASET "dely" {
            DATATYPE  H5T_IEEE_F64LE
            DATASPACE  SIMPLE { ( 287 ) / ( 287 ) }
        }
        DATASET "delz" {
            DATATYPE  H5T_IEEE_F64LE
            DATASPACE  SIMPLE { ( 72 ) / ( 72 ) }
        }
    }
```

Once a variable is extracted, it is deleted from the ASCII file to only keep the unconverted

[6]http://www.h5py.org/

values. In addition a new entry is added to the ASCII file automatically pointing to the new HDF5 data file. The ASCII file itself still remains the main entry point for SHEMAT-Suite. Depending on the presence of the HDF5 file link entry, the new or old parsing process is triggered.

Within SHEMAT-Suite a new HDF5 parsing interface was implemented. This interface wraps the most common reader functions and some additional help routines. The new HDF5 data file is opened once in the beginning and is kept in a global reachable file handle until all input datasets are loaded. The data layout (dimension, type and structure) in the HDF5 file was selected based on the SHEMAT-Suite internal data representation to avoid any conversion process.

Parallel I/O support for distributed data

In addition to these improvements and the direct HDF5 conversion capabilities, the new input format also allows the usage of distributed I/O calls calls in future implementations of SHEMAT Suite. Instead of reading global datasets with each individual processor, the usage of distributed I/O calls can help to avoid memory scalability problems once the application will be executed on larger scales.

In order to implement the distributed I/O calls in a first step, the *mpifw* branch of SHEMAT-Suite was used, providing a MPI-parallel version of the code. Because during the implementation of parallel HDF5 I/O routines it was discovered that the MPI parallelisation was not working as intended, e.g. a asynchronous adaptive time refinement led to deadlocks in some cases or data was still collected in global arrays defeating the purpose of a distributed memory approach, it was decided to change the code-base to another MPI-based branch of the code. Therefore the implementation was continued on the *PETSHEM_DD* branch, which also provides a MPI parallel version of the code using PETSC[7] for data distribution.

This PETSC support was implemented by WP4, in order to decrease the amount of memory each process required. Previously every process stored a copy of all data arrays for the whole system and this limited the size of systems that could be simulated and avoided further scalability. With the use of PETSC it is now possible to compute larger systems, as the data is now distributed between the processes. On the data input side, this distribution had to be adapted to avoid reading all data first and storing only the locally needed data afterwards. For this the HDF5 I/O routines were modified, so that each process only reads the data which are required by it. To do this, HDF5 hyperslabs were used, which allow selective input and output from a HDF5 data file. As a final step, to provide backwards compatibility, the HDF5 output routines were modified, in such a way that they produce a unified HDF5 file, which complies to the old output format, so that different branches and older versions of the code can make use of the data. Before this output wasn't available anymore, as the output structure relied on reading all system data out of a single process. To add this improved output capabilities again HDF5 hyperslabs were used. Together with adapting the output routines, some additional routines (e.g. the calculation of global data position for distributed non regular grids) which were not ported so far to the new PETSC scheme, were also adapted in the scope of this project by WP1.

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

Results

To validate the benefit of the new input parsing process, the input parsing time was measured using different file sizes (of the original ASCII main input file) and different number of cores on the JURECA[8] system. The comparison is done between the original ASCII based input file and the new converted input file. The time for the conversion process itself is included in the new timings.

Figure 42 shows a small input file, were the old and the new input format show nearly the same read duration and scaling behavior.

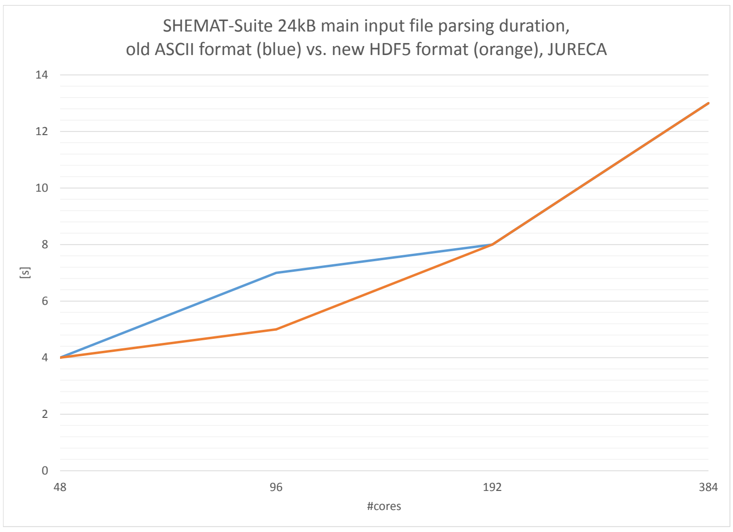


Figure 42: SHEMAT-Suite input parsing time on JURECA using different number of cores and a 24 kB main input file. The old ASCII format is marked in blue, the new HDF5 format is marked in orange.

Figure 43 shows a second input case, using a larger ASCII input file. The ASCII parsing process is already much slower in comparison to the new format due to the header line handling which is mentioned before.

Table 30 shows an (in context of SHEMAT-Suite) large input case, storing more than 200 MB within the single main SHEMAT-Suite ASCII input file. The parsing of this file takes nearly 45 minutes in serial by using the old parsing process. This process could be significantly improved and reduced to less then a minute.

Table 30: Serial SHEMAT-Suite input parsing time on JURECA using a 229 MB main input file.

Parsing time old ASCII format	Parsing time new HDF5 input format
2647s	26s

As shown in the figures, the new input format could speed up the parsing process. Additional variables could easily be added to the conversion process to allow future sustain-

[8]http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html

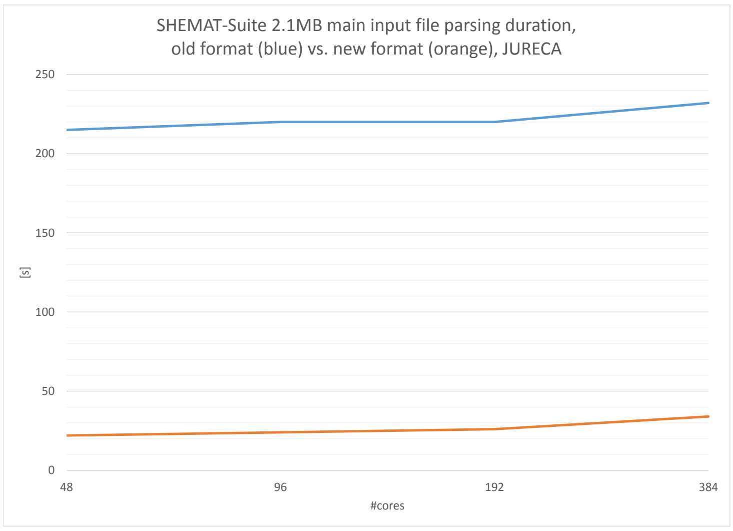


Figure 43: SHEMAT-Suite input parsing time on JURECA using different number of cores and a 2 MB main input file. The old ASCII format is marked in blue, the new HDF5 format is marked in orange.

ability.

As a final result it should be mentioned, that these changes to the code lead to savings in the time spent for computation on different clusters situated in Aachen and Jülich. The relative amount of saved time is estimated to be around 7 % of the total time spent and sums up to about 36 500 core-h. It should be mentioned again, that these savings are the result of a pure improvement of the I/O procedure and no further performance optimizations took place in this support activity, but the new adapted parallel I/O approach for distributed data finally also allows to investigate further scaling with the PETSC implementation of SHEMAT-Suite.

Hiepacs solvers for SHEMAT matrices

Activity type	Consultancy or WP1 support
Contributors	E. Agullo (WP1), L. Giraud (WP1), M. Hastaran (WP1), M. Kuhn (WP1), G. Marait (WP1), H. Buesing (WP4)

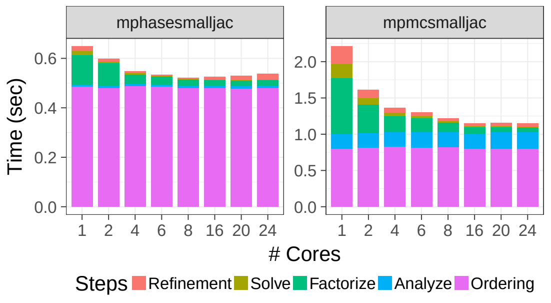
This section deals with the usage `Pastix` [5] and `Maphys` [4] Hiepacs sparse solvers for linear algebra to solve matrices coming from the SHEMAT-Suite. The matrices and their characteristics used for this performance test are given in Table 31.

The following parallel experiments are all performed on the academic platform PlaFRIM (Federative Platform for Research in Computer Science and Mathematics). The part of the cluster in use is composed of 2 Dodeca-core Haswell Intel Xeon E5-2680 @ 2.5 GHz nodes with 128 GB RAM per node.

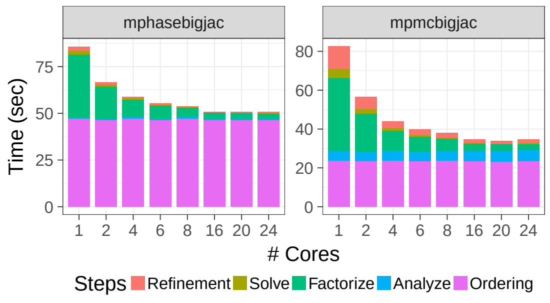
Figure 44 shows `Pastix` solver strong scaling on one node using multi-threading and Fig-

Table 31: Test matrices for Hiepacs solvers

Name	Columns	NNZ_A	NNZ_L	GFLOPS
MPhase Small	49152	438152	6222328	0.94
MPhase Big	3145728	28277768	715196138	825.27
MPMC Small	196608	1957132	19157988	5.61
MPMC Big	4320000	43160408	639275272	736.73



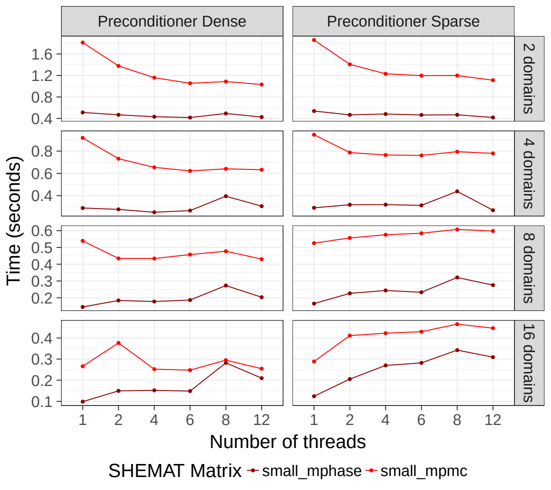
(a) Small matrices



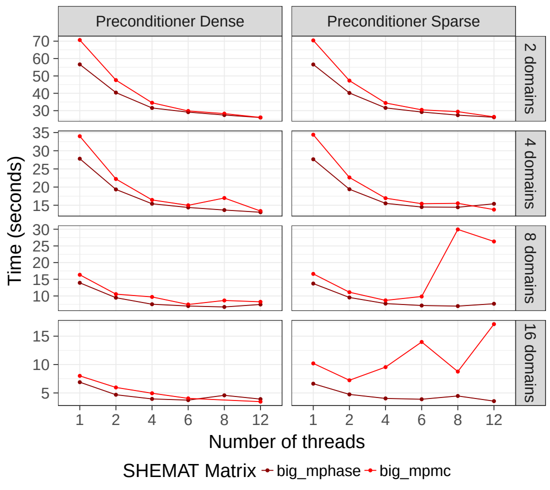
(b) Big matrices

Figure 44: `Pastix` multi-threaded performances on one computational node

ure 45 shows `Maphys` solver strong scaling on several nodes with multi-threading. Globally, these preliminary results show that both `Pastix` and `Maphys` solvers perform well into solving the four matrices. This study could serve as a basis to discuss the opportunity of integrating one of the two solvers into SHEMAT to perform further tests.



(a) Small matrices



(b) Big matrices

Figure 45: `Maphys` multi-threaded performances on 2 to 16 computational nodes

Table 32: Characteristics of matrices extracted from SHEMAT-Suite code, as test-cases on the EoCoE data server. More complex configurations (from the linear solver point of view) could be generated with varying permeability or variable density and viscosity.

Dev-branch	Matrix	unknowns	non-zeros per line	conditioning
Single-phase	mphase_big	3.15 x 10^{+06}	6.99	2.23 x 10^{+07}
	mpmc_big	4.32 x 10^{+06}	9.96	1.84 x 10^{+05}
	fine8	3.15 x 10^{+06}	6.99	2.37 x 10^{+07}
	steam_3D_model_25m	2.09 x 10^{+06}	8.09	7.06 x 10^{+03}
Multi-phase	head_big	8.31 x 10^{+06}	6.91	1.07 x 10^{+03}
	temp_big	8.31 x 10^{+06}	6.89	7.31 x 10^{+04}

MUMPS and ABCD solver for SHEMAT-Suite

Activity type	WP1 support
Contributors	I. Duff (CERFACS, WP1), P. Leleux (CERFACS, WP1), D. Ruiz (IRIT, WP1), F. S. Torun (IRIT-CNRS, WP1), H. Büsing (RWTH, WP4), J. Bruckmann (RWTH, WP4), J. Niederau (RWTH, WP4)

The target of this support activity was the optimization of the linear solver involved in each non-linear Picard-iteration of the code SHEMAT-Suite, where PDE coefficients depend on primary variables (pressure and temperature). This is a 3D problem with regular grid and classical 7 point stencil space discretization (7 diagonals in every Newton block, i.e. max 14 entries per row).

The goal is to obtain new insights into the problem and how to construct new appropriate preconditioners.

Each Picard-iteration implies the solution of a sparse linear system with single right hand side. The systems are unsymmetric, as there is advection in the physics/heterogeneity in the coefficients, and they change (structure and values) for each iteration. Target size of the systems is 10^8 and target timing of the order of the minute.

The current solution uses BiCGStab with ilu0 preconditioner launched through PetSc. The code has 2 development branches:

- single-phase flow with a pure OpenMP parallelism (8-16 cores),
- multi-phase flow with MPI parallelism (run on JUQUEEN up to 1024 cores).

As an example on matrix *head_big* (see Table 32) with 8 OpenMP threads, BiCGStab+ILU0 converges in 60.8s with 429 iterations and a residual of 2-norm 1×10^{-12}.

Method and Results for MUMPS

We will not give extensive details about the use of MUMPS in this section but will focus on the use of this solver to tackle SHEMAT-Suite linear systems. For more details on MUMPS principle and particularly on its latest feature: Block Low Rank Approximation (BLR), see MUMPS Section from deliverable 1.7 Software technology improvement.

Our tests were run mainly on CERFACS cluster Nemo: 1872 nodes with 128GB of memory and 2 sockets of 12 Intel Xeon E5-2680 v3 Haswell CPUs at 2.5GHz. In the following ”P x N cores” stands for P processes with N threads each. The test cases used for preliminary results were those of Table 32.

As new approach to solving the linear systems in SHEMAT-Suite, we decided to make use of MUMPS with BLR as a preconditioner for a simple iterative scheme like GMRES.

The relevance of this approach comes from several observations made after preliminary runs with MUMPS on SHEMAT-Suite extracted matrices:

1. MUMPS is capable of scaling on SHEMAT-Suite matrices as long as there is enough granularity (see Figure 46),
2. For most applications, there is no need to get a solution to machine accuracy: BLR feature approximates the LU factorisation,
3. On a theoretical point of view, BLR has an asymptotic complexity of $O(n^{(4/3)})$ with n being the first dimension of the 3D problem. Geometric multigrid will always be a better choice on purely elliptic problems, however on different problems (ex: Helmholtz or structural mechanics) the gains compared to using an iterative scheme could be greater. For more results, see [3],
4. BLR applied to SHEMAT-Suite data gives particularly good results as a slight compression is enough to already decrease flops and timing by a good factor with no loss (see Figure 47),
5. However, the memory consumption and timings stay too high compared to pure iterative schemes for this particular problem (see Figure 48 and Table 33).

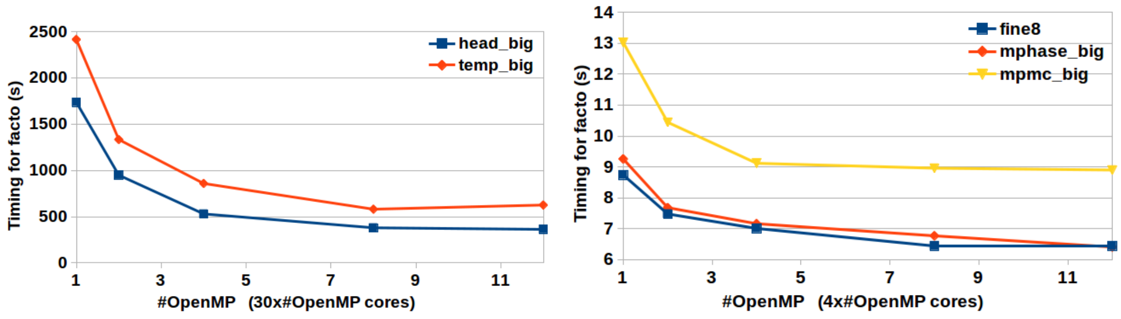
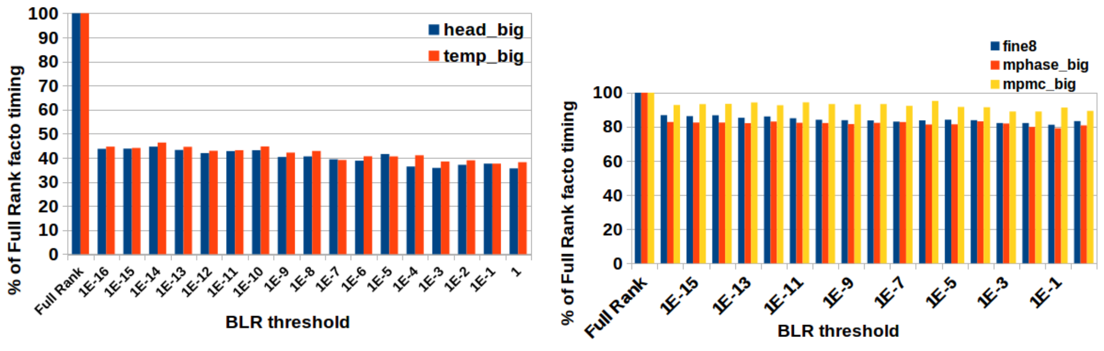


Figure 46: MUMPS factorisation phase depending on #OpenMP **(Left)** for single-phase with 30 MPI and **(Right)** for multi-phase with 4 MPI.

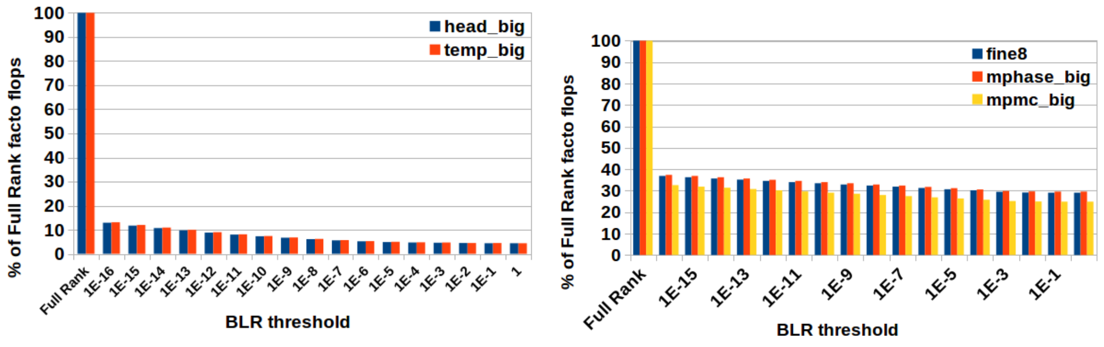
Table 33: Tests on the multiphase system with refinement level 8 on 4 cores: comparison of MUMPS and CPR-AMG (Constrained Pressure Residual Algebraic Multigrid) timings for the full simulation.

Refinement level	**MUMPS (No BLR)**	**MUMPS (BLR $\epsilon = 10^{-16}$)**	**CPR-AMG**
8	7:54.41 min	6:44.31 min	3:57.58 min

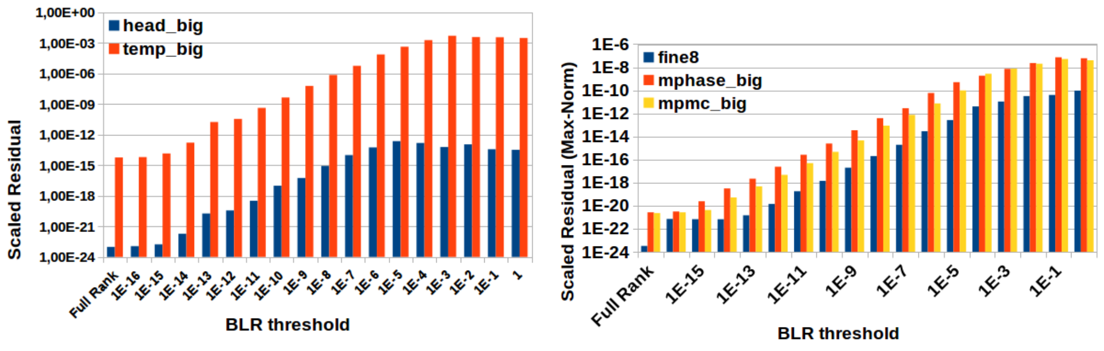
N.B.: Through the runs, we noticed that multi-phase matrices are faster to solve and thus look smaller from the direct solver point of view. We are still investigating this point as



(a) Factorisation timing



(b) Factorisation flops



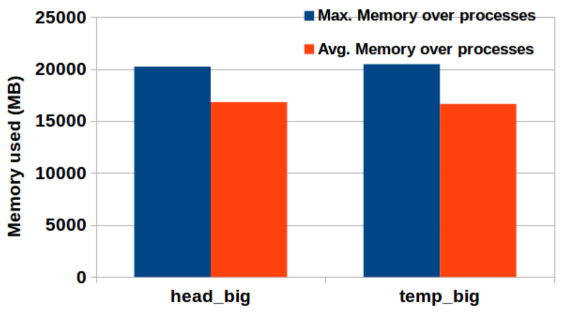
(c) Accuracy of solution

Figure 47: MUMPS used with Block Low Rank Approximation on SHEMAT-Suite **(Left)** single-phase with 30x12 cores and **(Right)** multi-phase with 4x4 cores.

only the geological structure is more intricate for single-phase while physics and structure (2x2 block compared to 1x1) are harder in multi-phase.

The method is also tested directly through PETSc using command-line options. We tested with several configurations in full simulations:

1. MUMPS+BLR as preconditioner on complete system for GMRES:
   -ksp_type gmres -pc_type lu -sub_pc_factor_package mumps -sub_mat_mumps_icntl_35 1 -sub_mat_mumps_cntl_7 1 x 10^{-1},
2. MUMPS+BLR as preconditioner on Block-Jacobi subsystems for GMRES:
   -ksp_type gmres -pc_type bjacobi -sub_pc_type lu -sub_pc_factor_package mumps



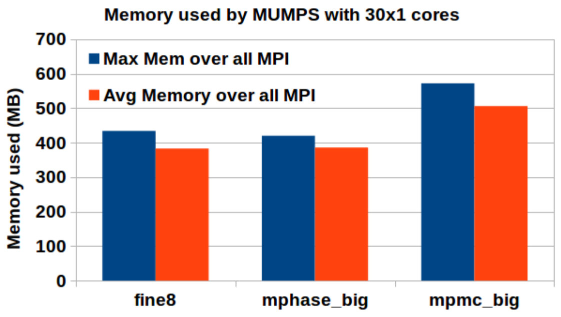


Figure 48: Memory needed by MUMPS with 30x1 cores (Left) for single-phase and **(Right)** for multi-phase.

-sub_mat_mumps_icntl_35 1 -sub_mat_mumps_cntl_7 1 x 10^{-1},

The goal is to handle more precisely the sub-domains from the Block-Jacobi preconditioner with MUMPS to decrease the number of iterations of GMRES. Relaxing the direct solver with BLR decreases MUMPS timing while increasing the number of iterations. However, on these problems, we could not achieve significant improvement in solving time.

Experiments on Steam Problem

There is a special kind of linear problem in SHEMAT-Suite, named "steam", which come from deep geothermal energy application. The problem is a supercritical water/steam two-phase flow problem for geothermal reservoirs (1 mass balance for two phases and 1 energy balance). We studied an example of the problem `steam_3D_Model_25m` which is a local 3D model with 1,049,600 grid cells. Up to now, no real effective iterative solution for these problems has been identified. The properties of the coefficient matrix are shown in Table 32.

Figure 49 shows strong scalability results with MUMPS using up to 15 nodes. MUMPS scales pretty well on this "steam" test problem. As mentioned before, the test machine offers 24 cores per node and for this experiment, we assigned 2 MPI processes for each node with 12 OpenMP threads for each MPI process. This setting of MPI & OpenMP was decided according to experiments summed up in Table 34 where we show the factorization time for different configurations. As seen in the table, we get the best performance when we have 30 MPI processes with 12 OpenMP threads for 15 distributed nodes.

Table 34: Parallel factorization time on 15 distributed nodes when varying the number of MPI process with OpenMP threads.

# Nodes	MPI per node	# MPI proc.	# OpenMP thr.	Factorization Time
15	24	360	1	80.13
15	12	180	2	55.69
15	6	90	4	39.89
15	2	30	10	26.25
15	2	30	12	26.17

Table 35 reports our experimental finding with and without activating the BLR feature

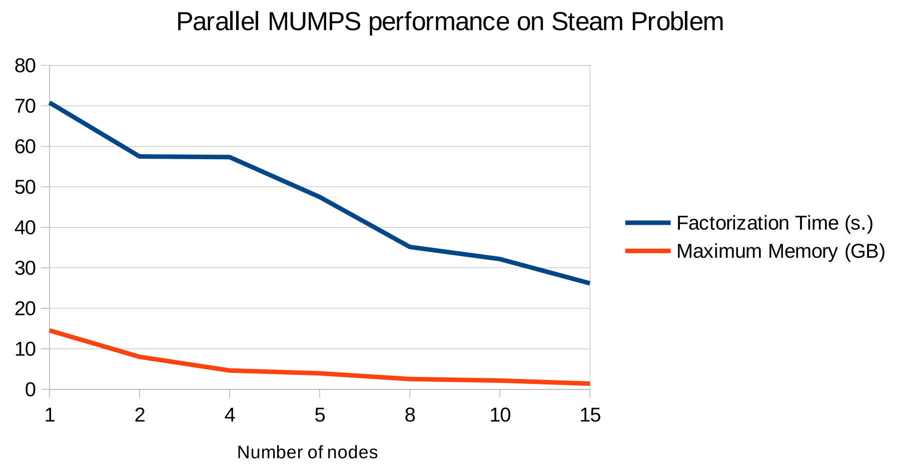


Figure 49: Distributed Memory scalability results for MUMPS. Since each node consists of two sockets, we used 2 MPI with 12 OpenMP threads for the experiments.

of MUMPS. In the table, factorization time is reduced drastically after using very low threshold (for the Block Low Rank level of approximation) while reaching the same relative residual. When we increase the BLR compression threshold from 10^{-16} to 10^{-10} and 10^{-6}, we do not see much improvement in the factorization time while the accuracy in the solution decreases slightly for thresholds above 10^{-16}.

Table 35: Discretization of MUMPS timings for different BLR thresholds. Times are given in seconds. Maximum memory consumption is given in megabytes.

BLR threshold	Analysis t.	Max memory	Factorization t.	Solution t.	Scaled Resi.
NO	18.61	1446	26.17	2.62	1.09×10^{-20}
10^{-16}	19.52	1775	14.14	2.77	1.09×10^{-20}
10^{-10}	19.56	1795	14.61	2.61	2.53×10^{-16}
10^{-06}	19.55	1784	14.04	2.82	6.36×10^{-13}

These various experiments suggest that for the “steam” test problem, a good strategy with MUMPS + BLR could be chosen for an efficient parallel solving method.

Method and Results for ABCD

We also investigated an iterative method for solving the steam problem. The method we used is block Cimmino [1, 2] which is an hybrid linear solver that uses row-block projections to approximate iteratively the solution. We have a distributed memory parallel version of block Cimmino algorithm in ABCD Solver [6] package. The detailed information about this solver can be found in ABCD Section from deliverable 1.7 Software technology improvement.

For the steam system, we have used uniform row-block partitioning for defining the row-blocks, as this gives already good convergence and reduced number of iterations. The number of partitions is an important parameter for performance. Therefore we have tried different number of row-blocks for block Cimmino and we have experienced these in parallel for finding the most efficient partitioning strategy. As a stopping criteria for block

Cimmino, the backward error $(\omega^{(t)})$ at iteration t

$$\omega^{(t)} = \frac{||x||}{||A||\,||x|| + ||b||} \tag{1}$$

is checked and if $\omega^{(t)} < 10^{-16}$ block Cimmino stops. We also compute the relative residual $(\rho^{(t)})$ at iteration t

$$\rho^{(t)} = \frac{||x^{(t)}||}{||b||} \tag{2}$$

Figure 50 shows timings of two most critical phases, namely factorization and iterative solution time, of the block Cimmino together with the sum of timings for these two phases. These experiments have been performed on CERFACS HPC Cluster Nemo with 15 distributed nodes and total of 360 cores $(15 \times 24 = 360)$. More specifically, there has been 180 MPI processes (12 MPI $\times$ 15 nodes) along with 2 OpenMP threads. As seen in the figure, we get the best parallel performance when the number of blocks is equal to 150. Although the factorization time is less when the number of blocks equals 2000, CG solution time is larger due to the higher number of iterations required for convergence.

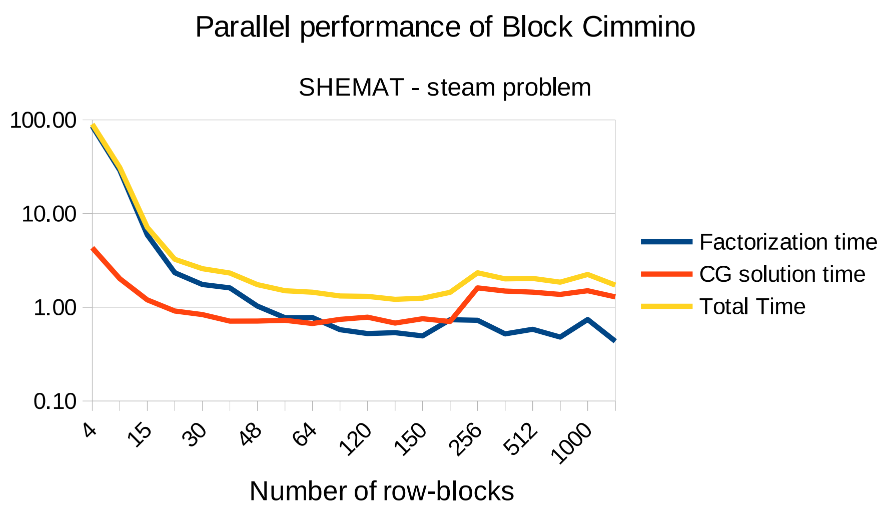


Figure 50: Parallel performance of Block Cimmino with varying number of row-blocks on 15 distibuted nodes with total of 180 MPI processes with 2 OpenMP threads per MPI.

It also important to note that initial $x^{(0)}$ for block Cimmino is given by SHEMAT-Suite and we have started with $\omega^{(0)} = 1.67 \times 10^{-06}$ and $\rho^{(1)} = 1.60 \times 10^{-03}$. We have also compared the computed solution of ABCD against MUMPS's solution. To this end, we compute the forward error (E_f) by considering the solution of MUMPS as the exact solution. Therefore we have found that

$$E_f = \frac{||x - x_{computed}||}{||x||} = 1.2 \times 10^{-3}, \tag{3}$$

which shows Block Cimmino gives comparable solution with MUMPS although Block Cimmino requires much less time for the solution.

Figure 51 shows the effect of increasing the number of partitions in terms of the number of iterations required for the convergence. As expected in row projection methods, when the number of row-blocks increases, convergence rate decreases in general, despite we can observe in this particular test problem that it stays very low even with the large number of partitions.

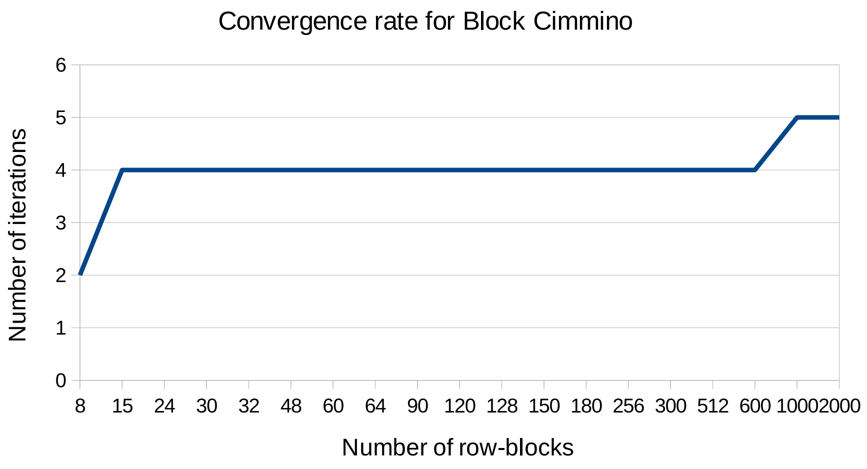


Figure 51: Convergence rate of block Cimmino for steam problem when we increase the number of row-block from 8 to 2000.

Table 36 shows the strong scalability results of block Cimmino. In addition, the last two columns of the table respectively show backward error (ω) and relative residual (ρ) after the final iteration t. The table also shows the maximum memory consumption among parallel processes. When there are more nodes, the maximum memory consumption is reduced.

Table 36: Parallel Block Cimmino results for steam test problem

# Nodes	Factorization Time (s)	CG Solution Time (s)	# CG Iterations	Maximum Memory (MB)	Backward Error	Scaled Residual
1	6.63	9.15	4	303	5.09E-17	8.86E-05
2	1.83	3.08	4	174	2.89E-17	5.47E-05
5	0.89	2.18	4	87	1.86E-17	3.69E-05
5	0.86	2.20	4	87	1.86E-17	3.69E-05
8	0.85	1.93	4	72	1.51E-17	3.03E-05
10	0.78	2.00	4	68	1.50E-17	3.01E-05
15	0.49	1.48	4	35	1.00E-18	2.04E-06

Figure 52 illustrates relative speedup curves for factorization and CG solution phases of Block Cimmino based on the respective values on Table 36. The scalability of the block Cimmino is good, when the number of nodes is increased up to 15. In CERFACS HPC Nemo setting, since the maximum allowed number of nodes is 15 we could not test with larger number of nodes than 15. These latest experiments tend to show that an hybrid solver for the steam problems could also be a method of choice. It remains to check if the quality of the results are compliant with the requirements of the application.

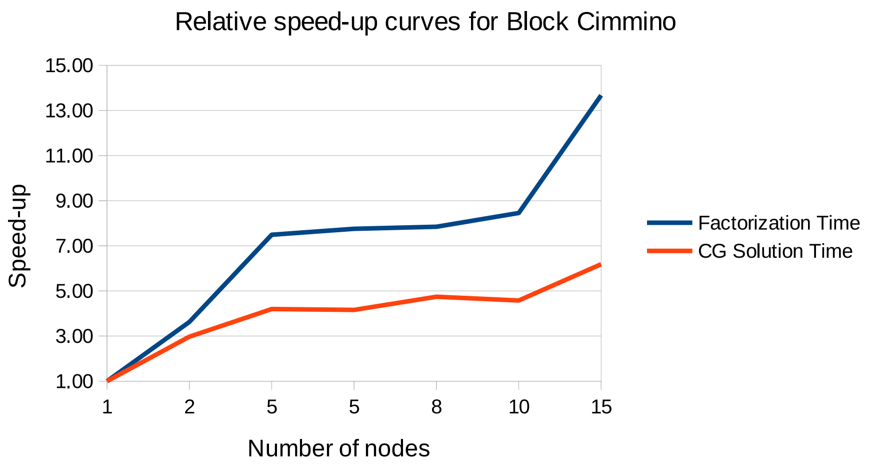


Figure 52: Speed-up curves for strong scalability results of Block Cimmino for the steam problem. On each node we run 12 MPI process with 2 OpenMP threads per MPI process. We fixed the number of row-blocks to 150.

References

[1] Mario Arioli et al. "A block projection method for sparse matrices". In: *SIAM Journal on Scientific and Statistical Computing* 13.1 (1992), pp. 47–70.

[2] LA Drummond et al. "Partitioning strategies for the block Cimmino algorithm". In: *Journal of Engineering Mathematics* 93.1 (2015), pp. 21–39.

[3] Théo Mary. "Block Low-Rank multifrontal solvers: complexity, performance, and scalability". PhD thesis. UT3, 2017.

[4] *Massively Parallel Hybrid Solver (Maphys)*. URL: `https://gitlab.inria.fr/solverstack/maphys`.

[5] *Parallel Sparse direct Solver (PaStiX)*. URL: `https://gitlab.inria.fr/solverstack/pastix`.

[6] Mohamed Zenadi, Daniel Ruiz, and Ronan Guivarch. *THE AUGMENTED BLOCK CIMMINO DISTRIBUTED SOLVER*. `http://abcd.enseeiht.fr/`. ABCD Solver v1.0-beta. 2015.

11. SolarNowcast

Code team:

- Isabelle Herlin (Inria), WP2
- Dominique Béréziat (LIP6), WP2
- Yacine Ould Rouis (MdlS), WP1

Performance metrics

Benchmark characteristics:

Domain size	501*501
Number of time steps	3600
Compile options	-O3 -xHost
Resources	1 node on JURECA
IO details	serial, every 10 timesteps
Type of run	the size of benchmark aims to be faithful to the target use of the program

Table 37: Performance metrics for Nowcasting Forecast module on the JURECA HPC system.

	Metric name	July 2016 (1/8/24 threads)	October 2016 (1/8/24 threads)
Global	Total Time (s)	169 / 34 / 26	76.5 / 11.5 / 4.8
	Time IO (s)	0.4 / 0.5 / 0.4	0.5 / 0.4 / 0.4
	Time MPI (s)	N.A.	N.A.
	Memory vs Compute Bound	N.A.	N.A.
IO	IO Volume (MB)	10529.8	1050.2
	Calls (nb)	80107	80107
	Throughput (MB/s)	2631.1 / 2613.6 / 3159.6	2316.0 / 2576.3 / 2475.2
	Individual IO Access (kB)	1016.4	1016.4
Node	Ratio OpenMP	-0.0	N.A.
	Load Imbalance OpenMP	N.A.	N.A.
	Ratio Synchro / Wait OpenMP	0.0	N.A.
	OpenMP Scalability Efficiency	ref / 62% / 27%	ref / 83% / 66%
Mem	Memory Footprint (KB)	33272 / 34656 / 43696	33780 / 39092/ 38116
	Cache Usage Intensity	N.A.	N.A.
	RAM Avg Throughput (GB/s)	N.A.	N.A.
Core	IPC	N.A.	N.A.
	Runtime without vectorisation (s)	169 / 33 / 27	86.9 / 13.3 / 5.6
	Vectorisation efficiency	1.0 / 1.0 / 1.0	1.13 / 1.16 / 1.16
	Runtime without FMA (s)	163 / 33 / 27	80.2 / 12.1 / 5.1
	FMA efficiency	0.96 / 1.0 / 1.0	1.05 / 1.05 / 1.06

Performance report

SolarNowcast aims to predict the solar irradiation at short term based on data acquired by fisheye lens webcams. Apart from the pre and post processing, the software includes two C/C++ components bound by a bash script:

1. MotionEstimation: It estimates the dynamics from a set of successive acquired images with an iterative minimization of an energy function J with the BFGS solver. J describes the discrepancy between the state vector X and the images. It is computed from the integration of the state vector X by a numerical model assuming the Lagrangian constancy of motion and the transport of the image brightness by motion. The integration is obtained with the second-order semi-Lagrangian scheme SETTLS, Stable Extrapolation Two-Time Level Scheme, solved with a single iteration. ∇J is obtained by the backward integration of the adjoint model, obtained as the result of the differentiation software Tapenade.

2. Forecast: It consists of a simulation of future images, based on the result of MotionEstimation, with a numerical model assuming the Lagrangian constancy of velocity and the transport of image brightness. The integration of the state vector is obtained with the second-order semi-Lagrangian scheme SETTLS, solved with an adaptive number of iterations.

The performance objectives of SolarNowcast are of real time order: to return prediction results for a given period (ex: 1 hour) in the lapse of time between two images acquisitions (10 seconds in the targeted production benchmark).

The following shows the performance analysis of the Forecast component on JURECA (Intel Xeon E5-2680), using Intel compiler. The results/benefits of this analysis and the following optimization work could be confirmed on other hardware, and using GNU compilers.

- Time performance of Forecast: The first time measurements on JURECA showed a slower run with GCC 5.3, performing in 190 seconds, compared to the Intel compile that ran in 169 seconds for a serial run, 26 seconds when exploiting all 24 physical cores on one node.

- Max memory use: between 33 and 43 MB. The whole data can probably fit in the cache.

- IOs: according to Darshan, 0.4 seconds serial. read bandwidth: 600 MB/s, write bandwidth: 2.6 GB/s. The IO time is rather insignificant compared to execution times from 1 to 24 threads. The output frequency is every 10 time steps (we write at 360 steps). The measured time spent in writing operations, including data copying and reorganisation, is approximately 0.8 seconds, which, we will see, will become significant at the end of the optimization process.

- Scalasca: gives bad results. It struggles with tracing small gnu, std, and omp calls. The problem could be solved with some time and communication with Scalasca support group, but the choice was made to report this issue and go on skipping this part, and using other tools, more adequate for single process codes.

- no-vec: The code is not benefitting from vectorization capabilities of the processors.

- no-fma: The code is not benefitting from the fused-multiply-add capabilities of the processors.

- Scalability: The scalability tests consist on 2 successive runs of Motion Estimation (ME) and Forecast, on 2 "windows" of datas. A window consists of a set of successive images (4 in this case) on which ME calculates the motion tendency, that Forcast uses for prediction. The second window is obtained by sliding the previous window by 1 image ahead, the new call of ME, using results of the previous call, makes less calculations, that's why it is much faster. Forecast then makes a similar calculation on the new tendency. MotionEstimation is obviously not multi-threaded. Forecast, on the other hand, uses multi-threading, with weak performances, as it quickly falls under 50% efficiency, over 8 threads [Table 38]. Further investigation (VTune) shows that it is both due to Amdahl's law and inefficient multi-threading.

Table 38: Scalability of Forecast and MotionEstimation at initial state, case size 501*501, JURECA node

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Forecast OMP efficiency
1	43	169	6	168	
2	43	96	6	92	91 %
4	43	59	6	58	71 %
8	45	41	6	40	52 %
12	43	31	6	31	45 %
24	43	26	6	26	27 %

- VTune: VTune gives a very interesting insight into the code, and puts the light on the main OpenMP and serial bottlenecks, thus allows to emit suppositions on some potential improvements:
  - IntegreAll: This routine implements the calculation of a new time step's state from the 2 previous ones, using finite differences and a semi-Lagrangian model. It is the main computational part of the program, and it is therefore the most costly. It also includes the calls to LinInterp. It needs refactoring, especially costly and uselessly redundant divide operations. It is also the only part using OpenMP. The omp pragma is applied on an inside loop, while the outer loop seems more fit for a coarser granularity parallelisation: this would reduce the number of threads splittings and mergings, improve the data locality and reduce concurrent accesses. Some operations in the inner loop could then benefit from SIMD operations.
  - LinInterp: contains a lot of conditional statements, for dealing with the boundary conditions within the semi-Lagrangian method, which are costly, prevent vectorization, and are prone to mistakes (a little one detected on one of the boundaries). Improving it could imply restructuring the X array and adding halo cells, if the method can accept a maximum displacement limit on one time step.
  - Memsets and memcopies become significant in the distributed behavior.

Application support

Activity type	WP1 support
Contributors	Y. Ould-Rouis (WP1), D. Bereziat (WP2), I. Herlin (WP2)

The optimization process was primarily based on measurements performed by 2 JUBE scripts, reproducing a scalability test, on 1 to 24 cores, and a VTune profile collection. Other tools were occasionally used, such as Intel vectorization reports. The "EoCoE JUBE integrated perf evaluation" was used to monitor the evolution, only at notable steps.

We could describe the optimization work as follows:

- Code understanding
- Exploring the major hot-spots through the VTune profile
- Checking the nested loops order, to avoid data access indirections. No notable problems found at this level.
- Checking for redundant operations. When spotted, factorize the operations.
  This translates, for example, in factorizing divisions in IntegreAll loops. In any case, a division is more expensive than a multiplication, so any constant divisor in an iteration should be replaced with a multiplication by its inverse.
  Looking at the bigger picture, the time loop, some costly operations present in the code, were redundant, or even useless. This was the case with "SetAll" calls (memsets) in IntegreAll, on arrays that were entirely rewritten in the next step. These operations were simply removed.
- Vectorization: Based on Intel compiler's opt-reports, the semi-Lagrangian calculation (IntegreAll) innermost loop, among others, was entirely rewritten in a shape easily vectorizable by the compiler, removing any ambiguity regarding instructions inter-dependance, and favouring data alignment when possible.
- OpenMP:
  - The OpenMP distribution was improved by dividing the main IntegreAll loop into bigger independent chunks, reducing the number of threads initializations, and improving the data locality and SIMD possibilities within each thread.
  - Thread binding, using kmp_affinity for Intel, OMP_PROC_BIND on GCC, improves the cache usage and reduces threads initiation time
- Memory Copy Optimization: Saving the state vector at the 2 previous time steps, necessary for the calculation of the next state, was using data copy. The code was adapted in order to use pointer copies. In addition to saving a precious mem_cpy time, this step allowed to unveil and fix an algorithmic bug.
- Porting on GNU compilers: All along the optimization process, the code performed much better on Intel compiler. However, the main bottleneck on GNU was spotted as being the use of the floor function in the projection.
  The standard library floor function is costly, as it performs a series of conditional statements in order to deal with overflow cases. In some cases it even had a problematic behavior, multiplying the execution time 4-fold (-O3 -march native, on

eider, with GCC 4.6.3). In a bound problem like ours, we can replace it with a much lighter, vectorizable function. This modification made a significant difference, bringing the GNU compiled code very close to the Intel compile performance, while not modifying the performance of the latter.
Another necessary modification on GCC was to remove a costly file reading part when not needed. This part reads a list of output times from the input file, using a search routine with a key word for each of these times. It is not needed when determining a constant output interval, thus it could be conditioned out. We strongly recommend to replace this routine with a more performant one, parsing the list of output time steps straight in one step.

- A word on MotionEstimation: After completing this work on Forecast, it was decided that the code owners will handle the optimization of MotionEstimation and try to apply the best practises described above alone, with the WP1 contact only providing performance analysis and advise, as a way of knowledge transmission.

Results

Table 39: Results and scalability on a JURECA node (Two Intel Xeon E5-2680 v3 Haswell CPUs 30 MB cache 2.5 GHz), Intel compiler, with best opt. flags "-O2 -xHost -ipo".

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Forecast improvement	Scaling efficiency
1	38.6	76.9	3.7	77.1	54 %	
2	38.9	39.9	3.7	39.8	58 %	97 %
4	38.4	22.0	4.0	22.0	62 %	88 %
8	38.3	11.5	3.8	11.6	71 %	83 %
12	38.3	8.5	3.8	8.2	73 %	78 %
24	38.1	4.8	3.8	4.8	81 %	67 %

Table 40: Results and scalability on a JURECA node, GNU 5.3 compiler, with best opt. flags "-O3 -march=native".

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Forecast improvement	Scaling efficiency
1	74.209	80.48	9.977	78.594		
2	73.642	41.021	9.988	40.69		97 %
4	73.581	23.322	10.108	23.279		85 %
8	74.228	12.098	10.242	12.052		81 %
12	73.564	8.481	10.282	8.431		78 %
24	73.652	5.1	10.45	5.077		65 %

Table 41: Results and scalability on Eider node (2x Intel Xeon CPU E5-2650 20 MB cache 2.00 GHz), GCC 4.6.3.

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2 (IntegreAll)	Forecast improvement	Scaling efficiency
1	161.305	117.892	11.423	116.500 (114.082)	53 %	
8	152.748	18.295	11.425	17.863 (16.4179)		81 %
16	153.043	10.448	11.421	10.494 (9.07102)		70 %

As described above, a full performance evaluation on the Forecast code allowed to identify a big optimization potential, both on serial and parallel levels. Optimization efforts improved the run time, on the targeted production benchmark, by more than 2 times on the serial run, 4 times on 8 threads, and more than 5 times on 24 threads, and raised the scalability efficiency to 70% on 16 threads and 66% on 24 threads, with both compilers, and on several machines.

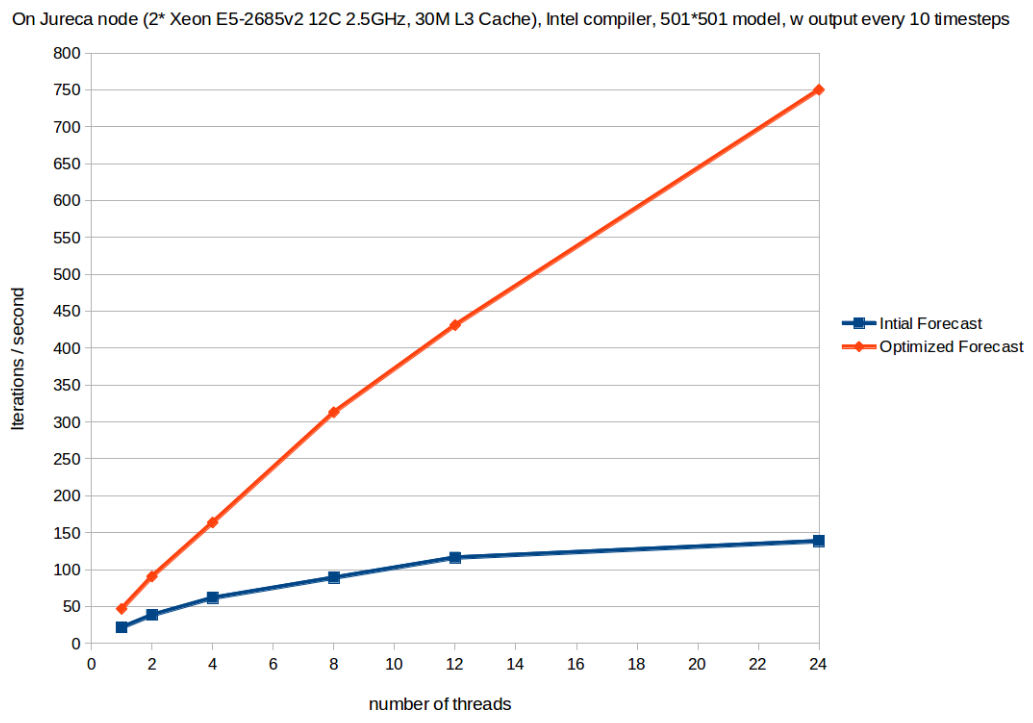


Figure 53: Performance and scalability of Forecast, before and after optimization.

Knowledge transfer

Activity type	WP1 knowledge transfer
Contributors	D. Bereziat (WP2), supported by Y. Ould-Rouis (WP1)

This part of this study addresses the optimization of the Motion Estimation (ME) module and has been done by the code owners following the best practises learned during the optimisation stage of Forecast module.

The optimization process was primarily based on measurements performed by 2 JUBE scripts, reproducing a scalability test, on 1 to 24 cores, and a VTune profiling. The "EoCoE JUBE integrated perf evaluation" was not used for this part.

Motion is computed by minimising a cost function with an steepest descent method. Each iteration has three steps: forward integration, backward integration and call of the solver to perform the steepest descent.

The forward integration is similar to the Forecast module: the model is integrated in time. In addition, cost function and departures are also computed. Cost function value is mandatory by the solver and departures are required for the computation of the cost function gradient. However the complexity of these calculations remains negligible in comparison of the integration of the model. Moreover they are can be easily distributed using Open MP directives. The model optimisation has already been investigated in the Forecast module.

Optimisation of adjoint model code

The backward integration requires to integrate backward in time the adjoint model. Adjoint model code is obtained by the automatic differentiation of the model code. This is performed by Tapenade[9].

In a first step, we reported the optimised model from Forecast module into ME one (the two models are identical). A first run on one thread of Eider node showed a time of 153 seconds against 132 before optimisation of model code (on the first window). A quick investigation with VTune indicated the bad performances were caused by the adjoint code (automatically obtained from Tapenade) and function `IntegreAll_b()`. Indeed, the adjoint code needs to store data in a stack with the inconvenience to break vectorization. After a discussion with Tapenade Authors, it appeared that the stack operations were due to a limitation of Tapenade to identify independent iterations in a loop. However, it is possible to reduce the number of stack operations if we explicitly declare, using a Tapenade directive in the code, independent iterations. After several tries, we inserted two directives of independent iterations with the result to totally eliminate stack operations. A run on one thread of Eider node gave 86 seconds (so an improvement of 36% compared to the non optimised code).

The second step was to study the scalability of ME module with OpenMP. It is important to notice the elimination of Tapenade stack operations allows now to distribute the independent iterations on several threads. The stack operation functions make use global arrays and make impossible to use OpenMP directives. We inserted an OpenMP directive for the main loop of `IntegreAll_b()`, at the same level than in `IntegreAll()`. As showed by Tables 42, 43 and 44, the scaling efficiency is not good for more than 4 threads. We introduced timers for the 3 main stages of the ME module: forward integration, backward integration, and solver call. We report these results in Tables 45, 46 and 47. As it can be seen, scaling efficiency is very good for forward and backward step. The solver remains mono-thread, the global scaling efficiency is not good in conformity with Amdahl's law.

Table 42: Scalability on Eider node (GNU Compiler with option `"-O3"`).

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Scaling efficiency (W1)
1	86.213	118.475	11.025	118.515	
8	37.261	17.899	4.535	17.900	29%
16	33.728	10.446	4.103	10.419	16%

Table 43: Scalability on JURECA node (GNU Compiler with option `"-O3 -march=native"`).

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Scaling efficiency (W1)
1	44.745	79.312	6.268	85.565	
2	29.472	41.197	4.364	41.771	77%
4	22.85	23.229	3.488	23.187	49%
8	18.475	12.345	3.068	12.084	31%
12	17.238	8.643	2.864	8.475	22%
24	16.195	5.136	2.88	4.986	12%

[9] `http://www-sop.inria.fr/tropics/tapenade.html`

Table 44: Scalability on JURECA node (Intel Compiler with option `"-O3 -xHost -ipo"`).

Threads	MotionEst W1	Forecast W1	MotionEst W2	Forecast W2	Scaling efficiency (W1)
1	49.363	68.033	5.499	68.672	
2	33.261	34.864	3.729	34.744	74%
4	26.509	19.605	2.908	19.722	47%
8	21.918	10.441	2.261	10.384	29%
12	20.679	7.765	1.969	7.288	19%
24	19.444	4.656	1.819	4.59	11%

Table 45: Scalability on Eider node (GNU Compiler with option `"-O3"`) and OpenBLAS.

Threads	MotionEst W1			
	Total	Forward	Backward	Solver
1	89.090	24.9864	44.3159	17.6397
4	43.054	8.3472	12.1346	21.2606
8	34.532	5.13174	7.0714	21.0328
12	31.344	4.62574	5.22171	20.136
16	28.864	4.55259	5.04882	17.8837

Optimisation of the solver

The last step is to work on solver scalability. We use L-BFGS solver version 3.0, written in Fortran. Using Vtune, we have identified 4 costly loops in the main part of the solver. Iterations inside theses loops being independent, we have introduced Open MP directives to distribute them among available cores. Scaling performances are presented in Table 48 and 49. As it can been seen, performance and scalability of Motion Estimation have been improved, but efficiency remains poor.

The solver makes call of BLAS functions `daxpy`, `dcopy` and `ddot`. We tried to link with OpenBLAS (a multithreaded implementation of BLAS), without notice some improvement (see tables 45, 46 and 47). We are not sure OpenBLAS calls are effective, this is currently under investigation.

Another way is to change the solver. For instance, HLBFGS [10] is an C++ implementation of L-BFGS with OpenMP directives that could be interesting. The objective to obtain a time computation close to 5 seconds for the ME stage appears realist if we obtain a good scalability on solver.

[10]`http://research.microsoft.com/en-us/UM/people/yangliu/software/HLBFGS/`

Table 46: Scalability on JURECA node (GNU Compiler with option `"-O3 -march=native"`) and OpenBLAS.

Threads	MotionEst W1				Scaling efficiency	
	Total	Forward	Backward	Solver	Total	Forward+Backward
1	44.862	12.7	21.8	9.1		
2	28.03	6.7	11.3	9.0	80%	96%
4	20.686	4	6.7	9.1	53%	81%
8	16.083	2.7	3.2	9.2	35%	73%
12	14.457	2.1	2.2	9.2	27%	67%
24	13.438	1.4	1.6	9.4	14%	48%

Table 47: Scalability on JURECA node (Intel Compiler with option `"-O3 -xHost -ipo"`) and OpenBLAS.

Threads	MotionEst W1				Scaling efficiency	
	Total	Forward	Backward	Solver	Total	Forward+Backward
1	49.889	15.8	18.1	14.4		
2	32.023	8.1	9.5	13.1	78%	96%
4	24.445	4.6	5.3	13.5	52%	86%
8	19.029	2.6	2.8	12.7	33%	78%
12	17.798	2.0	2.0	12.7	23%	70%
24	16.284	1.3	1.4	12.6	13%	52%

Table 48: Scalability on Eider node (best optimiser flags), after optimisation of LBFGS.

Threads	MotionEst W1				Scaling efficiency
	Total	Forward	Backward	Solver	Total
1	80.2	20.8	39.5	17.8	-
4	27.4	6.4	10.2	9.6	73%
8	21.4	4.4	7.1	8.6	46%
12	19.9	3.7	5.1	9.9	33.6%
16	21.4	3.3	4.2	9.6	23.4%

Table 49: Scalability on JURECA node with Intel compiler (best optimiser flags), after optimisation of LBFGS.

Threads	MotionEst W1				Scaling efficiency
	Total	Forward	Backward	Solver	Total
1	63.8	17.4	25.0	19.3	
2	35.7	8.4	12.3	13.7	89%
4	25.0	4.8	6.9	12.1	64%
8	17.8	2.8	3.6	10.5	44%
12	15.9	2.16833	2.5	10.2	33%
24	13.9	1.5	1.8	9.7	19%

12. Telemac

Overview

TELEMAC-MASCARET is an integrated suite of solvers for use in the field of free-surface flow.

The application submitted to EoCoE performance evaluation is a 3D multi-physics coupling for coastal simulation. It consists of a 2D free surface fluid dynamics model coupled with wave propagation and sediment transport. The code is pure MPI.

Code team:

- Yacine Ould-Rouis (MdlS) for WP1
- Antoine Joly (EDF) for external partners

Performance metrics

Case 1 characteristics:

Resources	2 nodes on JURECA (48 cores)
Domain size	Non structured multi-layer mesh (about 1800 points per process, 36 plans, 25 frequencies)
IO details	
Type of run	development run / production run

Application support

Activity type	WP1 support
Contributors	Y. OULD ROUIS (WP1)

The application support on Telemac focuses on the memory-bound behavior of the code, and finding a way to unlock the vectorization potential on the core level when possible. The work has been conducted based on the conclusions of the performance evaluation.

In the following, I describe point-by-point the different actions and steps taken in this work, and expose the results at the end.

First, it is important to note that the simple compiler upgrade from Intel 2017 to Intel 2018 changed the behavior of the code, improving the execution time by 10%, while reducing some MPI latencies and slightly modifying the execution profile as shown in Figure 55. This execution time will be taken as a reference for the following improvements.

QNLIN1 is the costliest routine (28% of the execution time). The main hot-spot here is made of 3 nested loops enclosing a series of 6 loops on the same dimension, separated by conditional (but non exclusive) statements. These inner loops, while very well vectorizable, had a bad cache usage that made them run at almost scalar speed, as the data had to be reloaded in the cache at each successive loop. Moreover, The calculations involve memory accesses of neighbours of each node on the different dimensions. Therefore the cache blocking appeared to be a good technique to try in order to improve the data locality.

Table 50: Performance metrics for telemac2d-tomawac-sysphe coupling on the JURECA HPC system.

	Metric name	original	after IO fix	after App Support
Global	Total Time (s)	857	701	519
	Time IO (s)	107	1.99	4.2
	Time MPI (s)	178	173	105
	Memory vs Compute Bound	1.26	1.44	1.36
	Load Imbalance (%)	20	23	19
IO	IO Volume (MB)	9972	9932	9912
	Calls (nb)	584 M	422 K	422 K
	Throughput (MB/s)	93	4997	2361
	Individual IO Access (kB)	0.02	74.15	74.32
MPI	P2P Calls (nb)	1390838	1390838	1390996
	P2P Calls (s)	14	13	7
	P2P Calls Message Size (kB)	1	1	1
	Collective Calls (nb)	306873	306873	306843
	Collective Calls (s)	163	159	97
	Coll. Calls Message Size (kB)	59	59	59
	Synchro / Wait MPI (s)	148	143	86
	Ratio Synchro / Wait MPI (%)	82.73	82.20	80.21
Mem	Memory Footprint	369MB	365MB	375MB
	L3 Cache Usage Intensity (%)	78	78	73
Core	IPC	0.97	0.96	0.94
	Runtime without vectorisation (s)	850	727	582
	Vectorisation efficiency	1	1.04	1.12
	Runtime without FMA (s)	815	702	584
	FMA efficiency	0.95	1.00	1.13

Cache blocking or **loop blocking** is the transformation of the memory domain of a given problem into smaller chunks, rather than sequentially traversing through the entire memory domain. Each chunk should be small enough to fit all the data for a given computation into the cache, thereby maximizing data reuse.

The first tries were successful beyond expectation, resulting in QNLIN1 running 4 to 5 times faster, saving 15% of the total execution time on full nodes. With a tuning benchmark displayed in Figure 54, this gain was maximized to 18%, reducing the execution time from 624 to 519 seconds, with a block size of 16 to 64 iterations.

The new time distribution in the code is as displayed in the new profiling in Figure 56.

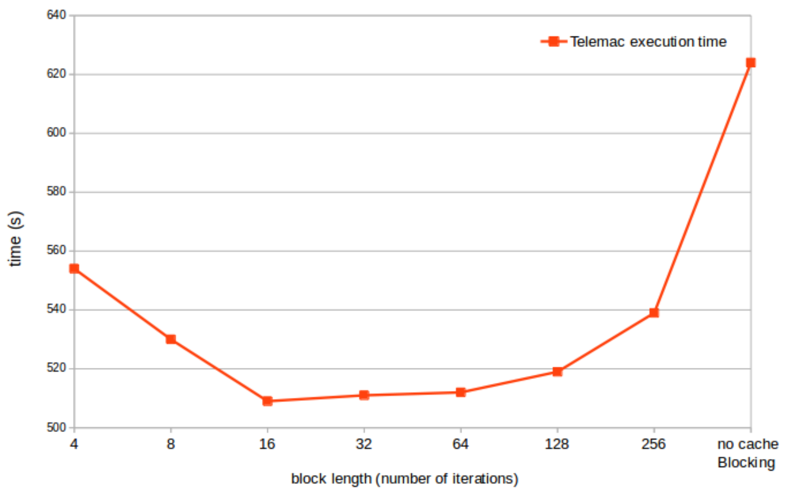


Figure 54: Result of the cache blocking introduced in QNLIN1 on the total run time, benchmark for different block sizes.

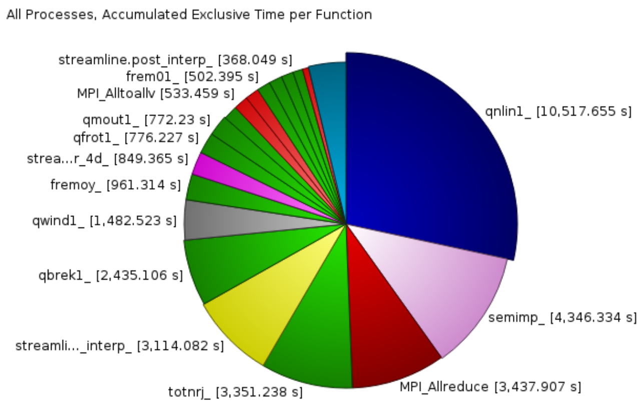


Figure 55: Telemac - Initial time profiling using Scalasca and Vampir visualization - Compiler: Intel 2018.

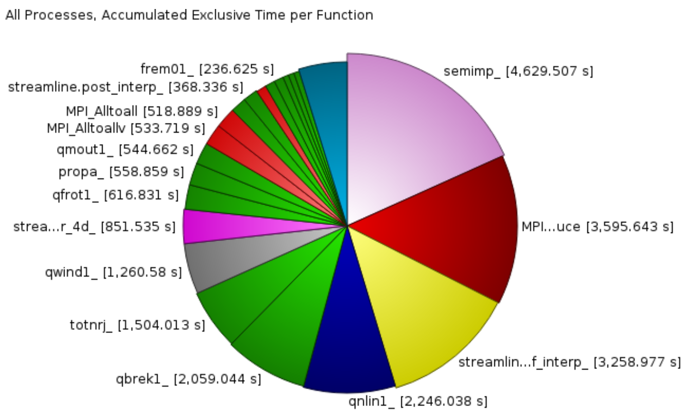


Figure 56: Telemac - time profiling after QNLIN1 optimization - Compiler: Intel 2018.

13. Tokam3X

Code team:

- Patrick Tamain (IRFM, CEA CADARACHE) for WP5
- Guillaume Latu (IRFM, CEA CADARACHE) for WP5
- Luc Giraud (Inria) for WP1
- Vineet Soni (Maison de la Simulation, CEA Saclay) for WP1
- Mathieu Lobet (Maison de la Simulation, CEA Saclay) for WP1

Architectures:

Tokam3X optimizations have been tested on three different Intel CPU archtectures:

- Intel Skylake: The Skylake partition of the Irene Joliot-Curie cluster at TGCC has been used. Each node has 2x24-cores Intel Skylake 8168 at 2.7GHz.
- Intel Knight Landing (KNL): Runs opn KNL have been done on the cluster Frioul at CINES. Each node of this cluster is equipped of a single Intel KNL 7250 @1.4 Ghz configured in quandrant cache mode.
- Intel Haswell: The Jureca machine at JSC has been used for testing this architecture. Each node has 2x12-cores Intel Haswell .

Case characteristics:

Three test cases have been prepared to investigate the performance issues of Tokam3X:

Small case	
Domain size	$N_r = 32$ x $N_\theta = 128$ x $N_\varphi = 16$, 30 steps
Resources	1 node on Jureca (24 cores)
IO details	Checkpoint written every 30 steps
Run description	A small case that can be run on a single node on a very short time. This test is perfect to test the profiling and tracing tools
Input name	PARAM_LIMITER_SMALL.TXT

Medium case	
Domain size	$N_r = 64$ x $N_\theta = 256$ x $N_\varphi = 32$, 30 steps
Resources	1 or 2 nodes on Jureca (24 cores)
IO details	Checkpoint written every 30 steps
Run description	A medium-size case that can be run on 1 or 2 nodes. Perfect for deeper and more realistic metrics without the load of production run.
Input name	PARAM_LIMITER_MEDIUM2.TXT

Large case	
Domain size	$N_r = 64$ x $N_\theta = 512$ x $N_\varphi = 32$
Resources	
IO details	
Run description	A large case closer to the production run.
Input name	PARAM_LIMITER_LARGE2.TXT

Benchmark code characteristics:

Tokam3X has been analyzed with Jube and associated performance tools. The performance results for the original version of Tokam3X (state of the code at the beginning of the project) is shown in comparison with the optimized one in Table 51. They have been obtained using the medium case. In the production runs, Tokam3X uses a multiple thread MPI communication pattern (MPI_THREAD_MULTIPLE). Unfortunately, this option is non-compatible with some performance analysis tools. The results shown in this report have therefore been obtained with a funneled thread version (MPI_THREAD_FUNNELED). The code has also been analyzed with Allinea perf-report, Allinea MAP, Intel Vtune Amplifier, Intel Advisor and Intel Trace Analyzer and Collector. For some Intel tools, only small and medium cases have been used with a relatively small number of iterations. For some unknown reason, the analysis gets stuck at a given iteration when the domain is too large without any error message. This may be due to the high number of communicators generated in Tokam3X.

Furthermore, information given by Jube may be partially wrong, because TOKAM3X (parallelised with OpenMP) uses the library PastiX parallelized with Pthreads. Pthreads is not considered during the tool analyses.

Performance metrics

Global performance:

Although table 51 indicates that the code is strongly compute bound. As mentioned previously, it might not be reliable, as it is not possible to instrument OpenMP and Pthreads simultaneously using some tools. In table 56, it is found that the code is around 40% memory bound for the medium case. It is important to note that this data represents the performance of the code outside of PastiX library. Results in the table is obtained from the Intel Vtune Amplifier.

For the large case, Allinea announces an average memory usage of 21.5 GiB (peak of 24.7 GiB) per node. The node memory usage is of 27 % which is low. subject to compute/cache optimizations, this number could be increased at no cost. The metric Load imbalance in Tab. 51 shows that the code could be 90 % faster (almost speedup of 2) with a perfect load balance. However, this conclusion may be wrong due to Pthreads in Pastix.

In tables 52,53 and 54 comparisons of the optimized version is given with the original version of the code on Haswell, Skylake and KNL computing architectures for small, medium and large cases, respectively. The results indicate the best speedup on the KNL architecture, but as expected, it is also the slowest among all. Interestingly, the GCC compiler shows better performance compared to the Intel one for small case. However, it is important to note that unlike GCC compiler, the elapsed-time found using the Intel compilers fluctuates significantly. Also, as seen in the tables, the computation time is largely dominated by the time in PastiX library. Since performance optimization of this library is out of scope for the current work, time spent in it is not considered in the analyses presented in this report.

MPI performance:

The MPI communications' metric in table 51 indicates that it constitutes a small fraction of the simulation time ($< 1\%$). However, as noted previously, it does not take into account the PastiX library. Moreover, it is obtained with the MPI_THREAD_FUNNELED mode. Table 55 shows the correct MPI performance results for MPI_THREAD_FUNNELED and MPI_THREAD_MULTIPLE configurations. As it can be seen, although MPI communications take less time in MPI_THREAD_FUNNELED, it is slower. While MPI_THREAD_MULTIPLE mode is faster and used in the production runs, the MPI calls comprise a significant part of the elapsed time. Also, all performance analyses in this report use MPI_THREAD_MULTIPLE mode, unless otherwise stated.

In an interesting test, the performance of the code for small and medium cases is obtained by varying the combination of number of MPI tasks and OpenMP threads in a single node of Haswell machine. The results are presented in Figure 57. As shown, neither pure-MPI nor pure-OpenMP has satisfactory performance. Additionally, the time spent in MPI communications increases with the number of MPI tasks as expected.

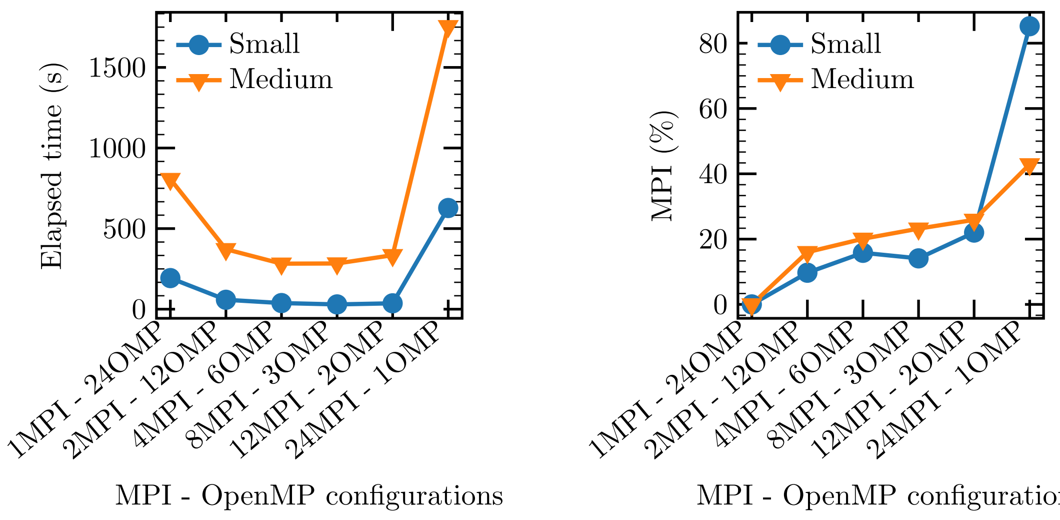


Figure 57: Elapsed time (left) and time taken in MPI calls (right) for small and medium cases using different MPI tasks and OpenMP threads in a Haswell node for 100 time-steps. Here, the elapsed time represents the total time including PastiX. MPI performance obtained from the Intel Trace Analyzer and Collector.

OpenMP performance:

OpenMP represents a small fraction of the overall time, around 13 % according to the Jube report. The thread parallelisation fraction may be artificially low because Pthreads used in Pastix is not taken into account. However, table 56 represents the performance of the OpenMP for small and medium cases found using Intel Vtune Amplifier. It is important to note that this analysis is obtained with a different OpenMP-MPI combination than the other analyses presented in this report; and is based on the code outside of PastiX. Hence, there is a little incoherence in the data given in the table compared to the data in other tables. As shown in the table, the OpenMP region comprises of 85-96% of the computation depending on the test case.

A big number of OpenMP regions have been merged; as a result, the number of OpenMP fork-join per iteration of the main computation loop has been significantly reduced. For small and medium cases, this reduction is noted to be 57 and 89 in the optimized version as opposed to 567 and 871 for the original version of the code.

Vectorization performance:

According to the Jube report (see table 51), the code is not vectorised and does not benefit from the Fuse Multiply Add instructions (FMA). However, since it includes the PastiX library, it may be not be correct as mentioned previously.

In order to enable vectorisation, the data structure of the entire code has been modified. Consequently, a large number of loops are now vectorised. The speedup gained as a result of vectorisation on the Haswell and KNL architectures is given in table 57.

IOs:

The IO volume is small and the time spent in IOs seems small in comparison to the simulation time (0.2 % according to Allinea, much less according to Darshan).

Nonetheless, hdf5 output files are now written serially. If the volume of data stays small, parallelisation of the IOs is not a priority, but should be considered anyway.

Conclusion:

Vectorisation is enabled in most of the loops by modifying the data-structure of the code. Except for the regions where the external library, PastiX, is used, almost all of the code has been merged under a single OpenMP region.

The major bottleneck of Tokam3X is the use of external library. It comprises of 77% and 98% of the computation time for the small and large case, respectively. Tokam3X efficiency thus depends on the future optimization in this library, in particular, for the Many-Integrated Core Architectures.

Application support

Activity type	Consultancy or WP1 support
Contributors	P. Tamain (WP5), G. Latu (WP5), L. Giraud (WP1), M. Lobet (WP1), V. Soni (WP1)

Since the start of the EoCoE project, two axes of development have been explored to improve the performances of the TOKAM3X tokamak edge plasma turbulence code. On the one hand, exhaustive studies have been carried out in sight of solving the main numerical bottleneck of the existing production version of TOKAM3X, ie, the 3D implicit solver for the vorticity operator; on the other hand, preliminary studies have started and produced first results concerning a possible porting of the whole numerical scheme towards High order Discontinuous Galerkin (HDG) methods. Both are expected to contribute to getting the code closer to being able to tackle full scale ITER simulations. Let us now detail each of these axes. Concerning the 3D implicit solver for vorticity operator, studies have focused on the possibility of replacing the current direct solver by a more scalable iterative method. A systematic study of preconditioner – iterative scheme options has been conducted (Fig. 58) and led to the identification of a promising preconditioner – iterative solver couple (the so-called "P9" preconditioner associated with a GMRES solver) showing a good balance of convergence rate and memory consumption. In collaboration with INRIA Bordeaux, the PASTIX solver used in the TOKAM3X code has been modified to include an implementation of a parallel GMRES solver with flexible pre-conditioner. Tests in large scale simulations are planned for the beginning of 2017. Note also that this study highlighted that the condition number of the matrix to invert could get extremely large (1014 or more) in some conditions of physical interest. In such cases, given the numerical precision of floating point operations, the solution found by iterative or even direct solvers can be very different from the analytical solution of the problem. To circumvent this issue, we have modified the equations of the physical model by adding electron inertia which was neglected before. This modification prevents the condition number of the matrix from diverging (gain of 103 or more on the condition number), making the system easier to solve with iterative methods and the solution more precise.

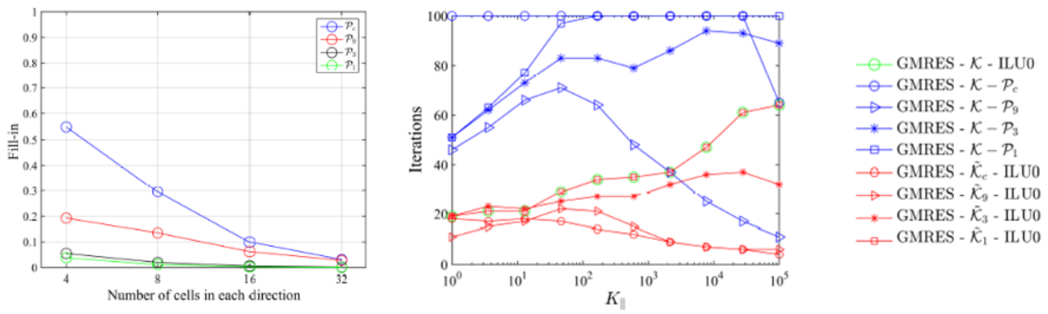


Figure 58: Left: memory consumption of the LU factorization of 4 different preconditioners relative to the memory consumption of the LU factorization of the total matrix. Right: number of iterations before convergence (100 = not converging) as a function of the parallel conductivity for different preconditioner-solver couples. Discretization mesh = 32x32x32, tolerance for convergence = 10-4 and 100 iterations maximum.

In parallel, studies have started on the usage of HDG methods. Adopting an HDG scheme implies a complete change of the numerical scheme of the code with the promise of considerably increasing its geometrical flexibility and allowing high-order spatial discretization. Preliminary studies have been carried out on a reduced version of the TOKAM3X model, including solely the parallel dynamics of the plasma in an isothermal assumption. Fig. 59 shows an application of the HDG version of the code to a simulation in the WEST geometry. The HDG scheme allows one to model precisely the wall's geometry among which the fine baffle structures at the bottom of the machine. It also makes possible the extension of the simulation domain to the center of the machine, which is particularly difficult with structured mesh algorithms and resolves questions about the boundary conditions one should use at the inner boundary of the domain. This work has been carried out in collaboration with the M2P2 laboratory (CNRS, Aix-Marseille University). Future work will focus on the implementation of heat transport equations whose properties (stiffness, anisotropy) will be a decisive test for the potentiality of HDG schemes to treat the whole TOKAM3X model in sight of ITER cases.

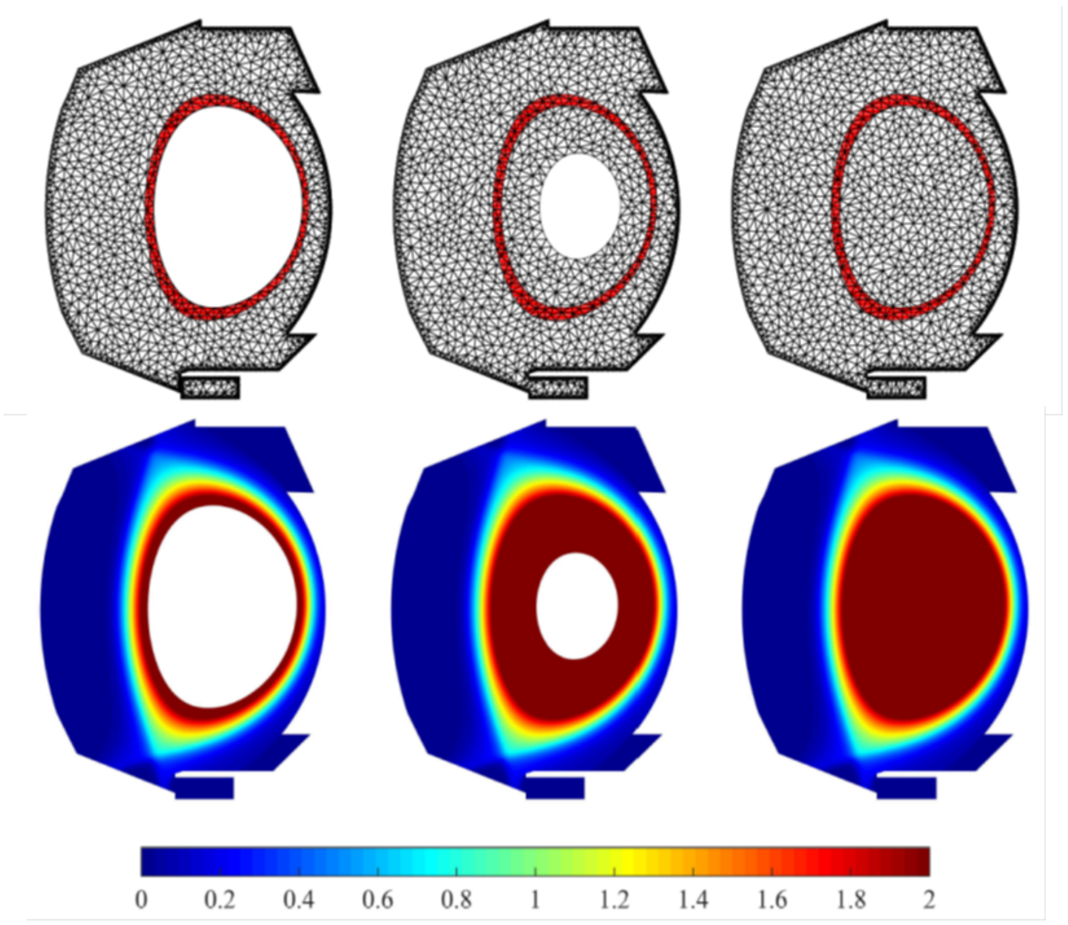


Figure 59: Top: mesh used in 3 different HDG simulations with fixed particle source location (the red-shaded area) extending the domain progressively towards the center of the machine. Bottom: corresponding equilibria obtained on the electron density field.

Table 51: Performance metrics from JUBE for Tokam3X on the JURECA HPC system performed for optimized and original version of the code. The metrics have been obtained with the medium-size case for 30 time-steps on 2 nodes. The results take into account the performance of PastiX library.

	Metric name	Optimized	Original
Global	Total Time (s)	214	226
	Time IO (s)	0.05	0.05
	Time MPI (s)	1.33	1.31
	Memory vs Compute Bound	1.05	1.04
	Load Imbalance (%)	89.31	86.87
IO	IO Volume (MB)	59.98	60.01
	Calls (nb)	8501	8560
	Throughput (MB/s)	1131.40	1193.38
	Individual IO Access (kB)	3.81	3.80
MPI	P2P Calls (nb)	1735	1717
	P2P Calls (s)	0.49	0.47
	P2P Calls Message Size (kB)	14	14
	Collective Calls (nb)	2006	2006
	Collective Calls (s)	0.55	0.56
	Coll. Calls Message Size (kB)	11	11
	Synchro / Wait MPI (s)	0.55	0.53
	Ratio Synchro / Wait MPI (%)	2.94	2.82
Node	Time OpenMP (s)	29.81	31.92
	Ratio OpenMP (%)	13.23	13.66
	Synchro / Wait OpenMP (s)	4.97	0.9
	Ratio Synchro / Wait OpenMP (%)	38.70	6.01
Mem	Memory Footprint	N.A.	N.A.
	Cache Usage Intensity	0.89	0.92
Core	IPC	0.69	0.66
	Runtime without vectorisation (s)	217	229
	Vectorisation efficiency	1.01	1.01
	Runtime without FMA (s)	227	232
	FMA efficiency	1.06	1.03

Table 52: Comparison of the computational time between the optimized and the original branch of the code for 100 time-steps on Haswell, Knights Landing and Skylake using the small case. The total computational time, the time spent in PastiX and the time outside PastiX are given.

Architecture	Time	Small case		
		Optimized	Original	Speed-up
Haswell	Total	52s	60s	
Intel 17 + IntelMPI	In PastiX	40s	42s	
1 node, 2 MPI, 12 OMP	Outside Pastix	12s	18.5s	1.5
Haswell	Total	45s	52s	
GCC 7.3	In PastiX	38s	39s	
1 node, 2 MPI, 12 OMP	Outside Pastix	7s	13s	1.9
KNL	Total	192s	247s	
Intel 18 + IntelMPI	In PastiX	165s	166s	
1 node, 4 MPI, 16 OMP	Outside Pastix	27s	81s	3
Skylake	Total	65s	71s	
Intel 18 + IntelMPI	In PastiX	53s	52s	
1 node, 2 MPI, 24 OMP	Outside Pastix	12s	18s	1.5

Table 53: Comparison of the computational time between the optimized and the original branch of the code for 100 time-steps on Haswell, Knights Landing and Skylake using the medium case. The total computational time, the time spent in PastiX and the time outside PastiX are given.

Architecture	Time	Medium case		
		Optimized	Original	Speed-up
Haswell	Total	426s	450s	
Intel 17 + IntelMPI	In PastiX	388s	392s	
2 node, 2 MPI, 12 OMP	Outside Pastix	38s	58s	1.5
KNL	Total	1007s	1168s	
Intel 18 + IntelMPI	In PastiX	926s	956s	
2 node, 4 MPI, 16 OMP	Outside Pastix	81s	212s	2.6
Skylake	Total	337s	384s	
Intel 18 + IntelMPI	In PastiX	304s	311s	
1 node, 2 MPI, 24 OMP	Outside Pastix	33s	73s	2.2

Table 54: Comparison of the computational time between the optimized and the original branch of the code for 30 time-steps on Haswell, Knights Landing and Skylake using the large case. The total computational time, the time spent in PastiX and the time outside PastiX are given.

Architecture	Time	Large case		
		Optimized	Original	Speed-up
Haswell	Total	1489s	1542s	
Intel 17 + IntelMPI	In PastiX	1458s	1496s	
4 node, 2 MPI, 12 OMP	Outside Pastix	31s	46s	1.5
KNL	Total	5249s	5252s	
Intel 18 + IntelMPI	In PastiX	5154s	5068s	
4 node, 4 MPI, 16 OMP	Outside Pastix	94s	183s	1.9
Skylake	Total	538s	565s	
Intel 18 + IntelMPI	In PastiX	506s	511s	
2 node, 2 MPI, 24 OMP	Outside Pastix	33s	54s	1.7

Table 55: MPI performance using small, medium and large cases for 30 time-steps on the Haswell architecture for MPI_THREAD_FUNNELED and MPI_THREAD_MULTIPLE modes. In this table, the elapsed time includes the time spent in PastiX. Data obtained from the Intel Trace Analyzer and Collector.

		Small (1node, 2MPI, 12OMP)	Medium (2node, 2MPI, 12OMP)	Large (4node, 2MPI, 12OMP)
MPI_THREAD_FUNNELED	Elapsed time (s)	16.8	242.9	2041.8
	MPI (%)	6.7	7.9	6.8
	Collective calls (%)	5.4	4.3	2.8
MPI_THREAD_MULTIPLE	Elapsed time (s)	22.6	215.2	1529.5
	MPI (%)	17.0	55.4	53.6
	Collective calls (%)	15.0	1.2	0.4

Table 56: OpenMP and other performance analyses using small and medium cases for 30 time-steps on the Haswell chips. In this table, the data corresponds to the performance outside of PastiX. Data obtained from the Intel Vtune Amplifier.

	Small 1 node, 2 MPI, 4 OMP	Medium 1 node, 2 MPI, 4 OMP
Total time (s)	3.5	14.9
OpenMP region (%)	84.9	96.2
OpenMP load-imbalance (%)	15.8	15.6
Memory bound (%)	37.1	39.9
Cycles per instructions	0.66	0.66

Table 57: Performance gain solely based on vectorisation using small, medium and large cases on Haswell and KNL architectures.

		Time	Vectorisation efficiency		
			Vectorised	Non-vectorised	Speed-up
Haswell	**Small**	Total	52s	58s	
	NSteps = 100	In PastiX	40s	43s	
	1 node, 2 MPI, 12 OMP	Outside Pastix	12s	16s	1.3
	Medium	Total	426s	428s	
	NSteps = 100	In PastiX	388s	386s	
	2 node, 2 MPI, 12 OMP	Outside Pastix	38s	42s	1.1
	Large	Total	1489s	1860s	
	NSteps = 30	In PastiX	1458s	1823s	
	4 node, 2 MPI, 12 OMP	Outside Pastix	31s	37s	1.2
KNL	**Small**	Total	192s	203s	
	NSteps = 100	In PastiX	165s	170s	
	1 node, 4 MPI, 16 OMP	Outside Pastix	27s	32s	1.2
	Medium	Total	1007s	1023s	
	NSteps = 100	In PastiX	926s	924s	
	2 node, 4 MPI, 16 OMP	Outside Pastix	81s	98s	1.2