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

D1.18 - M24
Application Performance Evaluation

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:	D1.18 - M24
	Lead Beneficiary:	CEA
	Contact:	Matthieu Haefele
	Contact's e-mail:	matthieu.haefele@maisondelasimulation.fr
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M24 31/09/2017
	Actual Date of Delivery:	M26 13/11/2017
	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 Performance Evaluation
	ID :	D1.18 - M24
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	M. Haefele (MdlS), P. Gibbon (JSC), Lührs (JSC), Rohe (JSC)
	Contributors:	U. Aeberhard (FZJ), J. Berndt (FZJ), D. Brömmel (FZJ), G. Houzeaux (BSC), S. Kollet (FZJ), G. Latu (CEA), E. Di Napoli (JSC), Y. Ould-Rouis (MdlS), M. Salanne (MdlS), W. Sharples (FZJ), I. Herlin (INRIA), D. Béréziat, M. Gusso (ENEA), M. Levesque (MdlS), C. Gageat (MdlS), A. Joly (EDF), P. Börner (FZJ), T. Breuer (JSC), F. Xing (BRGM), H. Owen (BSC), G. Houzeaux (BSC), T. Feher (MPG), J. Bruckmann (RWTH), A. Marin-Laflèche (MdlS), P. Tamain (CEA), L. Giraud (INRIA), M. Lobet (MdlS), J. Denev (KIT), T. Zirwes (SCC), P. Leleux (IRIT), M. Kuhn (INRIA), G. Marait (INRIA), R. Halver (JSC), J. Castagna (STFC), M. Kern (INRIA), S. Lopez (BRGM), C. Demetroullas (CyI), A. Funel (ENEA), G. Guarnieri (ENEA), X. Lacoste (Total)
	Reviewed by:	Haefele (MdlS), Gibbon (JSC)

Contents

List of Figures

List of Tables

1. Document release note

This document replaces D1.17 Application Performance Evaluation that has been delivered in M18. For readers already familiar with the previous document, the major additional contributions within this document with respect to the previous one can be found in the following sections:

- Section 4 has been revised. The automated performance evaluation process has been improved with additional features.
- Section 5 provides the updated table for all 19 codes evaluated to date.
- Section A now contains performance reports for an additional 14 codes which were subjected the improved performance evaluation process.

In a nutshell, the three joint EoCoE/POP workshops triggered further cooperation between EoCoE code teams and POP. This corroborated the collaboration and mutual exchange between these two COEs, including participation of EoCoE at the POP workshop at the HPC Summit Week in Barcelona on May 19th 2017, and POP participation at the HPC for Energy workshop organised by EoCoE in Brussels on 15 June 2017.

2. Motivation

As documented in the original proposal, the Energy oriented Center of Excellence (EoCoE) is a user driven consortium dedicated to tackling modelling challenges in the field of renewable energy. Consequently, the implementation and organization of the project places High Performance Computing (HPC) applications of the four chosen user communities at the heart of the project. Within in its transversal basis (WP1), the EoCoE project has gathered a comprehensive range of HPC expertise that aims to enhance the performance of these applications, thereby enabling them to effectively exploit the existing European computing infrastructure. Close interaction between WP1 and the application domains WP2-WP5 is a key feature of EoCoE, with the ultimate goal of expediting advances in simulations of low-carbon energy systems and technology.

In this context, application performance evaluation is an instrument of key importance, since it permits us to:

1. define the status of an application code at the moment when EoCoE HPC experts start to examine it,
2. monitor the impact of each code modification during the optimization process,
3. quantitatively assess the impact of such support activity when it comes to an end.

This deliverable report describes the status of performance evaluation activity over the first 18 months of the project, beginning with a dedicated workshop for this purpose, and various follow-up actions such as Section 4, which presents the definition of the EoCoE performance evaluation report and the performance metrics it uses; Subsection 4.3, on the establishment of an automated and reproduceable process that delivers all the required metrics; Section 5, which describes the system for monitoring progress in application optimisation.

3. Joint EoCoE-PoP benchmarking workshops

3.1 December 2015 in Juelich @ JSC

The first EoCoE-POP workshop on benchmarking and performance analysis brought together code developers of community codes associated with WP 2-5 with HPC experts associated with WP 1 and HPC experts from the CoE "POP". The goal of this 4-day event held at Jülich Supercomputing Centre from 8th-11th December, 2015 was to familiarise the developers from WP2-5 with state-of-the-art HPC performance analysis tools, enabling the teams to make a preliminary identification of bottlenecks, and to initiate the standardisation of benchmark procedures for these codes within the EoCoE project. The workshop comprised 4.5 hours of presentations on the benchmarking and performance tools followed by 12 hours of hands-on work supervised by the WP1 and PoP HPC experts.

Figure 1: Workshop participants and support activity during the first benchmarking workshop

As an initial step, all code developers were instructed on how to perform benchmarking within the JUBE[1] workflow environment, which will permit measurements to be documented, shared and rigorously reproduced over the project lifetime and beyond. Developers were then able to begin analysing their applications using specific HPC tools under the guidance of HPC experts (Score-P, Scalasca, Vampir, Paraver, Extrae, Darshan, VTune and others). Based on this face-to-face collaboration and common training, small teams of code developers and HPC experts from WP 1 were established, who have begun to follow up on the promising initial work to provide comprehensive benchmarks and performance data by the time the next workshop is held in June.

Each of the participating developer teams was allocated a WP1 mentor, tasked with assisting any follow-up benchmarking and tuning work, and acting as an initial contact point for enquiries going beyond the initial assessment (I/O issues, data management, visualisation etc). A summary of the participating codes is given in table 1. Four of these (ALYA, Metallwalls, PARFLOW and Gysela) belong to the set of codes already prioritised (triggered) for WP1 optimisation activity.

A further valuable outcome was the exchange of respective ideas and needs between code developers and HPC experts, as this helped clarifying the issues from either perspective and enabled both sides to interact more smoothly with a well defined focus on the next actions to be taken. For example, the requirements for a full code 'audit' from the EoCoE

[1] www.fz-juelich.de/jsc/jube

WP	Context	Code	Developer	WP1 contact
2	Wind farms	ALYA	Houzeaux (BSC)	Ould-Rouis (MdlS)
2	Ensemble forecasting	ESIAS	Bernd (FZJ)	Lührs (JSC)
3	Photovoltaics	PVnegf	Aeberhard (FZJ)	Di Napoli (JSC)
3	Materials	Metallwalls	Salanne	Haefele (MdlS)
4	Hydrology	PARFLOW	Kollet (FZJ)	Sharples
4	Geothermics	SHEMAT	Qu (RWTH)	Sharples
5	Plasma transport	Gysela	Latu (CEA)	Latu (CEA)

Table 1: Codes participating in first EoCoE benchmarking workshop

and POP perspectives were clarified: here it was decided that the initial benchmarking would take place within and immediately after the workshop by EoCoE WP1 members, whereas more in-depth follow-up analyses could be channelled via a formal request to POP at a later stage.

3.2 May 2016 in Saclay @ MdlS

The second joint EoCoE-POP workshop on benchmarking and performance analysis took place at Maison de la Simulation from 30th May - 2nd June 2016. The objectives and the organization of this workshop were similar to the previous one that took place in Jülich. A first version of the automated performance evaluation was available at that time and it sped up the process of getting started for all participants. This showed us that our methodology is improving and we plan to improve it further for the next workshop that will likely take place during the first semester of 2017.

This event welcomed the first two codes that are not part of the EoCoE consortium: ComPASS, developed at BRGM, the french national geological survey and Telemac, developed at EDF. The developers showed interest in joining this workshop and their feedback was good, they could learn about the performance tools as well as their codes. The framework in which they were welcome was not clear at the moment of the workshop. This experience will be used as a testbed for setting up an appropriate one for future codes that are not part of the consortium.

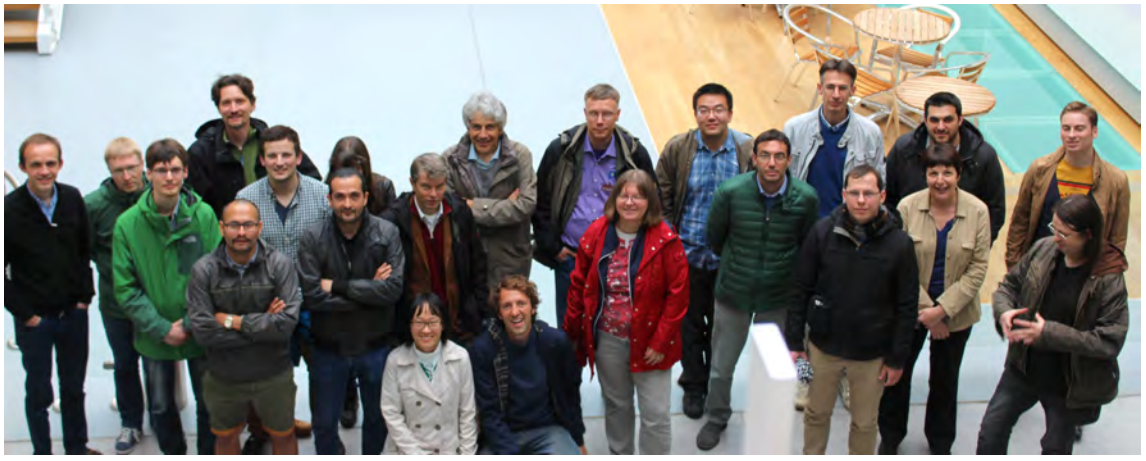


Figure 2: Workshop participants during the second benchmarking workshop

WP	Context	Code	Contact	WP1 Contact
2	meteorology	nowcast system	I. Herlin (INRIA)	Y. Ould Rouis (MdlS)
3	Quantum simulation	CP2K	M. Gusso (ENEA)	S. Lührs (JSC)
3	Molecular DFT	MDFT	M. Levesque (MdlS)	M. Haefele (MdlS)
4	River flows	TELEMAC	A. Joly (EDF)	Y. Ould Rouis (MdlS)
5	Particle transport	EIRENE	P. Börner (FZJ)	T. Breuer (JSC)
ext.	Geothermy	ComPASS	F. Xing (BRGM)	M. Haefele (MdlS)

Table 2: Codes participating in the second EoCoE benchmarking workshop

3.3 April 2017 in Barcelona @ BSC

In a joint effort, the two centres of excellence EoCoE and POP have once again hold a hands-on workshop on HPC benchmarking and performance analysis at Barcelona Supercomputing Centre from 24th to 27th of April 2017. It is the third event of its kind and has been held at BSC in Barcelona and has been supported by the French and the Spanish PATCs.

Improving again an already proven concept, it has brought together 17 experts from topical fields in energy research and tools and 11 experts from HPC science in order to tackle the transition of current R&D codes and applications towards exascale. Most of the scientific applications welcomed to this edition are not part of the EoCoE consortium. This shows the growing impact of EoCoE on the European HPC ecosystem.

The EoCoE performance analysis methodology has once again passed a new level of maturity. Experts from topical fields could really learn how to use advanced performance evaluation tools, get insight of the performance bottlenecks of their applications and bring back home JUBE based benchmarking tool to repeat, in a reproducible manner, this analysis on future optimised versions of their code.

Figure 3: Workshop participants during the third benchmarking workshop

WP	Context	Code	Contact	WP1 Contact
5	MHD	TOKAM3X	Patrick Tamain	M. Lobet (MdlS)
2	Weather	WRF-Solar	Constantinos Demetroullas	M. Lobet (MdlS)
2	Wind farms	ALYA	Albert Coca Abello	Y. Ould-Rouis (MdlS)
1	Solver	MUMPS	Philippe Leleux	Y. Ould-Rouis (MdlS)
1	Solver	Maphys	Gilles Marait	Y. Ould-Rouis (MdlS)
ext.	Geothermics	Compass	Simon Lopez	A. Marin-Laflèche (MdlS)
ext.	Geophysics	DIVA	Xavier Lacoste	A. Marin-Laflèche (MdlS)
ext.	Combustion	PARCOMB	Jordan Denev	T. Breuer (JSC)
ext.	CFD	OpenFOAM	Thorsten Zirwes	T. Breuer + S. Lührs (JSC)
ext.	Material	CP2K	Ari Seitsonen	R. Halver + S. Lührs (JSC)
ext.	Material	DL_MESO	Jony Castagna	R. Halver (JSC)

Table 3: Codes participating in the second EoCoE benchmarking workshop

4. EoCoE performance evaluation report and metrics definition

Performance evaluation has the obvious purpose to uncover bottlenecks and possibly other technical areas of improvement for the codes under consideration. In order to verify the impact and success of code changes it is mandatory to apply it *iteratively and continuously* in a regular manner. In particular, it is *not* sufficient to analyse a code once and from the results create an optimised version of a code in a single step.

4.1 Organizational structure and reporting

The EoCoE management has carefully engineered a lean yet efficient organisational structure which ensures that such an ongoing and continuous process involving code developers and HPC-experts can be achieved and monitored, with a minimum of bureaucratic overhead. The elements and ingredients for this collaborative micro-community are

1. Permanent code teams, consisting of at least one developer and one HPC-experts, to corroborate the collaboration between in a sustainable manner.
2. Code identity card filled by the application developer to initiate the analysis.
3. A well-defined set of global performance metrics to have a common perspective on progress and development. Ideally, most of the initial measures are obtained during an EoCoE performance workshop.
4. The possibility to add further application-specific performance metrics if necessary.
5. A technical infrastructure based on Git which allows all code teams to share their reports and to provide a basis from which best practice methods can be deduced.

Appendix A shows the full performance report for five codes: Metalwalls, ESIAS, Parflow, Gysela and Alya.

4.2 Metrics definition and performance tools

The definition of all global performance metrics is given in table 4. Several tools are used to extract them:

- The UNIX *time* command is used to measure total application wall time and the memory footprint of the first MPI rank of the application.
- Darshan[2] provides all metrics concerning IO
- Scalasca[3] provides all metrics concerning MPI, OpenMP and load balancing
- PAPI[4], used through Scalasca, provides all performance counters
- IdrMem[5] library is used to retrieve the memory footprint on systems where Slurm is not available.

Metrics Global.1, Global.2 and Global.3 might exhibit some inconsistencies as these three measures are extracted from three different runs performed with different binaries. This should not change the global picture as long as similar run times are observed for these three runs.

The MPI time (Global.3) is measured by Scalasca. But Scalasca will also measure MPIIO calls as part of the MPI time measurement, so this MPIIO time is substracted from MPI time during the metric extraction process.

The IO time (Global.2) is measured by Darshan. The IO time itself within Darshan is separated into POSIX and MPIIO time. The POSIX IO handling is a subset of the MPIIO handling, so typically it would be enough just to use the MPIIO timings (if available) to represent the total IO time. Of course there are also applications which use MPIIO and POSIX file IO at the same time. In such a case the maximum of both will be selected to represent the IO time metric.

Memory vs Compute Bound metric (Global.4) is computed with the runtime coming out of two dedicated runs. The two runs use the same amount of MPI ranks and threads but on twice the number of nodes. This leads to depleted resources, and, by using specific deployments, one has the chance to observe memory bandwidth effects. Typically on current dual socket systems, a compact and a scatter run are performed. The compact run packs all the MPI processes and threads on a single socket, whereas the scatter run distributes them evenly on the two sockets. Going from the compact run to the scatter one, the available computing power is kept constant while doubling the available memory bandwidth. As a consequence, if both runs exhibit the same wall time, this means that the memory bandwidth available has no impact on the application. So the code is strongly compute bound and the ratio run time compact / run time scatter is 1.0. On the other hand, if the scatter run is twice as fast, the ratio is than 2.0 and this means that the code is strongly memory bound.

The load imbalance metric (Global.5) gives the potential for code improvement if the load imbalance would be perfectly fixed. Thanks to the trace analysis, Scalasca is able to compute the critical path of the application and the overhead due to load imbalances between ranks/threads. The metric used here is simply the ratio overhead / critical path. For instance, if a 20% load imbalance is measured, fixing perfectly this load imbalance would improve the performance of the code by 20%.

[2] `http://www.mcs.anl.gov/research/projects/darshan/`
[3] `http://www.scalasca.org/`
[4] `http://icl.cs.utk.edu/papi/`
[5] `https://gitlab.maisondelasimulation.fr/dlecas/IdrMem`

		Metric name	Definition	Tool
Global	1	Total Time (s)	Total application wall time	*time*
	2	Time IO (s)	Average time spent in doing IO for each process	Darshan
	3	Time MPI (s)	Average time spent in MPI for each process	Scalasca
	4	Memory vs Compute Bound	1.0 means strongly compute bound, 2.0 means strongly memory bound	cf text
	5	Load Imbalance	Ratio of the load imbalance overhead towards the critical path duration	Scalasca
IO	1	IO Volume (MB)	Total amount of data read and written	Darshan
	2	Calls (nb)	Total number of IO calls	Darshan
	3	Throughput (MB/s)	IO.1 / Global.2	Computed
	4	Individual IO Access (kB)	IO.1 / IO.2	Computed
MPI	1	P2P Calls (nb)	Average number of peer to peer communications per MPI rank	Scalasca
	2	P2P Calls (s)	Average time spent in peer to peer communications per MPI rank	Scalasca
	3	P2P Message Size (kB)	Average message size in peer to peer communications per MPI rank	Scalasca
	4	Collective Calls (nb)	Average number of collective communications per MPI rank	Scalasca
	5	Collective Calls (s)	Average time spent in collective communications per MPI rank	Scalasca
	6	Collective Message Size (kB)	Average message size in collective communications per MPI rank	Scalasca
	7	Synchro / Wait MPI (s)	Average time spent in synchronization per MPI rank	Scalasca
	8	Ratio Synchro / Wait MPI	MPI.7 / Global.3	Computed
Node	1	Time OpenMP (s)	Time spent in OpenMP parallel region	Scalasca
	2	Ratio OpenMP	Ratio of the time spent in OpenMP parallel region towards the total calculation time	Scalasca
	3	Time Synchro / Wait OpenMP	Average time spent in synchronization/OpenMP overhead per thread	Scalasca
	4	Ratio Synchro / Wait OpenMP	Node.4 / Node.1	Computed
Mem	1	Memory Footprint	Average memory footprint of an MPI process	IdrMem/ Slurm
	2	Cache Usage Intensity	Cache Hit / (Cache Hit + miss) in Last Level Cache	PAPI
Core	1	IPC	Total number of instructions executed / Total number of cycles	PAPI
	2	Runtime without vectorization	Total application wall time compiled with vectorization disabled	*time*
	3	Vectorisation efficiency	Global.1 / Core.2	Computed
	4	Runtime without FMA	Total application wall time when compiled with FMA disabled	*time*
	5	FMA efficiency	Global.1 / Core.4	Computed

Table 4: Global performance metrics definition

Synchro / Wait MPI (MPI.7) is calculated by gathering the communication overhead except the pure communication time. This metric sums up the average waiting time per process (e.g. because of a MPI barrier operation) and the synchronisation time to start collective operations.

Metrics Mem.2 and Core.1 use the PAPI counter interface. The implementation of this interface and the available metrics are highly platform specific. Because of that not all applications might allow the extraction of these two metrics.

4.3 Automated metrics extraction process

The generation of the binaries as well as the execution of all necessary runs to generate the metric overview has been automated by using the JUBE environment. Specific metrics as well as a full metric overview can be created with a single JUBE execution.

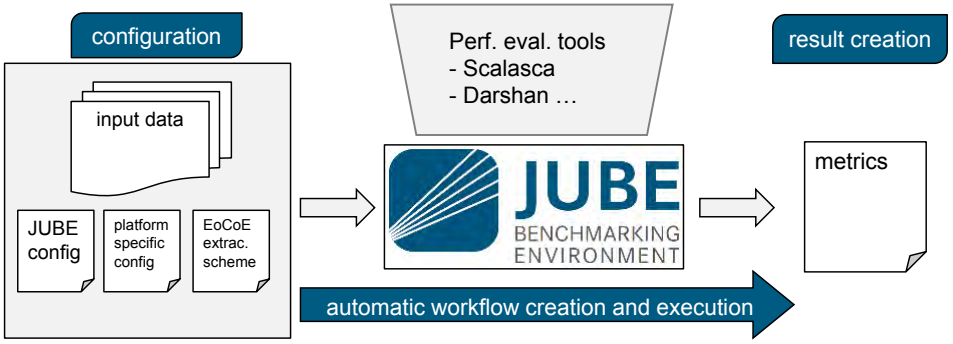


Figure 4: General JUBE workflow for the EoCoE metric extraction process.

Figure 4 shows the main workflow by using the JUBE environment. The application build and run procedure is included into a JUBE configuration file. This part is application specific. Platform specfic configuration datasets and the EoCoE specific execution scheme is added together with the relevant input data for the different benchmarking cases of the application. Within the JUBE environment, different runs are performed as written below. Different metric extraction tools like Scalasca and Darshan are called from within the JUBE environment. The final outcome of the execution is the set of metrics as shown in table 4.

Specifically, for the purpose of automation four separate code binaries are initially needed:

- Normal (ref)
- scalasca instrumented (scalasca)
- Normal plus "no-vectorization" (no-vec)
- Normal plus "no-fma" (no-fma)

If needed a separate executable could be created for the Darshan or the memory instrumentation.

Next, 9 runs are performed:

1. ref ⇒ reference run
2. ref ⇒ memory footprint run
3. ref + Darshan ⇒ IO metrics
4. scalasca profile run ⇒ CPU counters
5. scalasca trace analyse ⇒ Global, MPI, OMP
6. (no-vec) ⇒ Core, vectorization efficiency
7. (no-fma) ⇒ Core, FMA efficiency
8. ref compact run ⇒ mem vs comp. bound
9. ref scatter run ⇒ mem vs comp. bound

The dependencies between the different runs are also shown in Figure 5.

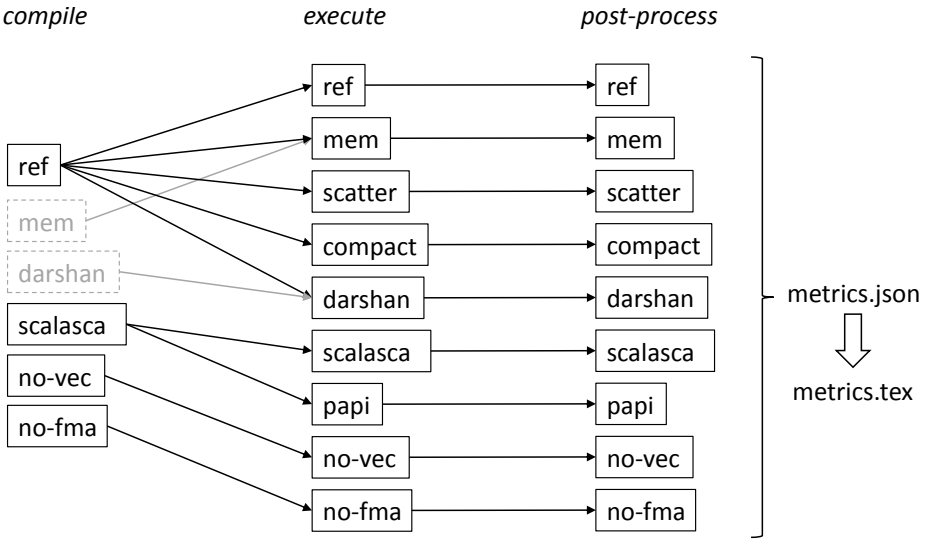


Figure 5: Steps in the automated JUBE workflow for the EoCoE metric extraction process.

All metrics paths could also be executed separately if needed.

A general EoCoE JUBE include file was created to cover these different runs to automatically build the underlying structure. This include file can be used in the application specific part of the metric extraction process, which avoids rewriting the structure multiple times. The file also covers the post-processing of the different tool output formats to create a parse-able final JSON file, which can be transformed into TeX table. To parse the different output formats, two Python scripts were created (mainly to parse the Scalasca and the Darshan output) which takes over the work to convert the binary formats into a ASCII based representation. These scripts are triggered automatically in the post-processing part of the JUBE run and can be used within all applications in the same way.

To allow to use this procedure as a blueprint for other code teams and eventually of

course also by the general public, via dissemination through WP 6, a JUBE template for new codes was created which allows an easier adoption. Within the project the relevant code examples, templates and include files were distributed via the Gitlab infrastructure. The metrics tables in the appendix shows the results of fully automated runs using this architecture.

The automation allows a reproducible way to rerun the full metrics extraction scheme to track code changes and improvements during the application support phase. It can also be used by the code developers themselves within a testing setup to validate future development projects.

5. Codes evaluated on the period Oct 2015 - September 2017

All codes mentioned in table 1 and 2 have established a close cooperation between HPC-experts and code developers following the above mentioned underlying lean management structure. They regularly update and report on their progress by means of the Code Diaries which are maintained on the Git structure along with code changes, automation processes and metrics.

Figure 6 shows the status of all codes regarding the implementation and analysation of the different profiling tools and of the benchmark automatisation process.

Code	WP	JSC Account	Data server account	Gitlab account	JUBE integration	Benchmarks defined in	Tools integrated in JUBE	Allinea report	Score-P profile	Score-P trace	Scalasca analysis	Vampir analysis	Extrae measurement	Parayer analysis	Darshan results	VTune analysis	Advisor analysis	Performance report	Total Progress (%)
GYSELA	WP 5	2	2	2	2	2	2	1	1	1	1	0	2	2	2	0	0	2	100
EIRENE	WP 5	2	2	2	2	2	2	2	2	1	1	0	1	0	0	0	0	2	100
TOKAM3X	WP 5	2	2	2	2	2	2	2	2	2	2	2	2	1	2	1	2	2	100
SHEMAT	WP 4	2	2	2	2	2	1	2	1	1	1	0	2	2	2	0	0	2	100
ParFlow	WP 4	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	100
TELEMAC	WP 4	2	2	2	2	2	2	2	2	2	2	2	1	1	2	2	2	2	100
Metalwalls	WP 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0	2	2	100
PVnegf	WP 3	2	2	2	2	2	2	2	2	2	2	1	0	0	2	0	0	0	90
CP2K	WP 3	2	2	2	2	2	2	0	2	2	1	2	1	0	2	0	0	2	100
MDFT	WP 3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	0	2	100
ALYA	WP 2	2	2	2	2	2	2	2	2	2	2	2	2	2	0	2	1	2	100
ESIAS	WP 2	2	2	2	2	2	2	0	2	2	2	0	0	0	2	0	0	2	100
nowcast system	WP 2	2	2	2	2	2	2	2	0	0	0	0	0	0	2	2	2	2	100
WRF-Solar	WP 2	2	0	0	2	2	1	0	2	2	2	0	2	2	2	0	0	2	100
MUMPS	WP 1	2	2	0	2	2	2	0	1	1	1	1	2	2	0	0	0	2	100
Maphys	WP 1	2	2	0	2	2	2	0	2	2	2	1	2	2	0	0	0	2	100
PARCOMB	ext	0	0	0	2	2	1	0	2	2	2	0	2	2	0	0	0	2	100
OpenFOAM	ext	2	0	0	2	2	2	0	2	2	2	2	2	2	0	0	0	2	100
DL_MESO	ext	2	0	0	2	2	1	2	2	2	2	0	2	2	2	0	0	2	100
Compass	ext	2	0	0	2	2	2	0	2	2	2	0	2	2	0	0	0	2	100
DIVA	ext	2	0	0	2	2	2	2	2	2	2	2	2	2	0	0	0	2	100
Legend																			
0	not started																		
1	in progress																		
2	established																		

Figure 6: Code benchmarking and analysis progress sheet

A. Performance evaluation reports

A.1 Metalwalls

Code ID card

Code name	Metalwalls
Scientific domain	WP3 Molecular dynamic
Description	Metalwalls is a classical molecular dynamics code that simulates energy storage devices: supercapacitors. These devices could replace in the future the batteries used in nowadays hybrid vehicles.
Languages	Fortran90 (20k lines)
Library dependencies	MPI, OpenMP is in project.
Programing models	MPI, OpenMP is in project.
Platforms	• PRACE Tier0 Mare Nostrum (20 MCPUh in 2016) • French Tier1 Occigen (5 MCPUh in 2015)
Scalability results	It has been ported on X86 architectures, scaling results are good up to 1000 cores.
Typical production run	24h on 64 - 512 cores
Input / Output requirement	• Size: 10 GB / 24h run • Single post-processing output: 50MB • Single restart output: 50MB
Application references	Merlet, C.; Rotenberg, B.; Madden, P. A.; Taberna, P.-L.; Simon, P.; Gogotsi, Y.; Salanne, M. Nature Materials. 2012, 11, 306–310
Contact	• Mathieu Salanne (mathieu.salanne@upmc.fr) • Matthieu Haefele (matthieu.haefele@maisondelasimulation.fr)

Performance metrics

Code team:

- Matthieu Haefele (MdlS) for WP1
- Mathieu Salanne (MdlS) for WP3

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

Performance report

According to Table 5, Metalwalls does not seem to need support on IO as less than 1% of execution time is spent in IO on a case that produces much more data than a production run. However, the IO metrics show a very large number of calls compared to the amount data written

	Metric name	03/01/2016
	Test-case	case1
Golbal	Total Time (s)	43.2
	Time IO (s)	0.3
	Time MPI (s)	12.4
	Memory vs Compute Bound	1.1
IO	IO Volume (MB)	35.8
	Calls (nb)	384000
	Throughput (MB/s)	105.0
	Individual IO Access (kB)	0.1
MPI	P2P Calls (nb)	0
	P2P Calls (s)	0.0
	Collective Calls (nb)	2721
	Collective Calls (s)	0.1
	Synchro / Wait MPI (s)	11.7
	Ratio Synchro / Wait MPI	94.8
	Message Size (kB)	908.4
	Load Imbalance MPI	24.8
Node	Ratio OpenMP	0.0
	Load Imbalance OpenMP	0.0
	Ratio Synchro / Wait OpenMP	0.0
Mem	Memory Footprint (B)	66 mB
	Cache Usage Intensity	N.A.
	RAM Avg Throughput (GB/s)	N.A.
Core	IPC	N.A.
	Runtime without vectorisation (s)	46.5
	Vectorisation efficiency	1.1
	Runtime without FMA (s)	44.6
	FMA efficiency	1.0

Table 5: Performance metrics for Metalwalls on the JURECA HPC system

on disk and this is typical for such ASCII based outputs. The implementation of binary based outputs would help here but it is not a priority.

The 30% time spent in MPI is mostly due to load imbalance. The root of this imbalance could be spot thanks to the analysis of the scalasca trace. It resides in the *cgwallrealE* subroutine. The uniform distribution of atom pairs leads here to a load imbalance because some pairs require more computations than others. The implementation of an ad hoc load balancing scheme that would distribute the load between the MPI processes rather than the pairs could solve the issue and let the code scale much better.

Table 5 shows a poor vectorization efficiency. The trace obtained with scalasca allowed us to identify the most intensive parts of the code. A careful examination of these code regions on top of a very good compute bound indicator of 1.1 gives the feeling that the vectorization efficiency could be improved.

During this code investigation, we also noticed a discrepancy between the size of the data structures manipulated in the intensive regions and the global memory footprint measured on Table 5. This memory footprint is much larger than expected, some progress can certainly be made in this area.

Finally, the fact that Metalwalls is a pure MPI code can be a limitation on nowadays multi-core architectures and will definitely be one with the upcoming many-core architectures. An OpenMP implementation that could extract a fine grain parallelism could alleviate this limitation.

As a conclusion, in order to improve Metalwalls, we would recommend the following roadmap:

1. Single core optimizations would cure the memory footprint issue as well as the vectorization one.
2. An ad hoc load balancing scheme would allow the code to scale better in its pure MPI form.
3. An OpenMP implementation would prepare the code for the upcoming architectures.

A.2 Esias

Code ID card

Code name	ESIAS (Ensemble for Stochastic Integration of Atmopheric Simulations)
Scientific domain	WP2: Meteo4Energy
Description	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 intital value and emission factor optimization using four dimensional variational data assimilation.
Languages	Fortran90 and C (500k lines)
Library dependencies	MPI, OpenMP, NetCDF, zlib, libpng, JasPer
Programing models	MPI, OpenMP
Platforms	• IBM Blue Gene/Q JUQUEEN
Scalability results	It has been ported on X86 architectures, scaling results are good up to 524288 cores (512 each ensemble member).
Typical production run	2h on 16384 - 32768 cores
Input / Output requirement	• Size: 1 TB / 24h run (1000 ensemble members, 1 GB each) • Single post-processing output: 10 GB (1000 ensemble members, 1 GB each) • Single restart output: 100 TB (1000 ensemble members, 1 GB each)
Relevant kernel algorithms	Particle Filtering, 4DVAR, Quasi-Newton Minimization (LBFGS), FFT
Software licence	None
Application references	W. C. Skamarock, J. B. Klemp, J. Dudhia et al., "A Description of the Advanced Research WRF Version 3". NCAR Technical Note, NCAR, Boulder, Colo, USA, 2008.
Contact	• Hendrik Elbern (h.elbern@fz-juelich.de) • Jonas Berndt (j.berndt@fz-juelich.de)

Performance metrics

Code team:

- Sebastian Lührs (FZJ) for WP1

- Jonas Berndt (FZJ) for WP2

Case characteristics:

The benchmark setup contains a random simulation period of 6 hours with 240x240x24 gridpoints 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 are used. Parallel NetCDF is used. This benchmark was selected to allow Scalasca Trace analysis, which were not posssible (due to the size) with the 24 hour benchmark. The metrics results by using the Darshan and the Scalasca instrumentation are given in Table 6.

	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
Node	Time OpenMP (s)	N.A.	N.A.
	Ratio OpenMP (%)	N.A.	N.A.
	Synchro / Wait OpenMP (s)	N.A.	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.	N.A.
Mem	Memory Footprint	N.A.	N.A.
	Cache Usage Intensity	N.A.	N.A.
Core	IPC	N.A.	N.A.
	Runtime without vectorisation (s)	N.A.	N.A.
	Vectorisation efficiency	N.A.	N.A.
	Runtime without FMA (s)	N.A.	N.A.
	FMA efficiency	N.A.	N.A.

Table 6: Performance metrics for Esias on the JUQUEEN HPC system

Performance report

I/O and metadata handling can be a bottleneck when using larger numbers of ensemble members. The Scalasca analyses highlighted these parts and the involved overhead. This will be tested in additional benchmarks by using a higher number of ensemble members.

The usage of the NetCDF4 instead of the pNetCDF library was tested but showed up much slower results, because the current implementation within the WRF backend uses only a serial filesystem access if NetCDF4 is activated.

Table 6 also highlights long waiting times within the MPI parts of the code.

The single core performance can still be improved by using a higher compiler optimization level but a direct change to `O3` create stability problems, or will change the final result and has to be checked. Especially vectorization wasn't successfully tested so far. Nevertheless the compiler settings on the BlueGene system could be optimzed by switching the default `O2` setting. This reduces the total execution time up to 25% as shown in Table 6 (current established compile setting on JUQUEEN: `-O3 -qnohot=noarraypad:level=2:novector:fastmath -qstrict=nolibrary -qdebug=recipf:forcesqrt -qsimd=noauto`).

OpenMP is available in WRF underneath the Esias ensemble creation, but currently the feature isn't used. The performance benefit towards a full MPI parallelization will be tested.

A.3 Parflow

Code ID card

Code name	ParFlow
Scientific domain	WP4: Environmental modelling (hydrology)
Description	ParFlow is a 3D variably saturated groundwater flow code with integrated overland flow and a land surface model and is used extensively as part of research on the water cycle in idealized and real data setups as part of process studies, forecasts, data assimilation frameworks, hind-cast as well as climate change projections from the plot-scale to the continent, ranging from days to years.
Languages	C (117k lines), Fortran90 (20k lines, the CLM land surface model)
Library dependencies	Silo (I/O), Hypre (preconditioner), KINSol (SUNDIALS, nonlinear solver)
Programing models	MPI2
Platforms	• Tier0 JUQUEEN IBM BG/Q, JUGENE IBM BG/P, etc. • Tier1 JURECA, etc. • Tier0/1/2 Linux clusters in Europe and the US
Scalability results	It has been ported on x86_64 and BG/Q architectures, scaling results are good up to 32k tasks on BG/Q. See references given below.
Typical production run	Depends on experiment, from minutes up to months; continental model domains (e.g., CONUS on BG/Q on 16384 cores)
Input/Output requirement	Highly variable, depending on spatial resolution, simulation time span and output interval, 40 GB / output interval (Kollet et al., 2010)
Main bottleneck:	CPU
Relevant algorithms:	ParFlow simulates saturated and variably saturated subsurface flow in heterogeneous porous media in three spatial dimensions using a Newton-Krylov nonlinear solver and multigrid-preconditioners.
Software licence:	GNU LGPLi v3
Application references:	• S. F. Ashby, F. R. D., A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations, Nuclear Science and Engineering 124 (1996) 145–159. • J. E. Jones, C. S. Woodward, Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems, Advances in Water Resources 24 (7) (2001) 763–774. doi:http://dx.doi.org/10.1016/S0309-1708(00)00075-0. • S. J. Kollet, R. M. Maxwell, Integrated surface-groundwater flow modeling: A free-surface overland flow boundary condition in a parallel groundwater flow model, Advances in Water Resources 29 (7) (2006) 945–958. doi:http://dx.doi.org/10.1016/j.advwatres.2005.08.006. • S. J. Kollet, R. M. Maxwell, Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model, Water Resources Research 44 (2) (2008) W02402. doi:10.1029/2007WR006004. • S. J. Kollet, R. M. Maxwell, C. S. Woodward, S. Smith, J. Vanderborght, H. Vereecken, C. Simmer, Proof of concept of regional scale hydrologic simulations at hydrologic resolution utilizing massively parallel computer resources, Water Resources Research 46 (4) (2010) W04201. doi:10.1029/2009WR008730. • R. M. Maxwell, L. E. Condon, S. J. Kollet, A high-resolution simulation of groundwater and surface water over most of the continental US with the integrated hydrologic model ParFlow v3, Geoscientific Model Development 8 (3) (2015) 923–937. doi:10.5194/gmd-8-923-2015.
Contact	Stefan KOLLET (stefan.kollet@fz-juelich.de)

Performance metrics

Code team:

- Wendy Sharples (FZJ) for WP1
- Stefan Kollet (FZJ) for WP4 (Carsten Burstedde, Jose Fonseca, Klaus Goergen, Ilya Zhukov, Ketan Kulkarni, Thomas Breuer, Bibi Naz, Jens-Henrik Goebbert, Lukas Poorthuis)

Case1 characteristics:

Domain size	50 x 50 x 40 regular grid
Resources	1 node on Jureca (24 cores)
IO details	Checkpoint written every 1 steps, ⇒ much larger than production
Type of run	development run

	Metric name	06/30/2016
	Test-case	case1
Golbal	Total Time (s)	4.05
	Time IO (s)	0.09
	Time MPI (s)	0.57
	Memory vs Compute Bound	NA
IO	IO Volume (MB)	183.11
	Calls (nb)	24002518
	Throughput (MB/s)	40
	Individual IO Access (kB)	NA
MPI	P2P Calls (nb)	10850
	P2P Calls (s)	0.14
	Collective Calls (nb)	2721
	Collective Calls (s)	0.01
	Synchro / Wait MPI (s)	0.796
	Ratio Synchro / Wait MPI	0.35
	Message Size (kB)	7.09
	Load Imbalance MPI	0.915
Node	Ratio OpenMP	0.0
	Load Imbalance OpenMP	0.0
	Ratio Synchro / Wait OpenMP	0.0
Mem	Memory Footprint (B)	23.1 mB
	Cache Usage Intensity	N.A.
	RAM Avg Throughput (GB/s)	0.008
Core	IPC	N.A.
	Runtime without vectorisation (s)	3.89
	Vectorisation efficiency	1
	Runtime without FMA (s)	3.83
	FMA efficiency	1.0

Table 7: Performance metrics for ParFlow on the JURECA HPC system

Performance report

ParFlow has undergone extensive performance analysis in addition to the performance metrics gathered (see PoP "ParFlow_POP_audit.pdf" report committed to the EoCoE ParFlow/docs repository).

ParFlow performance on a KNL cluster has also been assessed with a separate PoP report due at the end of this month.

According to Table 7, ParFlow does not seem to need support on IO as less than 1% of execution time is spent in IO on a case that produces as much data as production run. However binary files are not very portable compared to the standard climate science simulation file format, netCDF and thus much time is wasted in postprocessing- converting between binary to netCDF. In addition, there is a lot of postprocessing of data needed to turn output into scientifically valuable data, with the use of insitu visualization, these outputs could be generated interactively on the fly, further reducing postprocessing overheads.

It was determined that load imbalance was not an issue in this symmetric case upon analysis with scalasca (see PoP report) however in "real life" cases where much of the domain is inactive due to a land sea mask, adaptive mesh refinement would be desireable.

Memory footprint is an issue when scaling up to above 64,000 processors (on Juqueen- see PoP report), due to all cells having the COMPLETE grid information. Employment of an adaptive mesh refinement library would mean that each cell only stores neighbouring grid information, thus lowering the memory footprint.

Table 7 shows a fairly decent vectorization efficiency. Using Vector Advisor it was determined that nearly all loops that "could" be vectorised have many dependencies so it would take a huge amount of refactoring to get any better than this.

At the moment ParFlow is unable to take advantage of booster architecture. This is due to a heavy reliance on the solver library KINSOL. As KINSOL is tightly meshed with ParFlow at the moment this will take a considerable amount of refactoring.

As a conclusion, in order to improve ParFlow, we would recommend the following roadmap:

1. Memory improvement and load imbalance would be improved by addition of adaptive mesh library
2. Booster architecture could be utilized once reliance on KINSOL is removed (E.g. PETSc)- first evaluate and quantify the benefits using a MiniApp
3. NetCDF IO would improve portability and reduce postprocessing overheads
4. Postprocessing overheads would be further reduced with insitu visualization

A.4 Gysela

Code ID card

Code name	Gysela
Scientific domain	WP5 Fusion
Description	The GYSELA code is a non-linear 5D global gyrokinetic full-f code which performs flux-driven simulations of ion temperature gradient driven turbulence (ITG) in the electrostatic limit with adiabatic electrons. No assumption on scale separation between equilibrium and perturbations is done.
Languages	Fortran 90 + some routines in C ($\approx$ 50 000 lines)
Library dependencies	MPI, OpenMP, HDF5
Programing models	MPI, OpenMP
Platforms	• Fusion dedicated international machines (Helios, Marconi) • French Tier1 (Occigen, Occigen2, Curie, Cobalt) Total core-hours consumed in 2016: 113.6 Mh
Scalability results	• Strong scaling: 60% relative efficiency at 65 kcores on Curie (x86) and Turing (BG/Q) • Weak scaling: 91% relative efficiency at 459 kcores on Juqueen (BG/Q)
Typical production run	200h on 4096 cores
Input / Output requirement	• Size: 400 GB / 24h run (restart files) • Single post-processing output: 100 GB • Single restart output: 200 GB
Main bottleneck	complex memory patterns, communication costs at very large scale
Relevant kernel algorithms	Semi-Lagragian scheme, cubic spline interpolation, FFT, 2D poisson solver
Software licence	CEA proprietary software
Application references	V. Grandgirard & al., A 5D gyrokinetic full- global semi-Lagrangian code for flux-driven ion turbulence simulations, Computer Physics Communications, Volume 207, October 2016, Pages 35-68, ISSN 0010-4655, http://dx.doi.org/10.1016/j.cpc.2016.05.007
Contact	• virginie.grandgirard@cea.fr • guillaume.latu@cea.fr

Performance metrics

Code team:

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

Small 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 4 steps instead of 100 ⇒ larger than production
Type of run	development run

Large 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 8 steps instead of 100 ⇒ larger than production
Type of run	production run

Table 8: Performance metrics for Gysela on the JURECA HPC system (Small case)

Performance report

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. 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 `Large 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 hyperthreading 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 vaules 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 hyperthreading 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.

Efficiency of vectorisation should be investigated. One can expect better speedup than a factor 2 with (31.2s) or without vectorisation (68s).

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.

A.5 Alya

Code ID card

Code name	Alya
Scientific domain	Computational mechanics. In this project used for CFD for Wind energy
Description	The Alya System is the BSC simulation code for multi-physics problems, specifically designed to run efficiently in supercomputers. See web page: https://www.bsc.es/computer-applications/alya-system
Languages	Fortran90 (750k lines)
Library dependencies	Metis.
Programing models	MPI, OpenMP is in project.
Platforms	PRACE Tier0: Marenostrum, SuperMuc, Fermi, Juqueen, etc. Int: Blue Waters
Scalability results	It scalability has been tested in several Tier 0 European and international Supercomputers up to 130000 cores (SuperMuc).
Typical production run	12h on 128 - 512 cores
Input / Output requirement	• Size: 10 GB / 24h run • Single post-processing output: 500MB • Single restart output: 500MB
Main bottleneck	Memory access.
Relevant kernel algorithms	• Finite Element matrix calculation. • Iterative Solvers (GMRES, Deflated CG)
Software licence	It depends.
Application references	Alya: Towards Exascale for Engineering Simulation Codes', M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, and J.M. Cela. The International Conference for HPC, Networking, Storage, and Analysis. http://arxiv.org/pdf/1404.4881v1.pdf
Contact	• Guillaume Houzeaux (guillaume.houzeaux@bsc.es) • Mariano Vazquez (mariano.vazquez@ bsc.es)

Performance metrics

Code team:

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

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

	Metric name	jan2016.json	apr2016.json
Golbal	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
Node	Ratio OpenMP	N.A.	N.A.
	Load Imbalance OpenMP	N.A.	N.A.
	Ratio Synchro / Wait OpenMP	N.A.	N.A.
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

Table 9: Performance metrics for Alya on the JURECA HPC system

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 10: Detailed time performance on JURECA - January

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

Table 11: Detailed time performance on JURECA - April

Performance report

The Alya application submitted to EOCOE contains 2 major modules : NASTIN module, solving incompressible Navier Stokes equations and TURBUL module, solving turbulence equations.

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. The code has a master-slaves organization, with the rank 0 as master, and the rest as calculation processes.

The first performance audit results in january allow us to make the following observations :

- Low memory consumption for this size of benchmark, compared to the memory of 1 node (<10%).
- The scatter vs compact results show a strong memory bound behavior, especially in the solver parts that run 40% faster in the scatter mode.
- The time measurements through direct instrumentation, show the following distribution in the different parts of the code (wall time, expressed in seconds and percentage of the total) :
  Total time : 385 s , 100 %
  - NASTIN module : 204 s , 52 %
    - matrix assembly : 67 s , 17 %
    - solver : 133 s , 34 %
  - TURBUL module : 125 s , 32 %
    - matrix assembly : 88 s , 23 %
    - solver : 32 s , 8 %
- The Scorep trace collection introduces a rather big overhead (20%), despite filtering a long list of subroutines. We can read in the perf eval table an MPI time representing 20% of the execution time, most of it due to synchronization. But a close look to both paraver and Scalasca traces show that MPI occupies only 5.8% of the calculation loop on the calculation processes, which concludes in a good balance and MPI performance. The rest of the 20% are spent in rank 0 (master process) waiting for the calculation to be done (3.6%) and the basic serial IOs used in this benchmark (9.5% in the read and 1.8% in the write). However, these time losses should be put into perspective : The read time becomes less important for a production simulation length. In addition, a serial program allows to prepare very large data prior to the execution, so Alya can use a parallel read. Alya also has an HDF5 output option.
- Paraver analysis has shown a mean of 2 instructions per cycle in the matrix assembly parts. That denotes of a good performance. However the comparison between the ref and no-vec runs shows a very poor vectorization performance in these regions : especially

Figure 7: Screenshot of the VTune profiling of the original code

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle Poor Ok Ideal Over)	Spi. Tim.	Over... Time	Module	Function (Full)	Source File	Start Address
nsi_elmmat	38.898s	0s	0s	Alya.x	nsi_elmmat	nsi_elmmat.f90	0xc76630
bcsrai	21.695s	0s	0s	Alya.x	bcsrai	bcsrai.f90	0x442fc0
csrase	12.288s	0s	0s	Alya.x	csrase	csrase.f90	0x4d7770
tur_elmco2	10.257s	0s	0s	Alya.x	tur_elmco2	tur_elmco2.f90	0x1011dd0
tur_elmmsu	9.665s	0s	0s	Alya.x	tur_elmmsu	tur_elmmsu.f90	0x1023020
nsi_assemble_schur	6.823s	0s	0s	Alya.x	nsi_assemble_schur	nsi_assemble_schur.f90	0xc2d7e0
bsyje5	5.585s	0s	0s	Alya.x	bsyje5	bsyjes.f90	0x47a570
jacobi	5.473s	0s	0s	Alya.x	jacobi	jacobi.f90	0x616380
tur_elmop2	4.608s	0s	0s	Alya.x	tur_elmop2	tur_elmop2.f90	0x10260e0
for_cpstr	4.143s	0s	0s	Alya.x	for_cpstr		0x11040f0
elmca2	3.422s	0s	0s	Alya.x	elmca2	elmca2.f90	0x576fc0
ker_proper	3.385s	0s	0s	Alya.x	ker_proper	mod_ker_proper.f90	0x9e1f80
_intel_fast_memcmp	3.199s	0s	0s	Alya.x	_intel_fast_memcmp		0x111b060
prodxy	2.473s	0s	0s	Alya.x	prodxy	prodxy.f90	0xf22090
nsi_elmres	2.097s	0s	0s	Alya.x	nsi_elmres	nsi_elmres.f90	0xcaa760
gmrpls	1.979s	0s	0s	Alya.x	gmrpls	gmrpls.f90	0x5fc300
tur_elmgat	1.928s	0s	0s	Alya.x	tur_elmgat	tur_elmgat.f90	0x1019390
tauadr	1.539s	0s	0s	Alya.x	tauadr	tauadr.f90	0xfff3e0
nsi_elmsch	1.478s	0s	0s	Alya.x	nsi_elmsch	nsi_elmsch.f90	0xcb6b90
pscom_progress	1.401s	0s	0s	libpsco...	pscom_progress	pscom.c	0x6a30
memrp2	1.316s	0s	0s	Alya.x	memrp2	mod_memchk.f90	0xa356c0
solli1	1.279s	0s	0s	Alya.x	solli1	linlet.f90	0x6812e5
nsi_elmope_omp	1.260s	0s	0s	Alya.x	nsi_elmope_omp	nsi_elmope_omp.f90	0xc92150
tur_stdkep	1.218s	0s	0s	Alya.x	tur_stdkep	tur_stdkep.f90	0x106e920
pow	1.198s	0s	0s	Alya.x	pow		0x110c760
diagon	1.080s	0s	0s	Alya.x	diagon	diagon.f90	0x5276a0
vecnor	0.909s	0s	0s	Alya.x	vecnor	vecnor.f90	0x107c0d0
elmlen	0.890s	0s	0s	Alya.x	elmlen	elmlen.f90	0x587f20
shm_do_read	0.869s	0s	0s	libpsco...	shm_do_read	pscom_shm.c	0x11ae0
ufd_poll	0.859s	0s	0s	libpsco...	ufd_poll	pscom_ufd.c	0x14ea0
pscom_unlock	0.829s	0s	0s	libpsco...	pscom_unlock	pscom.c	0x6640
elmder	0.800s	0s	0s	Alya.x	elmder	elmder.f90	0x5825f0
__intel_memset	0.636s	0s	0s	Alya.x	__intel_memset		0x11249d0
all_precon	0.590s	0s	0s	Alya.x	all_precon	all_precon.f90	0x40fa30
deflcg	0.549s	0s	0s	Alya.x	deflcg	deflcg.f90	0x4ff590
nsi_turbul	0.479s	0s	0s	Alya.x	nsi_turbul	nsi_turbul.f90	0xd65700
nsi_elmga3	0.440s	0s	0s	Alya.x	nsi_elmga3	nsi_elmga3.f90	0xc6a4a0
tur_elmtss	0.440s	0s	0s	Alya.x	tur_elmtss	tur_elmtss.f90	0x1035890

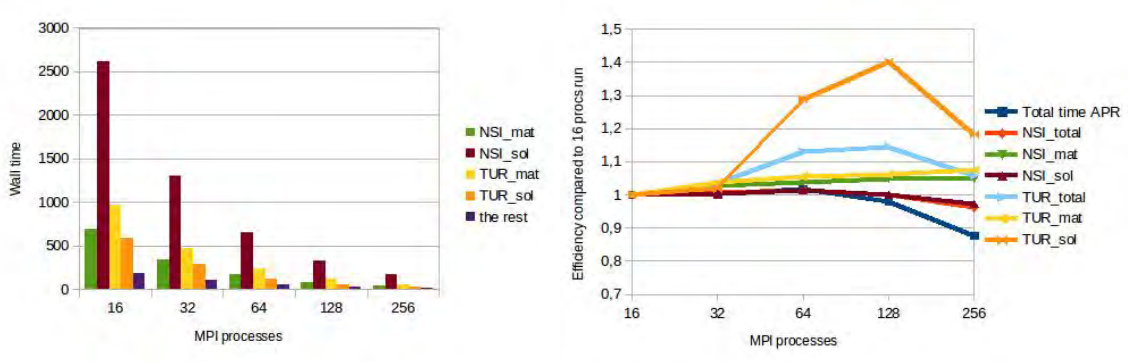
in NASTIN matrix assembly that runs 10% faster when vectorization is disabled. This strongly suggests the necessity to improve the vectorization of this part.

- VTune hotspot profiling shows that 75% of the time is spent in the 12 hottest subroutines. These top 12 hottest are :
  - In Nastin matrix assembly : *nsi_elmmat, nsi_assemble_schur, jacobi, elmca2, ker_proper*
  - In Turbul mat assembly : *csrase, tur_elmco2, tur_elmmsu, tur_elmop2, ker_proper*
  - In solvers : *bcsrai, bsyje5*
- IO performance evaluation has been conducted later in a separate step, using HDF5 parallel output. The outcome, using darshan and wall time measurement, shows negligible output time in the regular use, even for large models (64Melem on 64 nodes, generating 4.6Gb files). A regular use, according to Alya team, generates an output every 100 timesteps. In order to give an idea of IO time, Increasing the outputs frequency to every time step gives the following overheads on a weak scaling :
  - 1Melem on 1 node (16 processes) : still under 1% overhead.
  - 8Melem on 8 nodes (128 processes) : 8% overhead.
  - 64Melem on 64 nodes (1024 processes) : 15% time overhead.
- Strong scaling results show a good scaling of the main parts of the program, as shown in figure 8.

In conclusion

1. No identified need of IO level optimization.

Figure 8: ALYA strong scaling on 16, 128 and 1024 processes, on MareNostrum - Model size : 8Melem



2. MPI performance judged good in the actual context and code version.

3. Matrix assemblies (40% of exec time) : pathologies identified and potential optimization possibilities on the sequential level : memory and cache accesses, vectorization, padding... The code holders expressed a big need on this point.

4. Solvers (42%) : pathologies identified on the sequential level, mainly memory access indirections and unpredictible loop boundaries. The problem may be solvable with a large data restructuring. An other choice, prefered by code holders, is to put efforts into the solver's method itself, within WP1 task 2 dedicated to linear algebra.

A.6 Eirene

Code ID card

Code name	EIRENE
Scientific domain	Scientific domain: WP5: Monte Carlo particle transport
Description	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 linked, iteratively, into 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.
Languages	Fortran95 (∼170.000 lines)
Library dependencies	MPI
Programing models	MPI
Platforms	• PRACE Tier0 JUQUEEN • German Tier1 JURECA
Scalability results	Case dependent, linear speed up to 1500 cores tested
Typical production run	Case dependent (linear or non-linear mode) 20 sec (linear, 1D) up to three month (non-linear, iterative, 3D), Monte Carlo statistical error always prop.to 1/sqrt(CPU).
Input / Output requirement	• Size: <10 GB • Single post-processing output: 20 MB • Single restart output: 20 MB
Relevant kernel algorithms	linear Monte Carlo particle transport
Software licence	NONE
Application references	[1] Reiter, D.; Baelmans, M.; Börner, P. The EIRENE and B2-EIRENE codes Fusion Science and Technology, Vol 47, No. 2 (2005), p172-186 [2] A.S. Kukushkin, H.D. Pacher, V. Kotov, G.W. Pacher, D. Reiter, Finalizing the ITER divertor design: the key role of SOLPS modelling, Fusion Engineering and Design 86 (2011), pp. 2865-2873
Contact	• Petra Börner (p.boerner@fz-juelich.de) • Detlev Reiter (d.reiter@fz-juelich.de)

Performance metrics

Code team:

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

Benchmark characteristics:

The Eirene code is often coupled to other codes. For example, in the SOLPS-ITER code package the Eirene code is linked together with the B2 code, and Eirene is called by B2 at each iteration. The following test cases are inspired by a SOLPS-ITER test case, but they use the standalone version of Eirene.

The physical parameters for the test case correspond to the so-called ASDEX Upgrade plasma with 5 bulk ion species 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

Here we specify 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. We use 12 iterations 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. In the performance tables (Table 12) we will use the unit kP/s (thousand particles per second).

Performance report

Allinea Performance Reports

Figure 9 shows the summary of the Allinea Performance report for the two test cases. We

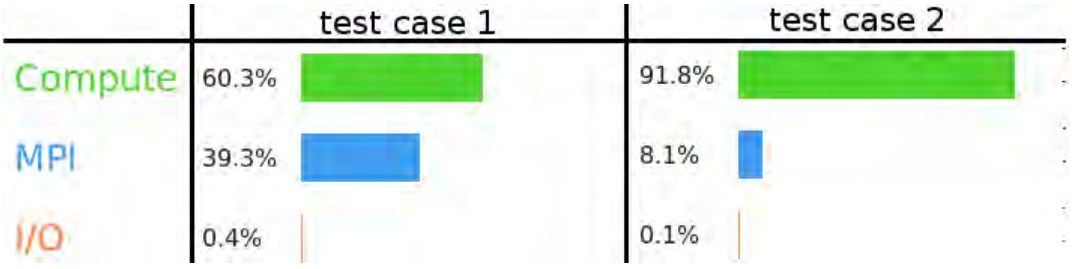


Figure 9: Allinea performance report for two test cases

can see that test case 1 spends large part of the execution time as MPI communication. This is a known problem that is caused by load imbalance. The current parallelization strategy is not optimal for test case 1. Test case 2 has lower MPI communication overhead.

For both the test cases, around 30% of the computation time is spent as numeric operation out of which only 0.7% are the vector numeric operations. Eirene implements a conventional history based Monte Carlo algorithm which is inherently scalar. Particles are followed individually,

therefore it is not surprising that the vectorization is very low. The rest of the computation time (70%) are memory accesses according to the Allinea Performance Report.

EoCoE benchmark table

Table 12 shows the summary of the benchmarks. Test case 2 has 5.8 seconds of MPI

	Metric name	test case 1	test case 2
Golbal	Total Time (s)	64.4	69.0 s, 666 kP/s
	Time IO (s)	0.2	0.1
	Time MPI (s)	36.0	5.8
	Memory vs Compute Bound	1.1	1.0*
IO	IO Volume (MB)	498.3	202.7
	Calls (nb)	650279	260403
	Throughput (MB/s)	2142.3	1842.7
	Individual IO Access (kB)	0.9	0.9
MPI	P2P Calls (nb)	37	15
	P2P Calls (s)	0.0	0.0
	Collective Calls (nb)	191543	76590
	Collective Calls (s)	2.2	0.9
	Synchro / Wait MPI (s)	10.8	4.4
	Ratio Synchro / Wait MPI	29.9	76.5
	Message Size (kB)	24.7	24.6
	Load Imbalance MPI	13.4	4.5
Node	Ratio OpenMP	N.A.	N.A.
	Load Imbalance OpenMP	N.A.	N.A.
	Ratio Synchro / Wait OpenMP	0.0	0.0
Mem	Memory Footprint (B)	148 mB	151 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)	65.8	68.0s, 664 kP/s
	Vectorisation efficiency	1.0	1.0*
	Runtime without FMA (s)	63.8	68.8s, 714 kP/s
	FMA efficiency	1.0	0.94*

Table 12: Performance metrics for Eirene on the JURECA HPC system for the two test cases. For test case 2, the figure of merit is the number of particles calculated per second, given as kP/s (thousand particles/sec). The starred (*) metrics are derived from kP/s values.

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 the 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 might be only measurement fluctuation.

A.7 MDFT

Code ID card

Code name	MDFT
Scientific domain	WP3 Chemistry - embedding method
Description	The molecular density functional theory and the MDFT code describe the solvation of arbitrary solutes like surfaces, proteins or electrodes at the molecular scale, like in a molecular simulation, but at a numerical cost reduced by orders of magnitude.
Languages	Fortran90, rev 2008, (20k lines)
Library dependencies	FFTW3
Programing models	OpenMP, MPI will be implemented after advices resulting from this EoCoE meeting.
Platforms	• PRACE Tier0: none • Tier1: none
Scalability results	Up to few cores by OpenMP. By the way it must be coupled to high performance codes for EoCoE (ab init and metalwalls).
Typical production run	Few minutes with 1 to 4 OpenMP threads
Input / Output requirement	• Single postprocessing output: 0.5 GB • Single restart output: 0.5 GB
Main bottleneck	memory access and memory capacity : very large arrays
Relevant kernel algorithms	FFT, various poisson solvers, spherical harmonics transforms, convolutions.
Software licence	none
Application references	• M. Levesque, R. Vuilleumier, D. Borgis, J. Chem. Phys. 137, 034115 (2012) • G. Jeanmairet, M. Levesque, R. Vuilleumier, D. Borgis, J. Phys. Chem. Lett. 4, 619 (2013) • V. Sergiievskyi, G. Jeanmairet, M. Levesque, D. Borgis, J. Phys. Chem. Lett. 5,1935 (2014)
Contact	• Maximilien LEVESQUE (maximilien.levesque@ens.fr) • Daniel BORGIS (daniel.borgis@ens.fr)

Performance metrics

Code team:

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

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 (1GB)
Comments	typical targetted production run, initially takes few tens of minutes, and uses 10 GB memory. It was chosen as a reference case for the performance evaluation

	Metric name	Initial (01'2017)	After app support (03'2017)			
				threads		
		w/o OpenMP	w/o OpenMP	4	8	24
Golbal	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			

Table 13: Performance metrics for MDFT on the JURECA HPC system - Compiler : gfortran - case : benchmark_mid

Performance report

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. Such quantities typically require 10 minutes to be computed with MDFT while several tens of hours using canonical molecular simulations.

The approach, recently developped at Ecole Normale Supérieure and Maison de la simulation, still lack maturity in the HPC domain. The EoCoE performance evaluation was not only the occasion to get better insight of the code but also to define the HPC roadmap on the intermediate and longer terms.

The first great added-value of the performance evaluation was to force the definition of test cases. These constraints of running a production test-case has exhibited a large overhead in the initialisation phase, that was fixed quickly by C. Gageat. This contribution was made before the generation of the first full performance evaluation, refered to as the column "01'2017" in table 13.

EoCoE Jube Perf Eval Results (January 2017)

All the analysis was performed on Jureca, applied on the chosen benchmark. The initial results are represented in the column "01'2017" of Table 13, and are described below :

- **Total time :** the program initially ran in 1530 seconds (25mn30s)
- **IO :** the IO is not problematic for this size of run, and takes less than 3 seconds. However the whole post processing takes around 110 seconds (1.3% of execution time). The output is composed of binary files, and ascii files (smaller). Darshan measurements give a total output volume of about 1GB, with a bandwidth of 500MB/s and default individual IO accesses of 4.03kB. Files however are larger, and written at once at the end of the program. It is therefore possible to think of a larger buffering.
- **OpenMP :** no OpenMP or multithreading was implemented in the code.
- **Memory footprint :** the memory footprint for this case size is around 10GB. When testing bigger cases, we very fast exceeded memory limits on a normal Jureca node.
  During this performance evaluation, we tracked down the parts of the code that require large memory allocations : MDFT lies on the minimization of a functional of a 6 dimensional field. It is thus heavily relying on a state of the art minimization technique called L-BFGS[6]. LBFGS represents 40% of the memory requirements. This turns out to be the a critical point for MDFT and has triggered discussions on the global roadmap of the code.
- **Cache usage intensity :** the L3 cache usage efficiency (from PAPI metrics) is average with 50% misses. The computation of energy_cproj_mrso, which represents one of the largest parts, consists of successive loops with different access orders on the data (ex : ffts on cartesian coordinates alternating with ffts on angular coordinates ...) Therefore there is no data structure that can have contiguous accesses everywhere, and the data copies are costly.
- **IPC :** the code computes at an average of 2.15 instructions per CPU cycle (from PAPI metrics). The processor works at 3.2 cycles per second, which is close to its Max Turbo frequency.
- **Vectorisation efficiency :** weak SIMD performance.

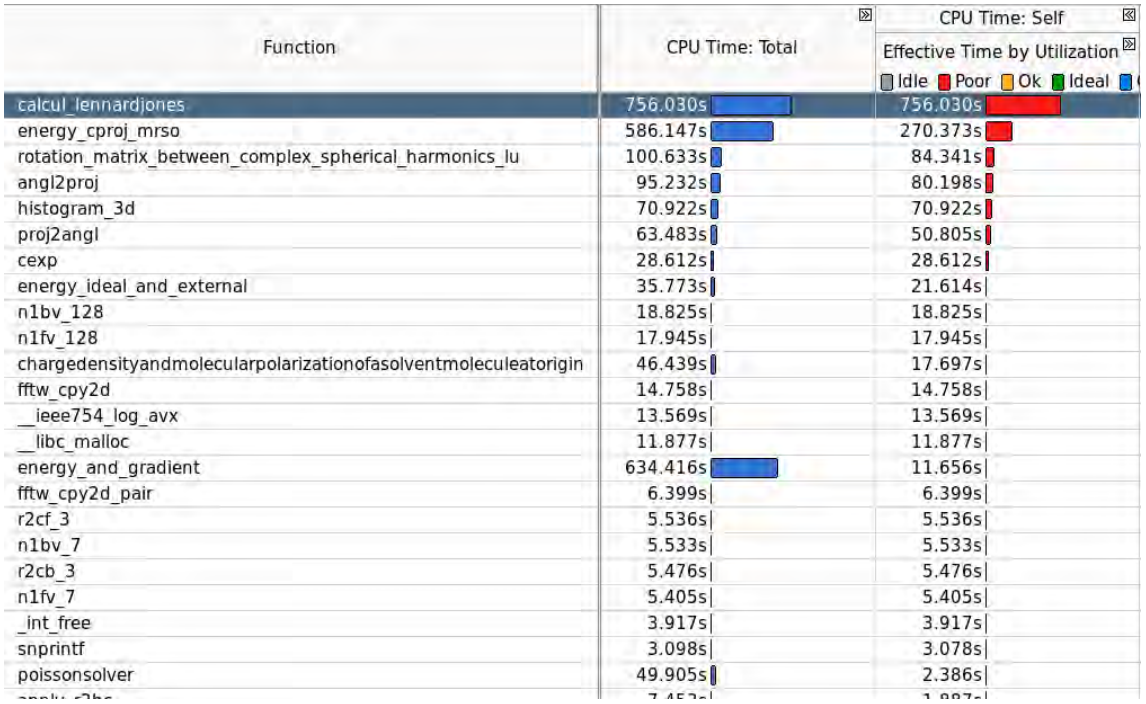
Hotspot analysis (January 2017)
With a closer look at score-p/VTune profiles, we can determine the hottest spots.

The table [figure 10] is a caption from VTune profiling. The left column represents the functions names, the right column represents the CPU exclusive (self) time, and the middle column represents the inclusive CPU time. Inclusive (resp. exclusive) time means the time spent in the function, including (resp. excluding) inner calls to other functions. Here are few comments about the hottest spots :

1. The **calcul_lennardjones** routine is the surprise of this analysis. While it was negligible on the first tests, it increases strongly with the number of solute molecule sites. It contains an inbricated loop, that can be distributed, and calculations that can be skipped for the most common particular case.
2. **energy_cproj_mrso** is the expected biggest calculation of the code. It represents a total of 38% of the total execution time, calling other expensive routines like *rotation_matrix_between_complex_spherical_harmonics_lu*, *angl2proj*, *proj2angl*, and fast fourrier transforms. We can identify 5 parts in this routine, that access the data - *deltarho_p* 4 dimensional array - in a different order. The first dimension of this array represents the angular discretization, and the 3 other dimensions the cartesian discretization. These 5 parts can be separately distributed.
   The parts 1 and 5 calculate angular projections. They call *angl2proj* and *proj2angl* on

[6] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on Mathematical Software, 23(4):550–560, December 1997.

Figure 10: VTune profiling for MDFT original code, on jureca, case : benchmark_mid, compiler : gfortran



Function	CPU Time: Total	CPU Time: Self Effective Time by Utilization Idle Poor Ok Ideal
calcul_lennardjones	756.030s	756.030s
energy_cproj_mrso	586.147s	270.373s
rotation_matrix_between_complex_spherical_harmonics_lu	100.633s	84.341s
angl2proj	95.232s	80.198s
histogram_3d	70.922s	70.922s
proj2angl	63.483s	50.805s
cexp	28.612s	28.612s
energy_ideal_and_external	35.773s	21.614s
n1bv_128	18.825s	18.825s
n1fv_128	17.945s	17.945s
chargedensityandmolecularpolarizationofasolventmoleculeatorigin	46.439s	17.697s
fftw_cpy2d	14.758s	14.758s
__ieee754_log_avx	13.569s	13.569s
__libc_malloc	11.877s	11.877s
energy_and_gradient	634.416s	11.656s
fftw_cpy2d_pair	6.399s	6.399s
r2cf_3	5.536s	5.536s
n1bv_7	5.533s	5.533s
r2cb_3	5.476s	5.476s
n1fv_7	5.405s	5.405s
_int_free	3.917s	3.917s
snprintf	3.098s	3.078s
poissonsolver	49.905s	2.386s

the contiguous angular dimension array for each cartesian coordinate.
The parts 2 and 4 compute three-dimensional ffts on the cartesian coordinates for each angle (non contiguous access). They therefore require costly data copies.
Finally, the central part 3 performs harmonics calculations for every angle and every cartesian coordinate. The inner loop contains a lot of conditional and select case statements, making this part not suited for SIMD.

MDFT HPC roadmap

1. Short term: memory footprint reduction of MDFT such that the kind of test-case used here fits into the memory of a current laptop, i.e. 8 GB. The replacement of the pseudo Newton-Raphson LBFGS minimisation technique by a steepest descent or conjugate gradients would cut the memory requirements by 40%. The price to pay is more iterations to reach convergence, that are higher time to solution. Maybe MDFT should be able to choose automatically the minimization technique to be used depending on the size of the system to be studied. C. Gageat & M. Levesque will implement this on their own and results will be presented in the application support section.

2. Short term: OpenMP implementation is necessary to trigger a more efficient use of the computing resources (memory bandwidth, caches and computation power). MDFT is heavily memory bound, so accessing to the full memory bandwitdh is critical to ensure the best utilisation of a single node. Even on laptops, the current multi-core aspect of modern processors prevents a single thread from accessing the full memory bandwitdth. Results will be presented in the application support section.

3. Long term: MPI implementation will be necessary on long term, in order to allow the computing of bigger problems, and with a strong emphasis on the compatibility with the

MPI implementation of a target electronic DFT code and MetalWalls. This represents a large amount of work and will probably trigger a formal application support request in order to get additional resources from WP1.

A.8 PVnegf

Code ID card

Code name	PV_NEGF
Scientific domain	WP3: mesoscopic opto-electronic device simulation
Description	PV_NEGF is a program for the simulation of mesoscopic opto-electronic devices based on the non-equilibrium Green's function formalism, primarily conceived for nano-structure photovoltaics.
Languages	Fortran90 (∼10k lines)
Library dependencies	Lapack, mkl
Programing models	not available (serial)
Platforms	Any
Scalability results	not available
Typical production run	12 h (1 cpu)
Input / Output requirement	∼100 MB / cpu (single run)
Application references	Theory and simulation of quantum photovoltaic devices based on the non-equilibrium Green's function formalism, U. Aeberhard, Journal of Computational Electronics 10, 394 (2011)
Contact	Urs Aeberhard (u.aeberhard@fz-juelich.de)

Performance metrics

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, eletron-phonon coupling - POP+AC, evaluation of LDOS, carrier density and current, scattering rates, absorption coefficient

Performance report

	Metric name	24/02/2017
Global	Total Time (s)	123
	Time IO (s)	5.91
	Time MPI (s)	N.A.
	Memory vs Compute Bound	0.98
	Load Imbalance (%)	67.25
IO	IO Volume (MB)	169.11
	Calls (nb)	1682277
	Throughput (MB/s)	28.60
	Individual IO Access (kB)	0.10
MPI	P2P Calls (nb)	N.A.
	P2P Calls (s)	N.A.
	P2P Calls Message Size (kB)	N.A.
	Collective Calls (nb)	N.A.
	Collective Calls (s)	N.A.
	Coll. Calls Message Size (kB)	N.A.
	Synchro / Wait MPI (s)	N.A.
	Ratio Synchro / Wait MPI (%)	N.A.
Node	Time OpenMP (s)	48.96
	Ratio OpenMP (%)	39.21
	Synchro / Wait OpenMP (s)	20.28
	Ratio Synchro / Wait OpenMP (%)	44.37
Mem	Memory Footprint	N.A.
	Cache Usage Intensity	0.82
Core	IPC	1.10
	Runtime without vectorisation (s)	107
	Vectorisation efficiency	0.87
	Runtime without FMA (s)	123
	FMA efficiency	1.00

Table 14: Performance metrics for PVnegf on the JURECA HPC system

A.9 Shemat

Code ID card

Code name	SHEMAT-Suite
Scientific domain	WP4: Geothermal Energy
Description	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 (MonteCarlo, EnKF, Bayes Inversion) as well as functionalities for two-phase flow to simulate CO_2 sequestration.
Languages	Fortran90 (1,321,811 lines)
Library dependencies	MPI, OpenMP (two-phase flow: PETSc)
Programing models	MPI, OpenMP (two-phase flow: PETSc)
Platforms	• RWTH Aachen University Cluster • Jülich JURECA (0.2 MCPUh in 2016)
Scalability results	Scaling results are good up to 96 cores.
Typical production run	24h on 12 - 96 cores
Input / Output requirement	• Size: 10 GB / 24h run • Single post-processing output: 50MB • Single restart output: 50MB • Input File: 1.5 GB
Application references	Clauser, C. (ed), 2003. *Numerical Simulation of Reactive Flow in hot Aquifers using SHEMAT/Processing Shemat*, Springer, Heidelberg-Berlin. Rath, V., Wolf, A., Bücker, M., 2006. *Joint three-dimensional inversion of coupled groundwater flow and heat transfer based on automatic differentiation: sensitivity calculation, verification, and synthetic examples*, Geophys. J. Int., 167, 453-466.
Contact	• Christoph Clauser (cclauser@eonerc.rwth-aachen.de) • Henrik Büsing (hbuesing@eonerc.rwth-aachen.de)

Performance metrics

Code team:

- Rene Halver (FZJ) for WP1
- Johanna Bruckmann (RWTH Aachen) for WP4

Benchmark case characteristics:

Performance report

According to Table 15, for the presented test case of a Monte Carlo based simulation Shemat used no peer-to-peer communication. The computations for this simulations consist of independent Monte Carlo realisations, that are computed on the individual MPI ranks and their results are

	Metric name	Standard compiler flags	Modified compiler flags
Golbal	Total Time (s)	75	74
	Time IO (s)	0.24	0.30
	Time MPI (s)	8.93	7.47
	Memory vs Compute Bound	1.04	0.97
	Load Imbalance (%)	7.54	6.58
IO	IO Volume (MB)	200.20	200.19
	Calls (nb)	99636	99635
	Throughput (MB/s)	821.35	670.25
	Individual IO Access (kB)	3.41	3.41
MPI	P2P Calls (nb)	0	0
	P2P Calls (s)	0.00	0.00
	P2P Calls Message Size (kB)	0	0
	Collective Calls (nb)	0	0
	Collective Calls (s)	0.00	0.00
	Coll. Calls Message Size (kB)	0	0
	Synchro / Wait MPI (s)	8.85	7.41
	Ratio Synchro / Wait MPI (%)	88.07	89.87
Node	Time OpenMP (s)	119.88	117.75
	Ratio OpenMP (s)	N.A.	N.A.
	Synchro / Wait OpenMP (s)	23.19	24.54
	Ratio Synchro / Wait OpenMP (%)	19.34	20.84
Mem	Memory Footprint	179188kB	177132kB
	Cache Usage Intensity	0.99	0.99
Core	IPC	0.73	0.69
	Runtime without vectorisation (s)	71	69
	Vectorisation efficiency	0.95	0.93
	Runtime without FMA (s)	71	73
	FMA efficiency	0.95	0.99

Table 15: Performance metrics for Shemat on the JURECA HPC system concerning compiler flags

collected afterwards by global collective operations. As a first action to improve the performance of Shemat, the hard coded compiler flags, that optimized the generated code for older hardware were replaced and the analysis run again. The results can be seen in the right column of Table 15.

During the implementation several workarounds for problems with the Scalasca instrumentation had to be implemented. As Shemat is written in Fortran, there were several small problems when the OMP directives were instrumented and two scripts had to be written to preprocess the sources. Also the newest Darshan version (3.1.3) had to be used in order to receive the results of the I/O performance.

In order to further improve the performance it is suggested to:

1. further investigate the hard-coded compiler flags, to analysis the impact of changes on the performance
2. investigate the possible improvements of the parallelisation of the Monte Carlo scheme
3. analysis of load balancing

A.10 SolarNowcast

Code ID card

Code name	SolarNowcast
Scientific domain	Irradiation forecast for predicting of photovoltaic power generation
Description	SolarNowcast is a nowcasting system based on two steps: first, estimation of motion from fisheye lens webcams data (MotionEstimation); second, irradiation forecasting from the obtained motion field at one hour horizon (Forecast).
Languages	C++, Bash scripting (15k lines)
Library dependencies	Inrimage, BFGS, OpenMP is in project.
Programing models	No parallelization
Platforms	Intel Xeon E5-2650 2.0 Ghz
Scalability results	No scalability results.
Typical production run	For now, a typical run takes a few minutes for images acquired every 5 minutes. The foreseen use of the program will rely on acquisition obtained every 10 seconds. Parallelization should allow real time processing.
Input / Output requirement	• Actual Size: 25MB • Foreseen size : 1To with full covariance matrices • Actual post-processing output : 25MB • Foreseen post-processing output : 25MB
Main bottleneck	CPU, memory access and capacity for full covariance matrices, L-BFGS minimizer, automatic generation of the adjoint by Tapenade (Inria software).
Relevant kernel algorithms	• Forward run of the model • Backward run of the adjoint • Energy minimization via conjugate gradient (L-BFGS)
Software licence	Inria proprietary software
Application references	Y. Lepoittevin, I. Herlin, "Modeling high rainfall regions for flash flood nowcasting", International Workshop on the Analysis of Multitemporal Remote Sensing Images, 2015. D. Béréziat, I. Herlin, " Solving ill-posed Image Processing problems using Data Assimilation", Numerical Algorithms, 2011, 56.
Contact	• Isabelle.Herlin@inria.fr

Performance metrics

Code team:

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

Benchmark characteristics:

Domain size	501*501
Number of timesteps	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

	Metric name	July 2016 (1/8/24 threads)	October 2016 (1/8/24 threads)
Golbal	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

Table 16: Performance metrics for Nowcasting Forecast module on the JURECA HPC system

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 Motion-Estimation, 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 hardwares, 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 unsignificant compared to execution times from 1 to 24 threads. The output frequency is every 10 timesteps (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 multithreaded. Forecast, on the other hand, uses multithreading, with weak performances, as it quickly falls under 50% efficiency, over 8 threads [Table 17]. Further investigation (VTune) shows that it is both due to Amdahl's law and unefficient multithreading.

Table 17: 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 timestep'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 parallelization : 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 timestep.
- Memsets and memcopies become significant in the distributed behavior.

A.11 Telemac

Code ID card

Code name	Telemac-Mascaret
Scientific domain	Fluid Dynamics
Description	The telemac-Mascaret system models free surface hydrodynamics for coastal and fluvial application. As such, several codes are part of the system to model hydrodynamics in 1,2 or 3 dimensions, sediment transport, or wave propagations.
Languages	Fortran77-Fortran2003 (300k lines), python 2.7
Library dependencies	MPI, Python, matplotlib, numpy, scipy
Programing models	MPI
Platforms	• Too dependent
Scalability results	It has been ported on X86 architectures, scaling results are good up to 4096 cores with 4 million elements.
Typical production run	24h on 64 - 512 cores
Input / Output requirement	• Size: X GB / 24h run • Single post-processing output: Y MB • Single restart output: Z MB
Relevant kernel algorithms	• Conjugate gradient • Conjugate residual • Conjugate gradient on normal equation • Minimum error • Squared conjugate gradient (not available) • BICGSTAB (biconjugate stabilized gradient) • GMRES (Generalised Minimum RESidual)Fully implicit Coat's type formulation
Software licence	GPL and LGPL licences
Application references	
Contact	• Antoine Joly (antoine.joly@edf.fr) • Jacques Fontaine (jacques.fontaine@edf.fr) • Chi-Tuan Pham (chi-tuan.pham@edf.fr) • Riad Ata (riad.ata@edf.fr) • Yoann Audoin (yoann.audoin@edf.fr)

Performance metrics

Code team:

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

Case1 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

	Metric name	original	after IO fix
Global	Total Time (s)	857	701
	Time IO (s)	107	1.99
	Time MPI (s)	178	173
	Memory vs Compute Bound	1.26	1.44
	Load Imbalance (%)	20	23
IO	IO Volume (MB)	9972.35	9932.84
	Calls (nb)	584 M	422 K
	Throughput (MB/s)	92.69	4996.97
	Individual IO Access (kB)	0.02	74.15
MPI	P2P Calls (nb)	1390838	1390838
	P2P Calls (s)	13.98	13.39
	P2P Calls Message Size (kB)	1	1
	Collective Calls (nb)	306873	306873
	Collective Calls (s)	162.58	158.78
	Coll. Calls Message Size (kB)	59	59
	Synchro / Wait MPI (s)	148.34	143.03
	Ratio Synchro / Wait MPI (%)	82.73	82.20
Mem	Memory Footprint	369MB	365MB
	L3 Cache Usage Intensity (%)	78	78
Core	IPC	0.97	0.96
	Runtime without vectorisation (s)	850	727
	Vectorisation efficiency	1	1.04
	Runtime without FMA (s)	815	702
	FMA efficiency	0.95	1.00

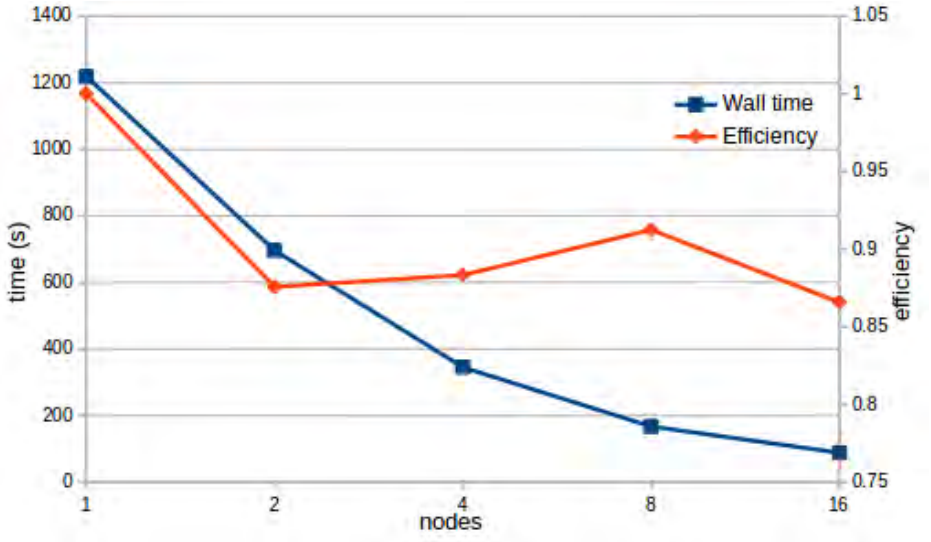
Table 18: Performance metrics for telemac2d-tomawac-sysphe coupling on the JURECA HPC system

Performance report

We have long discussed on the definition of an interesting use case for the performance analysis, considering the large range of applications covered by TELEMAC-Mascaret. We tested several cases (bump 3D which is a small and simple dam break case, Malpasset 3D dam break with a larger and more complex mesh,...) that showed very different behaviors. We finally settled for a case of study that is a 3 way multi-physics coupling for coastal simulation. It consists of a 2D free surface fluid dynamics model coupled with a wave propagation model and a sediment transport model. Its implementation involves the following modules :

- TELEMAC2D : solves the Saint-Venant equations for H,U,V (the depth and 2 velocity components) using the finite-element or finite-volume method. It is also the base of the coupling that calls the two other modules.
- TOMAWAC : simulates wave propagation in coastal areas. It solves simplified equations for spectro-angular density of wave action using a finite elements type scheme involving a method of chatacteristics.
- SISYPHE : simulates sediment transport and bed evolution.

Figure 11: Telemac - Strong scaling from 24 to 384 cores (1 to 16 nodes on Jureca)



The volume of the calculations (degrees of freedom per process) was also discussed, but there was no particular preference by the team, as the choice is user-specific. We finally decided to choose the biggest mesh provided for this case, and a number of cores that show a substantial work load and memory usage per core, based on some strong scaling tests 11.

The full EoCoE performance evaluation procedure was applied on the chosen benchmark, on 2 full JURECA nodes (48 processes) resulting in the table 18. The analysis is completed with a finer look on the Scalasca/PAPI/vampir profiles and traces produced on full or partly used (scattered binding) sockets, a VTune hotspot analysis, and strong scalability tests. I assume that the EoCoE performance evaluation procedure is described in a dedicated chapter, so no need for me to explain the definition of the reference, scatter, compact, no-vec or no-fma modes.

(1) IO fix - The first point of focus here will be the IO : On the original results on Jureca, displayed in table 18, We noticed a big bottleneck on one of the reading routines : Whether on a small or a big mesh in the three way coupling case, nearly 2 minutes were spent in reading velocities in a file, by the routine bord_tide_tpxo, a boundary condition routine for the tidal harmonic model. The solution to this bottleneck was to read the values in file into a single and contiguous complex array, instead of storing the values alterning between two arrays : one for real parts, and one for imaginary parts. After this IO fix, the resulting IO time was brought down to a couple of seconds. This improvement, however, is not confirmed yet on the code holders computers, where the execution stays slow.

The following analysis focuses on the measurements after the IO fix.

(2) Execution time and top-down profile - the test case initially ran in 857 seconds , it was brought down to 702 seconds after the IO fix.

From top down, 98% of the total time is spent in the TOMAWAC module, and spreads in the next level as follows :

- Prepro (6%) : this part prepares the propagation (assending of characteristic curves).
- Propa (33%) : the propagation part interpolates the values at the bottom of the characteristics.
- Semimp (59%) : this part solves the integration step of the source term using a scheme with variable degree of implication.

In the lowest level, the time is spent as follows :

- in Prepro : the time is mainly spent in the procedure streamline.schar41_per_4d, that contains unbalanced calculations, and the resulting waiting in collective reduction operations (p_max MPI_Allreduce).
- in Propa : the time is mainly spent in streamline.bief_interp, MPI_Allreduce reductions and MPI_Alltoall(v) collective communications. All show big unbalance.
- in Semimp : this part is purely computational. Time is mainly spent in totnrj (4% of total run), qwind1 (4%), qnlin1 (22%), qbreak (6%) and in the semimp routine (14% exclusive time).

(3) Node level performance and memory-bound behavior - The first indicator of a memory-bound behavior is the scatter vs compact test (table 18, mem vs cmp line), where the scatter mode runs 1.4 times faster than the compact mode thanks to a bigger memory bandwidth and a bigger L3 cache.

When analyzing the **L3 cache usage intensity** measured using Score-p combined with PAPI, we notice no difference between the compact and scatter modes (77% L3 cache hit rate), even though the scatter mode has twice as much cache for the same amount of data. This point might be puzzling knowing the previous result, but we have the right to question its accuracy.

On the other hand, the vectorisation analysis through Intel's opt-reports show that most of the costliest loops have a contiguous memory access, and a potential vectorization speedup of over 3 times. It is confirmed by Intel Advisor that estimates that 60% of the time is spent in vectorizable loops, and predicts a vectorization gain of over 2.4 times for the whole execution (figure 12. But it doesn't translate into facts, as the vectorization results in a gain of only 1.03 times on the reference benchmark, and merely above for the scatter run (1.08 times).

A deeper analysis using Advisor allows to establish a roofline model, showing loop level performance in relation to hardware limitations, including memory bandwidth and computational peaks. The figure 13 displays the result on 2 nodes, 24 processes per node, and figure 14 the result on 2 nodes, 1 process per node. The red dots represent the costliest loops (greater than 4% of the total run), the yellow spots represent the average loops, while the green dots represent loops that take less than 1% of the total run time. These results are in line with the mem. vs cmp., and no-vec no-fma results, showing a very memory-bound behavior, due to a too low arithmetic intensity (i.e. floating point operations per loaded byte).

(4) MPI performance and load balancing - A lot of time is spent in MPI communications, mainly due to load unbalance in calculation parts, as 22% of total CPU time is spent waiting in explicit or implicit MPI barriers.

The unbalance appears in different routines, at different stages of the simulation (cf. table 19). The most significant being streamline.bief_interp (figure 15) and streamline.schar41_per_4d (figure 16). The reason of this load unbalance could be the number of characteristic curves going in and out of each domain.

We will finally note that, unbalance put aside, a significant time is spent in collective operations : reductions and all to all communications.

Figure 12: Advisor survey, on 2 nodes, 24 task per node, summarizing the vectorization efficiency of the code

Program metrics

Elapsed Time	691,53s		
Vector Instruction Set	AVX, AVX2	Number of CPU Threads	1
Total GFLOP Count	654,36	Total GFLOPS	0,95
Total Arithmetic Intensity	0,11		

Loop metrics

Metrics	Total	
Total CPU time	688,60s	100,0%
Time in **113** vectorized loops	406,28s	59,0%
Time in scalar code	282,32s	41,0%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	3,38x	85%
Program Approximate Gain	2,40x	

Top time-consuming loops

Loop	Self Time	Total Time	Trip Counts
[loop in bief_interp at streamline.f:7910]	79,752s	79,752s	66096
[loop in qnlin1 at qnlin1.f:287]	58,465s	58,465s	459
[loop in qbrek1 at qbrek1.f:126]	44,199s	44,199s	114; 3
[loop in semimp at semimp.f:1062]	37,869s	37,869s	103275
[loop in totnrj at totnrj.f:83]	34,277s	34,277s	459

Figure 13: Roofline model (Advisor), on 2 nodes, 24 task per node, showing loop level performance in relation to hardware limitations (memory bandwidth and computational peaks)

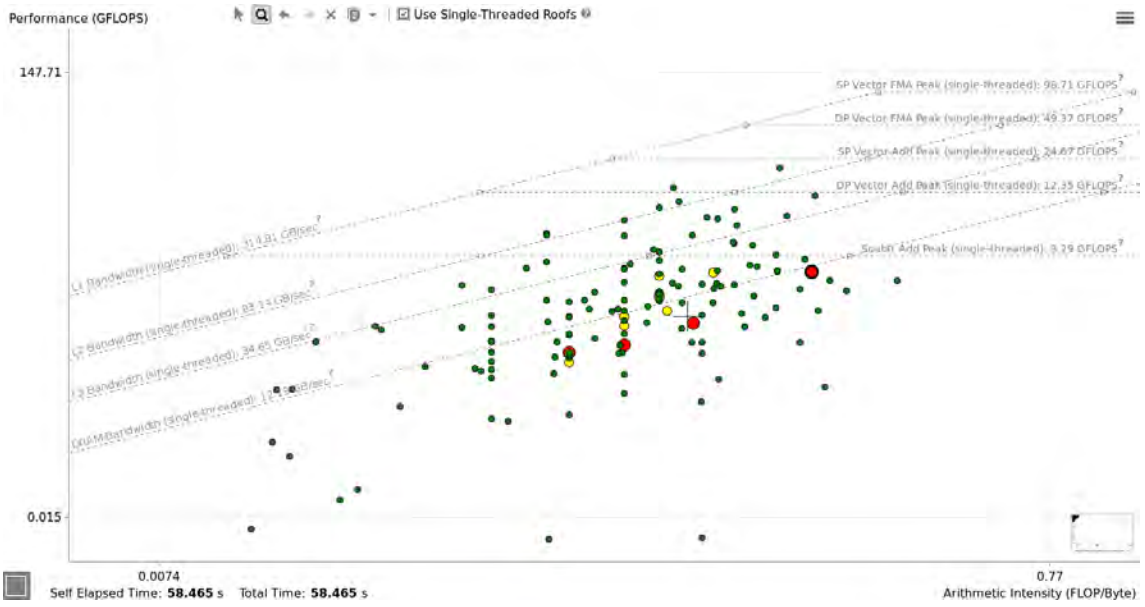
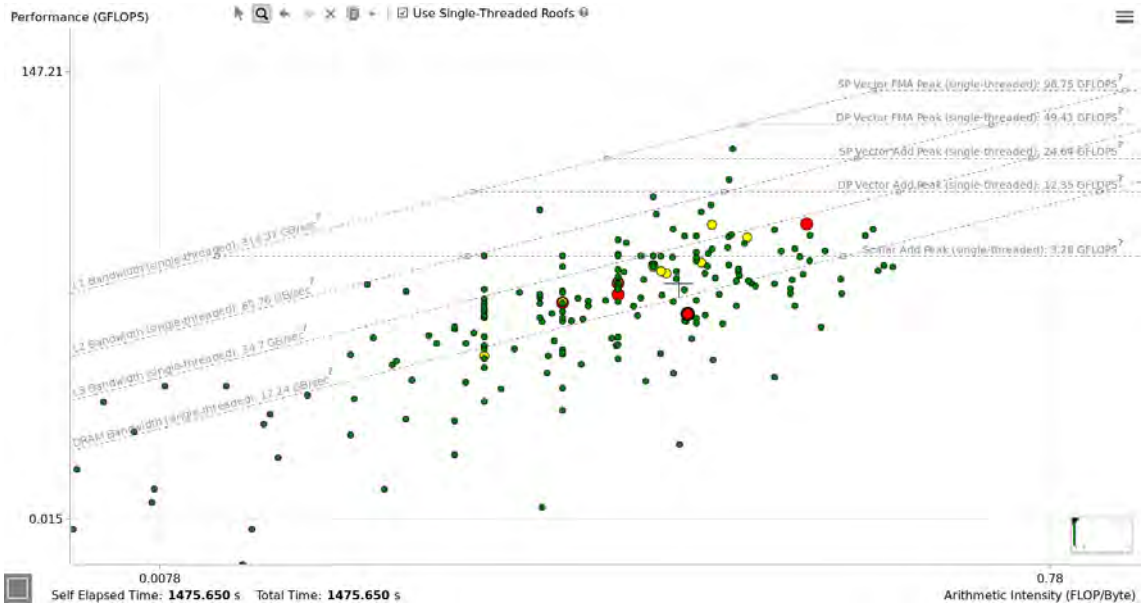


Figure 14: Roofline model (Advisor), on 2 nodes, 1 task per node, showing loop level performance in relation to hardware limitations (memory bandwidth and computational peaks)



Routine	CPU time (s)	load unbalance	mem vs cmp	vec speedup	L3 hit rate
total	33.6K	22%	1.46	1.04	78%
qnlin1	7600	7 %	1.76	1.01	88%
semimp	4700	6 %	1.88	1	79%
streamline.bief_interp	3100	34%	1.24	1	54%
qbreak1	2120	10%	1.96	1.06	79%
totnrj	1490	10%	1.91	0.98	65%
qwind1	1260	9 %	1.16	1	78%
streamline.schar41_per_4d	850	46%	1	1	84%
qfrot1	750	7 %	1.87	1.01	78%
qmount1	680	17%	1.88	1.03	79%
propa	575	19%	1.3	1	75%
streamline.post_interp	370	31%	1.28	1	67%
fremoy	360	20%	1.8	1	
MPI_Allreduce	6000	32%	1.18		
MPI_Alltoallv	1060	76%	0.9		
MPI_Waitall	555	76%	0.95		
MPI_Alltoall	555	39%	1.23		

Table 19: Hotspots profiling - with data on load balance and node level performances from Score-p, Scalasca, PAPI

Figure 15: Telemac - Scalasca trace visualization with Vampir - focus on streamline.bief_interp call showing unbalance

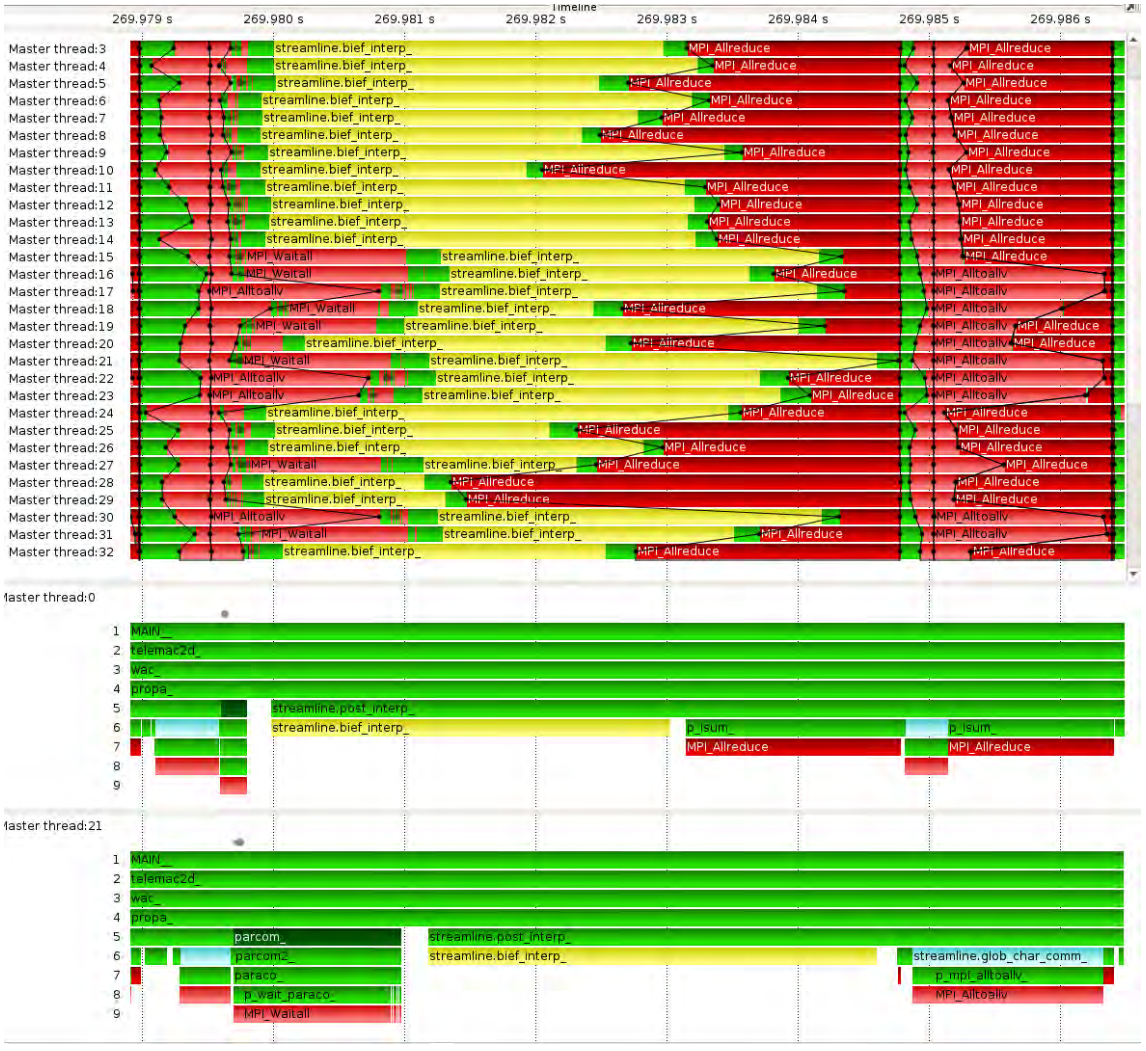
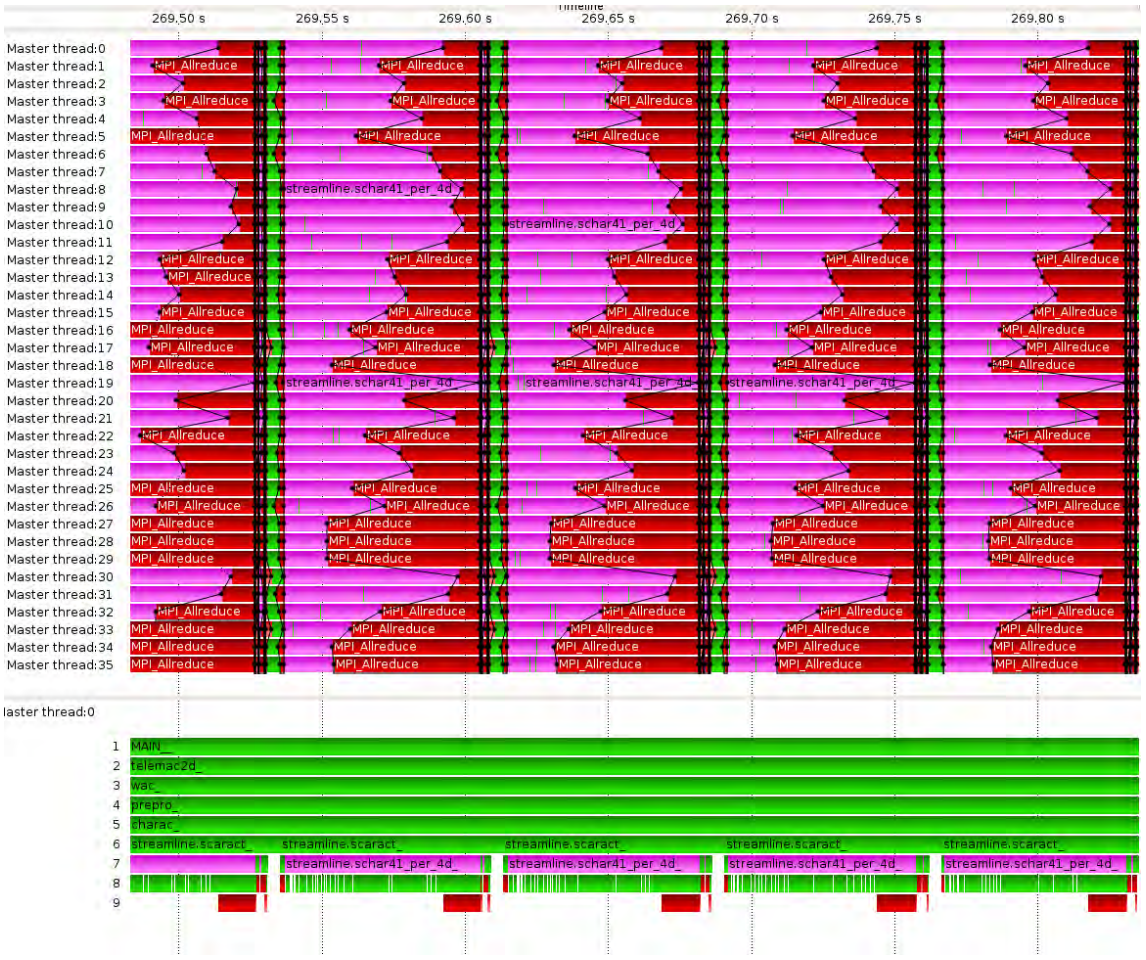


Figure 16: Telemac - Scalasca trace visualization with Vampir - focus on streamline.schar41_per_4d calls showing unbalance



A.12 Tokam3X

Code ID card

Code name	TOKAM3X
Scientific domain	WP5: Fusion4Energy
Description	TOKAM3X is a 3D fluid turbulence code for the edge plasma of tokamaks (experimental magnetic fusion reactors). It aims at tackling 2 of the main issues related with fusion research: 1- plasma confinement understanding in particular through the physics of edge improved confinement modes; 2- power and particle exhaust while insuring plasma conditions compatible with plasma facing materials.
Languages	Fortran90
Library dependencies	MPI, OpenMP, HDF5, PASTIX and/or MUMPS
Programing models	MPI, OpenMP
Platforms	• CCAMU (mesocenter of Aix-Marseille University) (600kCPUh in 2015) • French Tier1 Curie (250kCPUh in 2016) • French Tier1 Occigen (750kCPUh in 2016) • EUROFUSION Tier0 Marconi (2MCPUh in 2016)
Scalability results	It has been ported on X86 architectures, scaling results are good up to 500 cores. Extensive profiling is still to be performed.
Typical production run	40-120h on 100 - 500 cores
Input / Output requirement	• Size: 10-20 GB / 24h run • Single post-processing output: 10GB • Single restart output: 50MB
Application references	P. Tamain et al., Plasma Phys. Control. Fusion 57, 054014 (2015). P. Tamain et al., J. Comput. Phys 321, 606-623 (2016).
Contact	• Patrick Tamain (patrick.tamain@cea.fr) • Eric Serre (eric.serre@univ-amu.fr)

Performance metrics

Code team:

- Patrick Tamain (IRFM, CEA CADARACHE) for WP5
- Guillaume Latu (IRFM, CEA CADARACHE) for WP5
- L. Giraud (Institut) for WP1
- Mathieu Lobet (Maison de la Simulation, CEA Saclay) for WP1

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

First Case characteristics: 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

Second Case characteristics: 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

Third Case characteristics: 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 table 20. They have been obtained using the large case. Tokam3X used 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 and Intel Advisor. For Intel tools, a very small case has been used with a 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 (parallelized with OpenMP) uses the library Pastix parallelized with Pthreads. Pthreads is not considered during the tool analyses.

Performance report

Global performance:

The code is globally compute-bound and not DRAM memory-bound (not meaning that there is no cache issues). The same conclusion is given by Allinea. 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. 20 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.

MPI performance:

The code does not suffer from MPI communication issues since MPI communications con-

	Metric name	Code state Workshop Barcelone April 2017
Global	Total Time (s)	444
	Time IO (s)	0.06
	Time MPI (s)	2.99
	Memory vs Compute Bound	1.02
	Load Imbalance (%)	89.51
IO	IO Volume (MB)	102.76
	Calls (nb)	521
	Throughput (MB/s)	1714.77
	Individual IO Access (kB)	94.98
MPI	P2P Calls (nb)	4296
	P2P Calls (s)	0.97
	P2P Calls Message Size (kB)	14
	Collective Calls (nb)	1116
	Collective Calls (s)	0.94
	Coll. Calls Message Size (kB)	28
	Synchro / Wait MPI (s)	1.64
	Ratio Synchro / Wait MPI (%)	4.32
Node	Time OpenMP (s)	58.18
	Ratio OpenMP (%)	10.96
	Synchro / Wait OpenMP (s)	0.87
	Ratio Synchro / Wait OpenMP (%)	5.29
Mem	Memory Footprint	N.A.
	Cache Usage Intensity	0.93
Core	IPC	0.68
	Runtime without vectorisation (s)	464
	Vectorisation efficiency	1.05
	Runtime without FMA (s)	359
	FMA efficiency	0.81

Table 20: Performance metrics from JUBE for Tokam3X on the JURECA HPC system performed after the EoCoE workshop of May 4th 2017. The metrics have been obtained with the medium-size case on 1 nodes.

stitute a small fraction of the simulation time ($< 1\%$) as shown in Tab. 20. This information is validated by Allinea that give a MPI fraction of 0.1 %. Although the fraction is small, MPI synchronization and wait fraction time over the total MPI time represents 54 %. Half of the time is spent in point-to-point calls and half in collective ones. Surprisingly, Allinea announces 100 % of collective calls which is obviously a bug of the profiling tool.

OpenMP performance:

OpenMP represents a small fraction of the overall time, around 11 % according to both Jube report and Allinea reports.

The thread parallelization fraction may be artificially low because Pthreads used in Pastix is not taken into account.

Allinea gives a computation fraction of 33 % for a synchronization time of 67 % in OpenMP. This may be due to the overly fine-grained parallelism with a high number of successive OpenMP region frequently opened and closed.

Vectorization performance:

The code is not vectorized (vectorized instructions represent less than 1 %) due to the loop complexity and structures. This is shown by Tab. 20 and confirmed by Intel Advisor and Allinea

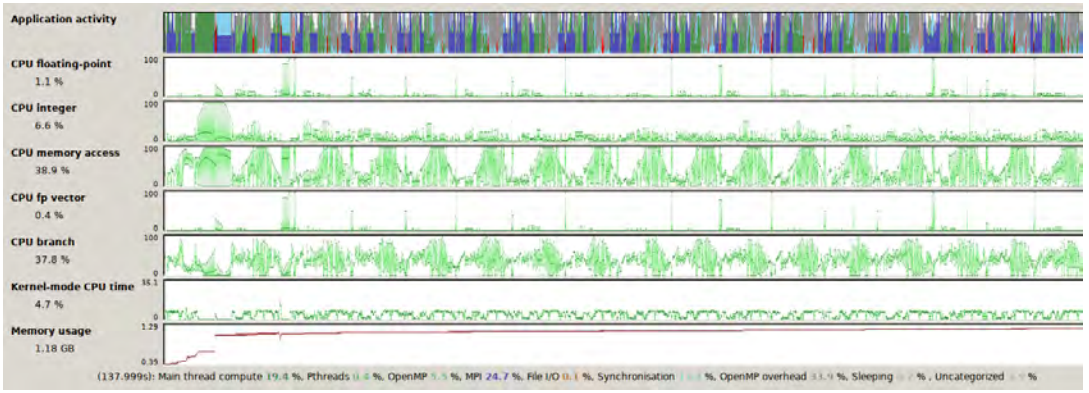


Figure 17: Application activity time line given by Allinea MAP using the medium-size test case.

MAP (see Fig. 17). According to the latter one, CPU instructions are dominated by integer operations (7 %), branching (38 %) and data load/store (39 %).

The code does not benefit from Fuse Multiply Add instructions (FMA).

IOs:

The IO volume is small and the time spent in IOs seems small in comparison to the simulation time (0.1 % according to Allinea, much less according to Darshan).

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

Conclusion:

Vectorization and FMA are the major bottlenecks of Tokam3X. They represent the main potential of speedup by refactoring the most time-consuming OpenMP loop.

Tokam3X efficiency enhancement is impaired by the use of external libraries such as Pastix that represents a large fraction of the total simulation time. Tokam3X efficiency therefore depends on future optimization in these libraries particularly for Many-Integrated Core Architectures.

A.13 PARCOMB

Code ID card

Code name	PARCOMB3D v. 3.0
Scientific domain	Computational Fluid Dynamics, Turbulent Combustion
Description	PARCOMB3D is a structured-grid numerical code for solving fluid flow and combustion problems resolved in time. The solver uses a 6th order explicit central differences scheme in space and a 4th order explicit Runge-Kutta integration in time. The chemical mechanisms used are detailed (e.g. for H2) or reduced (for fuels like CH4 or C3H8), containing typically 9 to 30 chemical species. The multicomponent gas mixture is described in detail and the complexity of the algorithm for the properties of the gas-mixture is O(N2), where N is the number of chemical species. PARCOMB is a ...
Languages	FORTRAN 90 and FORTRAN 77
Library dependencies	MPI 2.0
Programing models	MPI based on domain decomposition
Platforms	• PRACE Tier0 X (HLRS-CRAY in 2014) • German Tier1 Y (ForHLR II, 200 000h, Karlsruhe in 2017)
Scalability results	It has been ported on X86 architectures, scaling results are good up to 2000 #cores cores.
Typical production run	72 hours on 600 - 2000 cores
Input / Output requirement	approx. 100 Gb per run • Size: 100 GB / 24h run • Single post-processing output: 100000 MB • Single restart output: 10000 MB
Application references	Denev, J. A. and H. Bockhorn 2013 'DNS of Lean Premixed Flames', In: High Performance Computing in Science and Engineering '12, Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2013, Springer Verlag Berlin Heidelberg, W.E Nagel, D.B.Kroener and M. Resh (Eds.), pp. 245-258
Contact	• Jordan Denev (jordan.denev@kit.edu)

Performance metrics

Code team:

- Jordan Denev (KIT) for code developer
- Thomas Breuer (JSC) for WP1

Case1 characteristics: The run duration is approx. 1 Minute on 20 cores (1 node). The problem size is chosen to be able to run also on a single core. It is a three-dimensional test case simulating a turbulent propane-air premixed flame. As input it reads one-dimensional results (about 38 very small files in ASCII-format) and writes out 76 result-files (about 1Mb each). It also writes 2 files in tecplot format (commercial visualization tool) with a size of 22 MB (each) using a precompiled tecplot-library. The test case uses propane as a fuel which contains 28 species and 143 elementary

reactions; this is a relatively large chemical mechanism. Therefore, for this case, it is expected that computation of the chemical part and of the mixture transport properties will consume the most CPU-time.

Domain size	150 x 30 x 30 regular grid
Resources	1 node on ForHLR2 at KIT, Germany (20 cores)
I/O details	Checkpoint written every 5 steps instead of 5000 ⇒ much larger than production
Type of run	development run

	Metric name	metrics_4.json
Global	Total Time (s)	50
	Time IO (s)	N.A.
	Time MPI (s)	5.72
	Memory vs Compute Bound	N.A.
	Load Imbalance (%)	N.A.
IO	IO Volume (MB)	N.A.
	Calls (nb)	N.A.
	Throughput (MB/s)	N.A.
	Individual IO Access (kB)	N.A.
MPI	P2P Calls (nb)	653
	P2P Calls (s)	2.77
	P2P Calls Message Size (kB)	403
	Collective Calls (nb)	354
	Collective Calls (s)	0.10
	Coll. Calls Message Size (kB)	51
	Synchro / Wait MPI (s)	3.19
	Ratio Synchro / Wait MPI (%)	63.13
Node	Time OpenMP (s)	N.A.
	Ratio OpenMP (%)	N.A.
	Synchro / Wait OpenMP (s)	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.
Mem	Memory Footprint	N.A.
	Cache Usage Intensity	N.A.
Core	IPC	N.A.
	Runtime without vectorisation (s)	N.A.
	Vectorisation efficiency	N.A.
	Runtime without FMA (s)	N.A.
	FMA efficiency	N.A.

Table 21: Performance metrics for PARCOMB on the JURECA HPC system

Performance report

The profile generated with Score-P shows that about 10% of the time for the test case is consumed in the `MPI_Barrier`. Further on, the profile showed that the main portion of the CPU-time is consumed in the following 4 user-subroutines: `transport_tab` (21.1%), `rhs_terms_3d` (15.5%), `reac` (13.7%) and `newton` (7.5%). The result for the subroutines `rhs_terms_3d`, `reac` and `newton` was expected. However, the time consumed in subroutine `transport_tab` was larger than expected. There is already a possibility in PARCOMB to call this routine only once per time step (with, practically, a very small loss of accuracy), instead of calling it at each of the four Runge-Kutta sub-steps (within one time-step). This type of reduction of the CPU-time (a trade-off between CPU-time and accuracy) should be used in future runs.

Thanks to the help of the support team, profiling of only one particular loop with Score-P was also tested successfully. This allowed obtaining a working example for future use of this

helpful metrics. It will serve for further optimization of the PARCOMB-3D code, as well as for user support of other numerical codes (mainly codes for Computational Fluid Dynamics, CFD) related to energy simulations which are used in the work of the SimLab Energy at SCC, Karlsruhe Institute of Technology.

A complete work-cycle using the JUBE workflow environment has been carried out together with the developer's team. First, some adjustments of the environmental variables have been performed, which are required for the work on the fh2-supercomputer at SCC, Karlsruhe. This allows the seamless continuation of the work with JUBE on local machines in Karlsruhe after the workshop. During this adjustment, an understanding was developed about the way JUBE is configured. Also, during this process, the sequence of the algorithm steps which are required for the work with JUBE was better understood. The work with the XML-language for configuring the simulations and setting the control parameters has been explained and the targeted example has been configured. The example contained a propane flame propagation for which different compiling options and analyses (with Scalasca) have been applied. On the particular example using the PARCOMB-3d code several tasks have been learned. Between them are tracing the status of the already completed performance measurement simulations, performing intermediate error analysis and continuation of the work. The worked example helps understanding how to handle workflows which have a considerable degree of complexity due to a large number of combinations of input parameters. A table with results from JUBE attached to the present report have been created.

The trace analysis with Cube revealed imbalanced MPI communication: the number of sent and received messages differs. This is seen also by the different amount of transfered information - slightly more information is sent than received. Cube analysis shows that this imbalance appears in the initializing subroutine `init_parcomb` which explains why a number of previous numerical tests were not always initialized properly. The problem appears most likely during communication across boundaries of the initial solution when expanding a previous one-dimensional solution to two or three dimensions. A closer look at the subroutine `init_parcomb` reveals that actually the number of operators with `send` and `recv` is not balanced. However, due to the large number of cases for initialization and the relatively large number of `send` and `recv` operators, the solution of this problem will require further investigation.

A known problem in production simulations with PARCOMB is the relatively large time for writing the results in serial mode. The trace analysis with Cube revealed also some I/O problems for the test case - relatively large MPI communication time (see also above). Thes increased communication times relate mainly to the subroutines `save_var` and `save_var_tec` in which all cores send their values to the master process. As a future improvement, it should be possible to avoid the duplication of sending the information to the master process if the program writes the same variables in two different formats.

As a conclusion, in order to improve the performance of PARCOMB, we would recommend the following roadmap:

1. Guide line 1: Reduce the calls of subroutine `transport_tab` to one - or maximum two - per one time-step in future simulations. This is guided using an input parameter called `ITE_TRANS`.

2. Guide line 2: Find out the exact reason/place for the communication imbalance in subroutine `init_parcomb` for the problem used as a test case in the present workshop. Implement a corresponding correction improving this problem.

3. Guide line 3: Decrease MPI-communication times in `save_var` and `save_var_tec` during output of results. This will be achieved through avoiding duplication of work related to information exchange with the master prozess when simultaneously more than one output-formats are active (selected by the user).

A.14 OpenFOAM

Code ID card

Code name	OpenFOAM (modified solver "ebiDNS solver")
Scientific domain	Turbulent Combustion
Description	For the direct numerical simulation of turbulent flames, we use the open-source code OpenFOAM to solve the compressible Navier-Stokes equations. The standard OpenFOAM solver "rhoReactingBuoyantFoam" has been modified in order to include detailed chemistry and molecular transport. Additional OpenFOAM classes have been written which employ routines of the open-source library Cantera for computing transport coefficients and chemical reaction rates. For each computational cell, the chemical reaction rates are integrated over the simulation time step using SUNDIALS CVODE in order to reduce the stiffness of chemical source terms and to allow larger simulation time steps.
Languages	C++
Library dependencies	MPI, Sundials CVODE
Programing models	MPI (Domain decomposition)
Platforms	• PRACE Tier0 Hazel Hen • German Tier2 ForHLR II
Scalability results	Scaling results are good up to 15,000 cores for typical cases `https://link.springer.com/chapter/10.1007%2F978-3-319-47066-5_16` (doi:10.1007/978-3-319-47066-5_16)
Typical production run	1 to 2 weeks on 1,000 – 15,000 cores (depending on case size)
Input / Output requirement	• Size: 20 – 50 GB / 24h run • Single post-processing output: 5 – 50 GB • Single restart output: 5 – 20 GB
Application references	• F. Zhang; H. Bonart; T. Zirwes; P. Habisreuther; H. Bockhorn; N. Zarzalis. Direct Numerical Simulation of Chemically Reacting Flows with the Public Domain Code OpenFOAM. In High Performance Computing in Science and Engineering 14, Nagel, Wolfgang E. and Kröner, Dietmar H. and Resch, Michael M. (ed.), Springer International Publishing, p. 221-236, (doi:10.1007/978-3-319-10810-0_16) 2015. • Zhang, Feichi; Zirwes, Thorsten; Habisreuther, Peter; Bockhorn, Henning. A DNS Analysis of the Correlation of Heat Release Rate with Chemiluminescence Emissions in Turbulent Combustion. In High Performance Computing in Science and Engineering '16, Nagel, Wolfgang E.; Kröner, Dietmar H.; Resch, Michael M. (ed.), Springer International Publishing, p. 229-243, (doi:10.1007/978-3-319-47066-5_16) 2017. • Zirwes, Thorsten; Zhang, Feichi; Denev, Jordan A.; Habisreuther, Peter; Bockhorn, Henning. Automated Code Generation for Maximizing Performance of Detailed Chemistry in DNS of Turbulent Combustion. In High Performance Computing in Science and Engineering '16, Nagel, Wolfgang E.; Kröner, Dietmar H.; Resch, Michael M. (ed.), Springer International Publishing (submitted)
Contact	• Thorsten Zirwes (thorsten.zirwes@kit.edu) • Feichi Zhang (feichi.zhang@kit.edu)

Performance metrics

Code team:

- Thorsten Zirwes (Steinbuch Centre for Computing)

- T. Breuer and S. Lührs

Case1 characteristics:

The first case is a much smaller test case compared to typical runs. It represents an extreme test case for load imbalance. The computational domain is one dimensional and decomposed into 20 sub-domains. The flame is only present in the first sub-domain, so that only one process has to compute all of the flame's chemistry. This kind of load imbalance can happen in production runs if, for example, most of the computational domain is needed to compute the gas flow and the flame is only in a small part of the domain. Since computation of the flame's chemistry can take considerable amounts of computing time, this leads to load imbalances. By default, OpenFOAM decomposes the domain so that every process roughly has the same number of grid cells, regardless of the position of the flame.

Domain size	10,000 × 1 × 1 cells, block structured mesh (treated by OpenFOAM as unstructured mesh)
Resources	1 node on Jureca (20 cores)
IO details	only files to initially start the simulation are read. No output
Type of run	development run

Case2 characteristics:

The second case is a bit larger than the first one but still considerably smaller than typical runs. It is a two dimensional domain with spherical symmetry. The flame starts from the center of the sphere and burns over time through the whole domain. At each time, there is a load imbalance because the flame is always only in a small part of the domain.

Domain size	500 × 1,000 × 1 cells, block structured mesh (treated by OpenFOAM as unstructured mesh)
Resources	6 nodes on Jureca (120 cores)
IO details	only files to initially start the simulation are read. No output
Type of run	development run

Both test cases use less than 200 MB RAM per core. Typically, RAM is not a limiting factor due to the high number of CPU cores needed for the simulations. The default way OpenFOAM handles I/O is to read/write one file per process per time per conservation equation×2. In typical runs, we use 1,000 – 15,000 processes and solve 20 – 60 conservation equations. This means that at the start of the simulation, 10,000–2,000,000 files must be read (or written in order to start from a later time). Due to this, we limit I/O so that we only write out the full simulation data needed for a restart at most once a day.

Performance report

Table 22 shows the performance metrics for **Test Case1** generated with JUBE. It should be noted that recompiling OpenFOAM takes about 4 hours on 10 cores. Therefore, only the code developed directly by us is recompiled for the runs without vectorization or FMA. OpenFOAM itself is unchanged and Sundials as external dependency too.

Some remarks on the used software: OpenFOAM is used in Version 4.1 and Version 1612+, which are the latest version of the two most commonly used OpenFOAM forks.

Compiling OpenFOAM with Score-P has proven difficult. Because OpenFOAM has its own Make system (`wmake`), there are many changes needed within the OpenFOAM source code in order to compile it with Score-P. Additionally, OpenFOAM is distributed with third party libraries which need modification too. By following a guide for JURECA (https://trac.version.fz-juelich.de/vis/wiki/Software/OpenFOAM/config_ScoreP) and adapting it for our OpenFOAM version, we managed to compile OpenFOAM with ScoreP on JURECA. Full traces generated by

	Metric name	metrics_15.json
Global	Total Time (s)	10
	Time IO (s)	N.A.
	Time MPI (s)	2.61
	Memory vs Compute Bound	N.A.
	Load Imbalance (%)	16.09
IO	IO Volume (MB)	N.A.
	Calls (nb)	N.A.
	Throughput (MB/s)	N.A.
	Individual IO Access (kB)	N.A.
MPI	P2P Calls (nb)	145557
	P2P Calls (s)	2.25
	P2P Calls Message Size (kB)	0
	Collective Calls (nb)	20335
	Collective Calls (s)	0.30
	Coll. Calls Message Size (kB)	0
	Synchro / Wait MPI (s)	2.05
	Ratio Synchro / Wait MPI (%)	78.35
Node	Time OpenMP (s)	N.A.
	Ratio OpenMP (%)	N.A.
	Synchro / Wait OpenMP (s)	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.
Mem	Memory Footprint	13016kB
	Cache Usage Intensity	N.A.
Core	IPC	N.A.
	Runtime without vectorisation (s)	10.5
	Vectorisation efficiency	1.05
	Runtime without FMA (s)	10
	FMA efficiency	1.00

Table 22: Performance metrics for OpenFOAM on the JURECA HPC system

ScoreP for runs with 20 processes for a duration of 8 seconds (wall clock time) were almost 100 GB in size and increase simulation times by a factor of 100. Therefore, we used a filter file which removes all functions located in the `Foam` namespace. This effectively removes all functions from OpenFOAM and only leaves MPI functions and function from our own code. In our case, this is acceptable since most of the computational time is spent in our own routines. Traces with EXTRAE are about 2 GB for 20 processes and 8 seconds of wall clock time.

Figure 18 shows results from ScoreP for **Test Case2**. The highlighted entries show a load imbalance between the processes. The function `int_rhs_gri30` is called more frequently on some processes than others.

These load imbalances are created by the chemistry computations for the flame. Before the conservation equations for the species masses can be solved, we first have to compute the chemical reaction rate of each specie, which is the source term for the conservation equation. Computing the chemical reaction rates requires the solution of a coupled, non-linear set of ODE's. In locations in the domain where the flames burns due to very fast chemical reactions, this ODE system becomes very numerically stiff. We therefore use the external library SUNDIALS CVODE, which is a stiff ODE solver, to compute the reaction rates in each grid cell of the domain. But cells where the ODE system is very stiff require more iterations by the ODE solver, so that processes which solve sub-domains where the flame is present, need more time than processes of a sub-domain, where there is no flame or chemical reaction at all. The function `int_rhs_gri30` is written by us and computes the chemical reaction rates.

Similarly, Fig. 19 and Fig. 20 also demonstrate these load imbalances measured with ScoreP

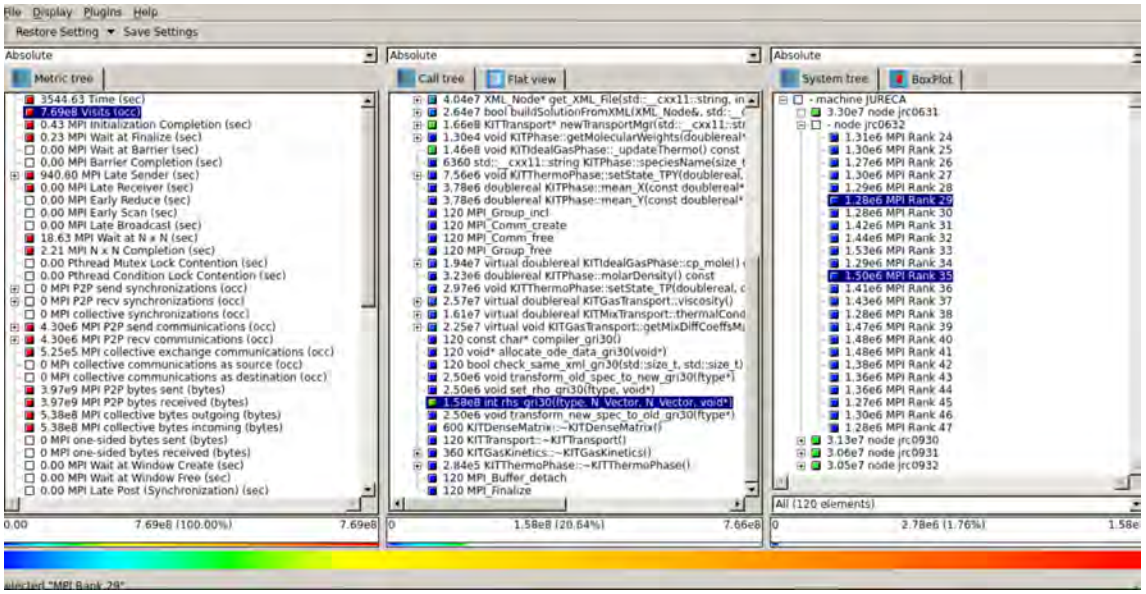


Figure 18: ScoreP results for Test Case2.

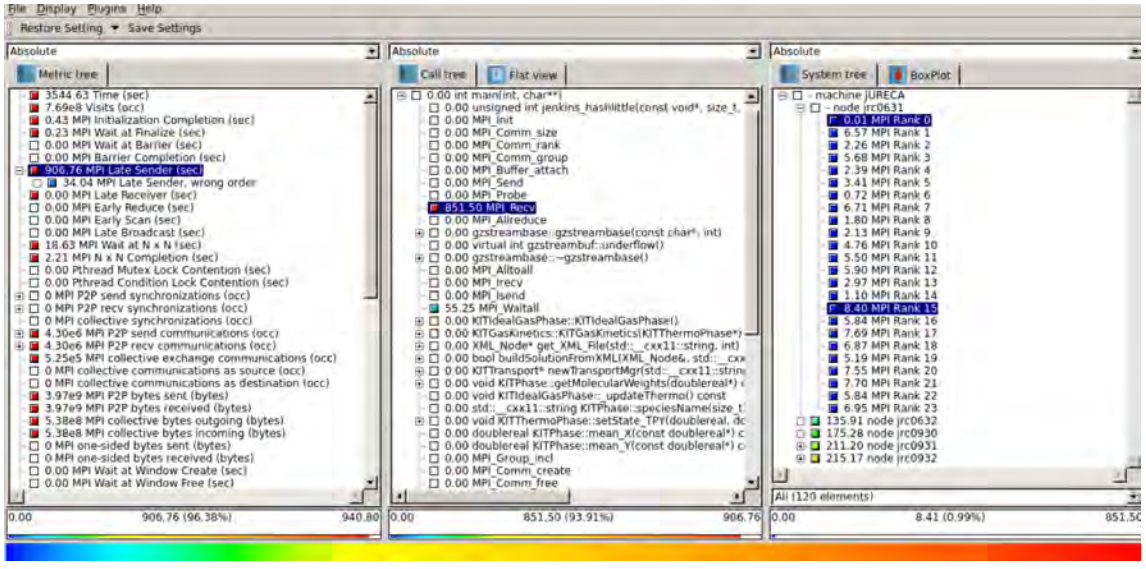


Figure 19: ScoreP/Scalasca/Cube results for Test Case2.

MPI call profile @ test_trace.chop1.prv@jrl12

	Outside MPI	MPI_Send	MPI_Recv	MPI_Isend	MPI_Irecv	MPI_Waitall	MPI_Allreduce
THREAD 1.1.1	76.45 %	0.02 %	15.40 %	0.50 %	0.63 %	3.14 %	3.87 %
THREAD 1.2.1	78.49 %	0.01 %	13.21 %	0.64 %	0.73 %	3.07 %	3.85 %
THREAD 1.3.1	86.46 %	0.01 %	4.40 %	0.50 %	0.59 %	3.70 %	4.34 %
THREAD 1.4.1	78.89 %	0.01 %	12.52 %	0.85 %	0.93 %	2.90 %	3.89 %
THREAD 1.5.1	86.86 %	0.01 %	4.89 %	0.80 %	0.88 %	2.85 %	3.71 %
THREAD 1.6.1	84.69 %	0.01 %	7.35 %	1.08 %	1.18 %	2.13 %	3.56 %
THREAD 1.7.1	89.97 %	0.01 %	1.05 %	0.51 %	0.59 %	3.69 %	4.18 %
THREAD 1.8.1	77.50 %	0.01 %	13.42 %	0.53 %	0.59 %	3.62 %	4.34 %
THREAD 1.9.1	88.01 %	0.01 %	3.45 %	0.64 %	0.76 %	3.12 %	4.01 %
THREAD 1.10.1	87.32 %	0.01 %	4.40 %	1.01 %	1.06 %	2.39 %	3.82 %
THREAD 1.11.1	82.31 %	0.01 %	9.54 %	1.09 %	1.18 %	2.16 %	3.73 %
THREAD 1.12.1	80.33 %	0.01 %	11.32 %	0.97 %	1.01 %	2.64 %	3.73 %
THREAD 1.13.1	78.81 %	0.01 %	13.39 %	0.98 %	1.11 %	2.23 %	3.47 %
THREAD 1.14.1	85.08 %	0.01 %	6.54 %	0.88 %	0.91 %	2.92 %	3.68 %
THREAD 1.15.1	89.34 %	0.01 %	2.63 %	1.04 %	1.06 %	2.39 %	3.53 %
THREAD 1.16.1	74.25 %	0.01 %	18.11 %	1.39 %	1.40 %	1.67 %	3.16 %
THREAD 1.17.1	72.95 %	0.02 %	18.66 %	1.03 %	1.06 %	2.65 %	3.63 %
THREAD 1.18.1	73.71 %	0.01 %	17.99 %	1.01 %	1.04 %	2.72 %	3.54 %
THREAD 1.19.1	73.53 %	0.01 %	18.22 %	1.01 %	1.04 %	2.61 %	3.59 %
THREAD 1.20.1	80.04 %	0.01 %	11.63 %	1.02 %	1.06 %	2.61 %	3.64 %
THREAD 1.21.1	73.70 %	0.01 %	18.38 %	1.20 %	1.24 %	2.22 %	3.26 %
THREAD 1.22.1	73.98 %	0.01 %	17.68 %	0.87 %	0.92 %	3.01 %	3.53 %
THREAD 1.23.1	74.53 %	0.01 %	17.39 %	1.02 %	1.06 %	2.46 %	3.54 %
THREAD 1.24.1	73.65 %	0.01 %	17.88 %	0.70 %	0.78 %	3.18 %	3.81 %
THREAD 1.25.1	74.75 %	0.02 %	17.39 %	1.22 %	1.20 %	2.13 %	3.31 %
THREAD 1.26.1	73.19 %	0.01 %	18.16 %	0.69 %	0.77 %	3.43 %	3.76 %
THREAD 1.27.1	72.88 %	0.01 %	18.80 %	0.90 %	0.90 %	2.88 %	3.63 %
THREAD 1.28.1	73.16 %	0.01 %	18.13 %	0.50 %	0.64 %	3.81 %	3.76 %
THREAD 1.29.1	72.77 %	0.01 %	18.76 %	1.05 %	1.04 %	2.80 %	3.57 %

Figure 20: EXTRAE/Parave results for Test Case2.

and EXTRAE.

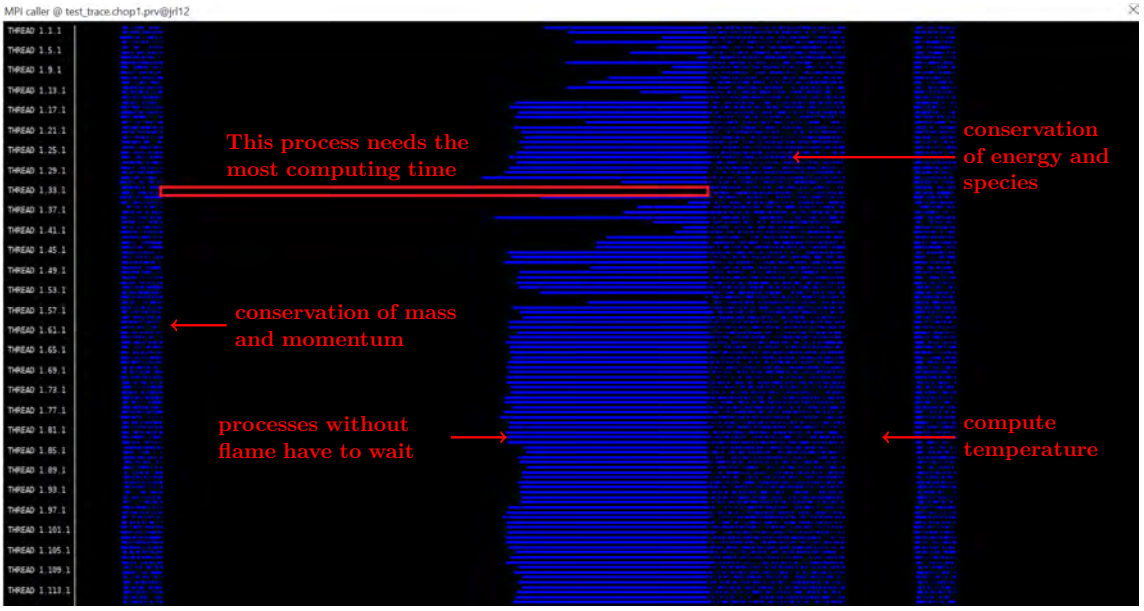


Figure 21: EXTRAE/Parave results for Test Case2.

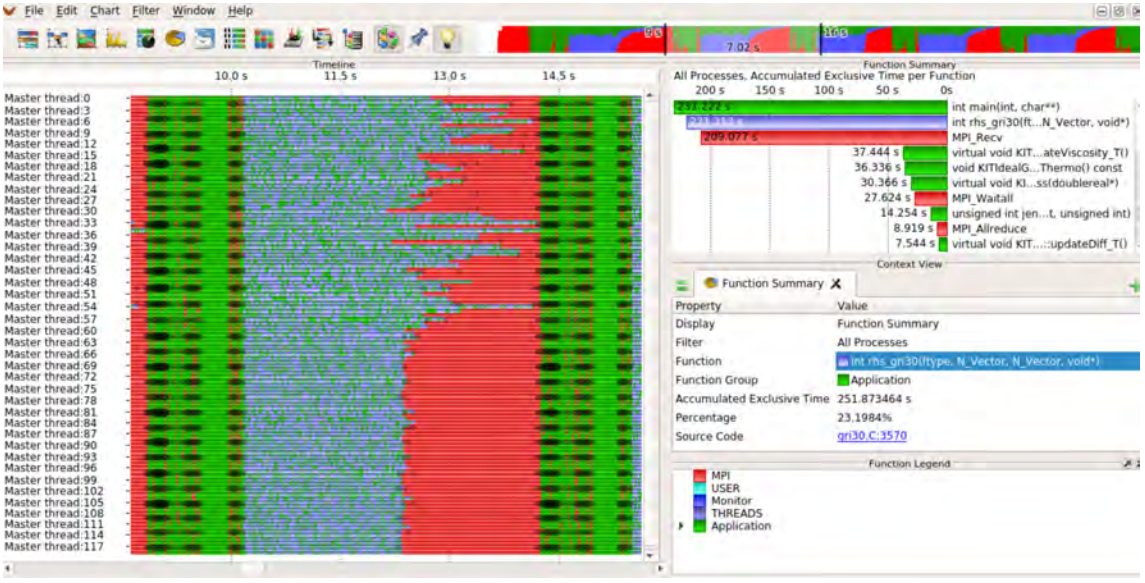


Figure 22: ScoreP/Vampir results for Test Case2.

Figure 21 shows one time step of the simulation measured with EXTRAE. MPI communication is depicted in blue. The communication for the solution of all conservation equations is distributed evenly between the processes. But the computation of the chemical reaction rates requires different amounts of time among the processes so that almost all processes have to wait before they can solve the conservation equations for energy and the chemical species masses.

The same time step is shown in Fig. 22 for the ScopreP results shown in Vampir. Depicted in red are the MPI waiting times. Green shows the useful work done for the solution of the conservation equations. The function `int_rhs_gri30` is colored in light blue. It is clear that this function, which computes the chemical reaction rates and requires most of the computational time, causes the load imbalances.

In the past, we focused optimization efforts on node level performance optimization because

most of the simulation time is spent on chemistry computations (like in `int_rhs_gri30`) and these are inherently serial because no communication is needed. You can read more details about this in the last paper listed in section A.14. The next issue to resolve will be the parallel load imbalance. The problems are:

- By default, OpenFOAM decomposes the domain so that each process has roughly the same number of grid cells, regardless where the flame is.
- This means, that the solution of all conservation equations (total mass, momentum, energy, 20–60 chemical species) takes the same time for all processes (as shown in Fig. 22)
- But the computation of chemical reaction rates, which are the source terms needed to solve the 20–60 conservation equations for the chemical species, take different times on each processor.
- This problem becomes even worse if we use adaptive mesh refining to locally resolve the flame, because processes which need longer to compute the chemical source terms per cell due to the presence of the flame will also have more cells in total compared to processes in regions with no flame.
- If the domain decomposition would be weighted by the location of the reactive zones of the flame, each process would have a different number of grid cells. This might help to make the time needed to compute the chemical reaction rates more evenly distributed but create load imbalances for the solution of the conservation equations, which only depend on the number of cells per process.
- The flame is not stationary but can move freely through the domain. This makes it hard to predict where the flame will be and might make efficient domain decomposition impossible.

Since the EoCoE workshop, we started to experiment with the tolerances for the SUNDIALS ODE solver. We relaxed the tolerances, which decreases the time discrepancy between cells with violent chemical reaction and without chemical reaction, because the maximum number of iterations (in cells with especially stiff ODE systems) is reduced. The simulations results are almost unaffected by this so that the load imbalance is still there but less severe as shown in the examples above.

A.15 MUMPS

Code ID card

Code name	MUMPS
Scientific domain	WP1: Numerical Linear Algebra
Description	MUMPS ("MUltifrontal Massively Parallel Solver") is a package for solving systems of linear equations of the form Ax = b, where A is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric, on distributed memory computers. It was developped inside a consortium started around CERFACS, INPT, inria, ENS-Lyon and Bordeaux-Univeristy.
Languages	Fortran90 with Interfaces for C, Matlab and Scilab
Library dependencies	MPI, OpenMP, Scotch, Metis, BLAS, BLACS, Scalapack
Programing models	MPI, OpenMP
Platforms	• PRACE Curie • CALMIP • Cluster Nemo CERFACS
Scalability results	It has been ported on a lot of architectures with varying number of cores on different interconnection networks. With partial differential equation in 3D problems, Mumps typically solves few tens million variables on few thousands of cores.
Typical production run	Depends on the size and structure of the sparse linear system to solve.
Input / Output requirement	idem
Main bottleneck	Memory consumption in the case of a large fill-in during elimination.
Relevant kernel algorithms	• Multi-frontal Gaussian Elimination • BLAS, SCALAPACK • Partitioning algorithms (Metis, scotch)
Software licence	CeCILL-C license
Application references	`http://mumps.enseeiht.fr/index.php?page=doc`
Contact	• Philippe Leleux (leleux@cerfacs.fr) • Mumps developers support (mumps-dev@listes.ens-lyon.fr)

Performance metrics

Code team:

- Philippe Leleux for code developer
- Yacine Ould-Rouis for WP1

This document is the result of a study conducted during the hands-on workshop on HPC benchmarking and performance analysis at Barcelona Supercomputing Centre from 24th to 27th of April 2017, see section 3. Its purpose was to evaluate the performance of our code: the solver Mumps, through runs on the supercomputer Jureca (Jülich Research on Exascale Cluster Architectures).

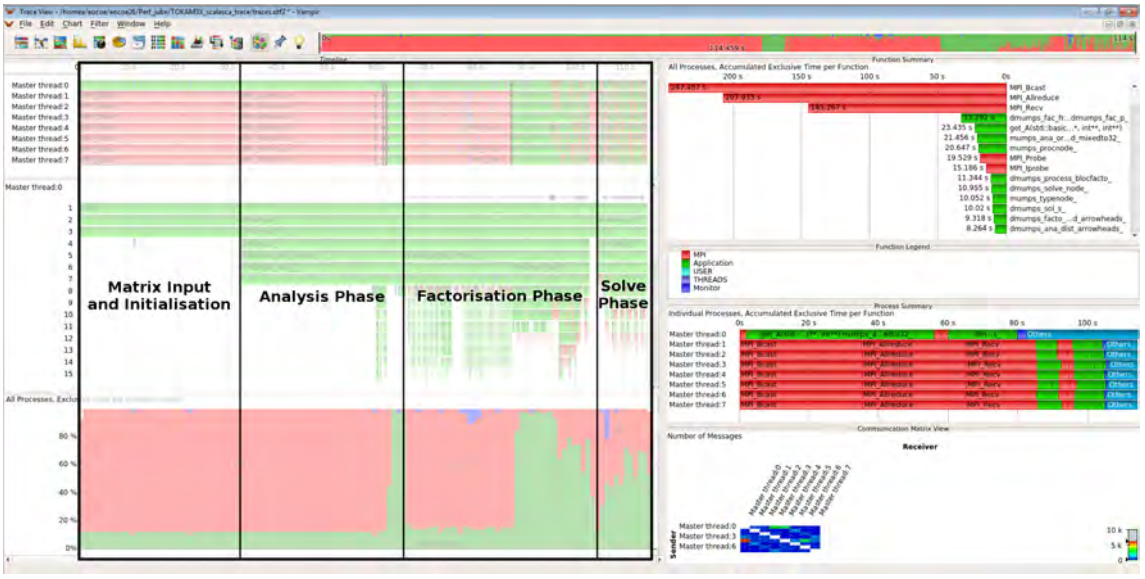


Figure 23: Screenshot from Vampir on out Test Case, matrix TOKAM3X solved on 8x1 cores, with highlighted phases of the application code.

This machine posseses 1872 nodes with 128GB of memory and 2 sockets of 12 Intel Xeon E5-2680 v3 Haswell CPUs at 2.5GHz. On this architecture, we chose to allocate complete nodes for the runs with 1 MPI process per socket maximum and its associated OpenMP threads are distributed inside the same socket. In the following "P x N cores" stands for P processes with N threads each.

MUMPS (MUltifrontal Massively Parallel direct Solver):
Mumps is a package for solving systems of linear equations of the form Ax = b, where A is a sparse matrix. The solver has an Hybrid MPI/OpenMP model based on distributed dynamic scheduling, see [1] and [2] for more details.

Our application code starts with a sequential initialization phase followed by the actual solving by the solver. MUMPS follows a multifrontal scheme, which is a direct method, composed of 3 steps (see Figure 23):

- Analysis: preprocessing of the matrix (ordering, scaling, partitioning,...) and symbolic Factorisation. From the adjacency graph, this step allows the construction of an "elimination tree", decomposing the global system in smaller interconnected parts (fronts) for the factorisation. There exist 2 versions of this phase: one sequential and one parallel, we opted for the sequential option.
- Factorisation of the input matrix: this step makes use of 2 levels of parallelism, one introduced by the tree structure and the second is at node level where large fronts are solved by several processes.
- Solve: Forward/Backward substitution.

Our application code was compiled using Intel-2017.2.174 with associated IntelMPI and MKL libraries. Also MUMPS is dependent on external efficient parallel libraries:

- ordering: METIS-5.1.0,

- MKL dense kernels: BLAS/SCALAPACK.

Case characteristics:
For this study, we used a sparse matrix in real double precision arithmetics coming from a 3D problem: TOKAM3X_mat_inertia_divertor from the CEA project TOKAM3X (matrix abbreviated as **TOKAM3X**). This project focuses on the simulation of plasma turbulence in a TOKAMAK and is included in Work Package 5 of EoCoE focused on fusion reaction energy, see [3].

Domain size	242 501 degrees of freedom with 119 non-zeros per line mean
Resources	8x1 cores on Jureca
IO details	Reading input matrix and right hand side from files during initialisation only
Type of run	development run

This case was chosen because it is quick enough to be solved within the time dedicated in this study and is representative of the actual behaviour of the solver.

	Metric name	TOKAM3X
Global	Total Time (s)	128
	Time IO (s)	N.A.
	Time MPI (s)	55.4
	Memory vs Compute Bound	1
	Load Imbalance (%)	N.A.
IO*	IO Volume (MB)	
	Calls (nb)	
	Throughput (MB/s)	
	Individual IO Access (kB)	
MPI	P2P Calls (nb)	2 181
	P2P Calls (s)	22.2
	P2P Calls Message Size (kB)	472.8
	Collective Calls (nb)	5 270
	Collective Calls (s)	32
	Coll. Calls Message Size (kB)	34.5
	Synchro / Wait MPI (s)	41.4
	Ratio Synchro / Wait MPI (%)	48.9
Node	Time OpenMP (s)	186
	Ratio OpenMP (%)	100
	Synchro / Wait OpenMP (s)	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.
Mem	Memory Footprint	16 692 280kB
	Cache Usage Intensity	N.A.
Core	IPC	N.A.
	Runtime without vectorisation** (s)	127
	Vectorisation efficiency**	0.99
	Runtime without FMA (s)**	129
	FMA efficiency**	1.01

Table 23: Performance metrics for MUMPS on the JURECA HPC system with both test cases. For details about the metrics, see 4. *IO only in initialization, not relevant for Mumps. **Metrics not representative: the solver makes very efficient use of both methods thanks to the intense use of BLAS3 in the numerical factorisation phase.

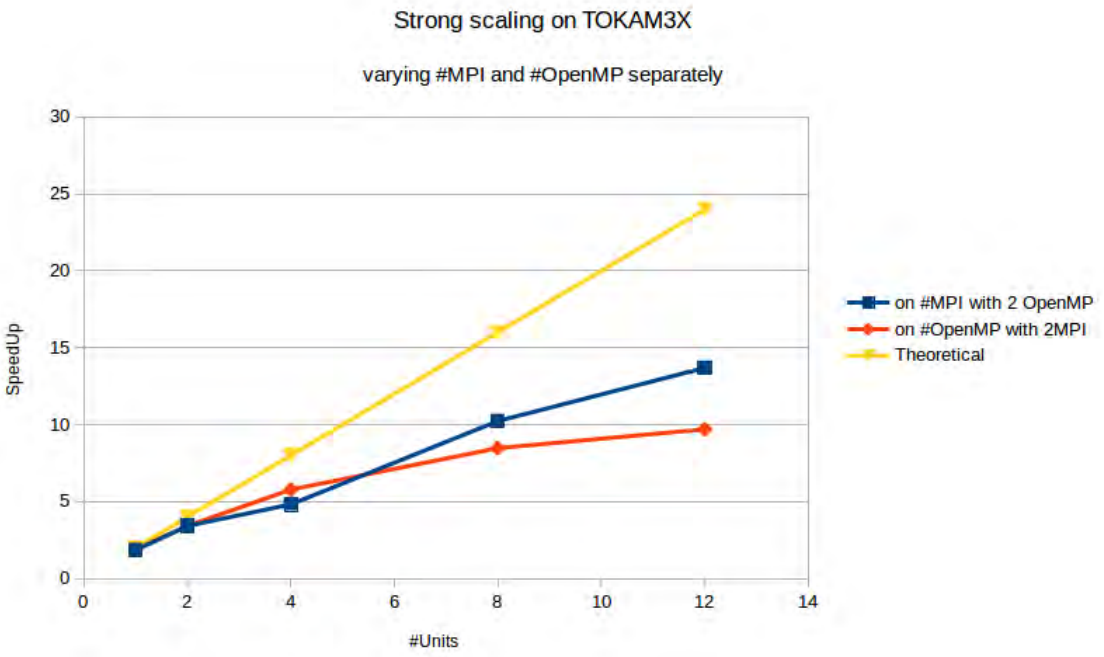


Figure 24: Speed Ups in strong scaling of MUMPS Factorisation phase on matrices TOKAM3X compared to a sequential run. Separately, we take 2 threads and vary the number of MPI processes; then we take 2 MPI and vary the number of OpenMP threads. The first point is the speedup with 2 cores: either 1x2 (blue) or 2x1 (red).

Performance report

Goals:
We present results of Strong Scaling on TOKAM3X in Figure 24, varying MPI and OpenMP separately. We observe a good scalability with this number of cores: speedups are reasonnable for a fairly small matrix. The results are slightly better when augmenting the number of MPI processes than when augmenting the number of threads. The question now is can we understand the performance? This is what performance analysis is about: trying to locate bottlenecks in the code that could be overcome, if any.

In the context of EoCoE, several tools[7] were available to us:

1. First, JUBE is a powerful benchmarking tool launching tests with different configurations automatically. It was configured to output automatically several specific metrics. JUBE calls directly other performance tools and will give us general results, most notably their overhead.

2. Through JUBE, we instrumented MUMPS and generated a trace profile of the run on TOKAM3X with the tool Score-P. A functionality of this same tool (scorep-score) outputs number of calls and time spent for each function in MUMPS. PAPI (Performance Application Programming Interface) is used internally by Score-P to collect low level performance metrics, allowing the profiling.

3. This profile, analyzed with the visualizer Vampir, will give us precise insight of the communication time along the run.

[7] http://www.vi-hps.org/

4. Finally, Scalasca will compute statistics on Communication and Computational balance with the associated functions. Those 2 last tools provide the location of regions of the code with possible improvements.

Performance Analysis:
According to EoCoE specific metrics in Table 23:

- As explained in the caption, MUMPS makes a very efficient use of both vectorisation and FMA through extensive use the BLAS dense kernel which is not shown here,
- Memory footprint is big even for this not-so-big test as expected for a direct solver.

JUBE is a powerful tool which was configured in EoCoE to automatically launch several tests, see table 24 for detailed timings on our test case. From this table, we observe that there is a big overhead due to performance tools (65% for scalasca, 35% for PAPI). According to scorep-score output on test case 1, this could be explained by very short functions being called often in MUMPS (see Table 25 for the first functions output in scorep-score). Only filtering these first 5 routines would reduce the trace size from 4GB to 1GB. This should be done before future performance analysis. Also, the number of calls of these functions looks surprising and could be checked.

mode	#tasks	#threads	Analysis(s)	Facto(s)	Solve(s)
ref	8	4	8,93	26,72	0,47
scalasca	8	4	13,11	46,68	0,85
papi	8	4	11,25	37,22	0,72
No-vec	8	4	8,96	26,51	0,47
No-fma	8	4	8,96	26,27	0,47

Table 24: MUMPS timings with matrix TOKAM3X on the JURECA HPC system. **scalasca:** run with scalasca tracing enabled; **papi:** run with PAPI counters enabled; **No-vec:** run with vectorisation disabled; **No-fma:** run with FMA (Fused Multiply-Add) disabled.

type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
USR	901,433,546	141,280,281	21.21	2.3	0.15	mumps_procnode_
USR	629,315,362	74,758,067	9.56	1.0	0.13	mumps_typenode_
USR	582,314,590	42,870,461	5.33	0.6	0.12	mumps_typesplit_
USR	580,992,490	22,345,865	3.57	0.4	0.16	std::operator—(std::_Ios_Iostate, std::_Ios_Iostate)
USR	580,992,490	22,345,865	4.24	0.5	0.19	std::operator&(std::_Ios_Iostate, std::_Ios_Iostate)

Table 25: First 5 functions appearing in scorep-score output from Test Case 1 on TOKAM3X matrix

We can now take a closer look at the trace generated by Score-P. We find that where Analysis phase is completely sequential by choice, Factorisation also presents a rather high ratio of MPI communication, that the time spent in MPI communications compared to the time in computation (see Figure 25).
This phase starts with the long region1 (about half of the Factorisation) where almost only the Master is computing, we identify this part as the data distribution of the fronts over processes done only by the Master (*dmumps_facto_send_arrowheads_/dmumps_facto_recv_arrowhd2_*). Region1 is due to the choice of centralized input matrix, which rarely occurs in actual applications.
Then, at the end of Factorisation, region2 also presents a high ratio of MPI (around 30%) where processes are intermittently computing and communicating. Region2 is followed by region3 that has the lowest ratio of computation. We identify region2 as the actual Factorisation of the fronts

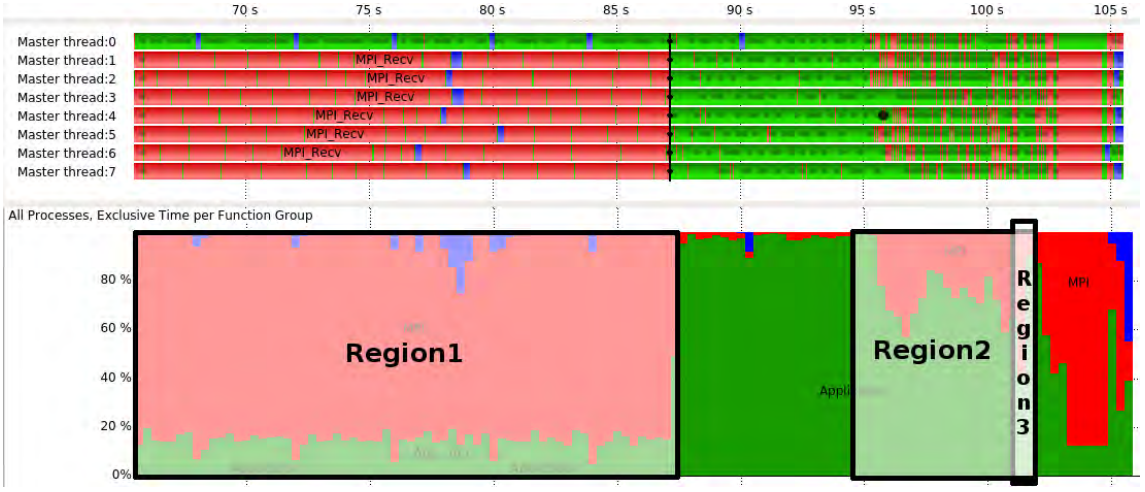


Figure 25: Screenshot from Vampir on our Test Case, matrix TOKAM3X on 8x1 cores, showing the Factorisation phase with 3 highlighted regions of high MPI communications ratio.

(*dmumps_process_bloc_facto_*). Region2 is the actual multifrontal factorisation of the matrix going through the elimination tree while region3 is the solving of the root using SCALAPACK. The latter has a high ratio of communication (around 60%) as the matrix to solve with the dense kernel is small (n=4320). Also, we noticed the whole region2 presents a surprising disparity in number of messages exchanged between processes, see Figure 26.

Finally, Scalasca confirmed our observations and pointed to specific functions with issues (timings are accumulated over processes involved in such events):

- MPI Synchronisations:
  - Late Sender (a receiver must wait for a message to arrive): *dmumps_fac_driver / dmumps_facto_recv_ar* (20.21s),
  - Wait States: *dmumps_fac_driver* (143.21s)
- Computational imbalance in:
  - *dmumps_fac_driver*: Overload (10.48s), Underload (5.53s) and Non-participant (20.61s),
  - *dmumps_fac_driver / dmumps_facto_send_arrowheads_*: Single participant (17.63s).

Conclusion:
Performance tools, ex. Vampir, were already used to improve MUMPS. The EoCoE performance analysis was still a good chance to get a new insight of the code and look for further improvements. We had some issues with the performance tools though, in particular OpenMP parallelisation was not traced.

In order to further improve MUMPS, we would recommend:

- Observe the behaviour of OpenMP parallelization,
- Check the number of calls of small functions from scorep-score; communication times in the factorization driver; Disparity in number of messages exchanged in Factorisation region2,
- Evaluate the separate use of the abundance of features in MUMPS. In particular, distributed input will be the format used in actual applications. Also Block Low Rank, which is a brand new feature would be very interesting to examine.

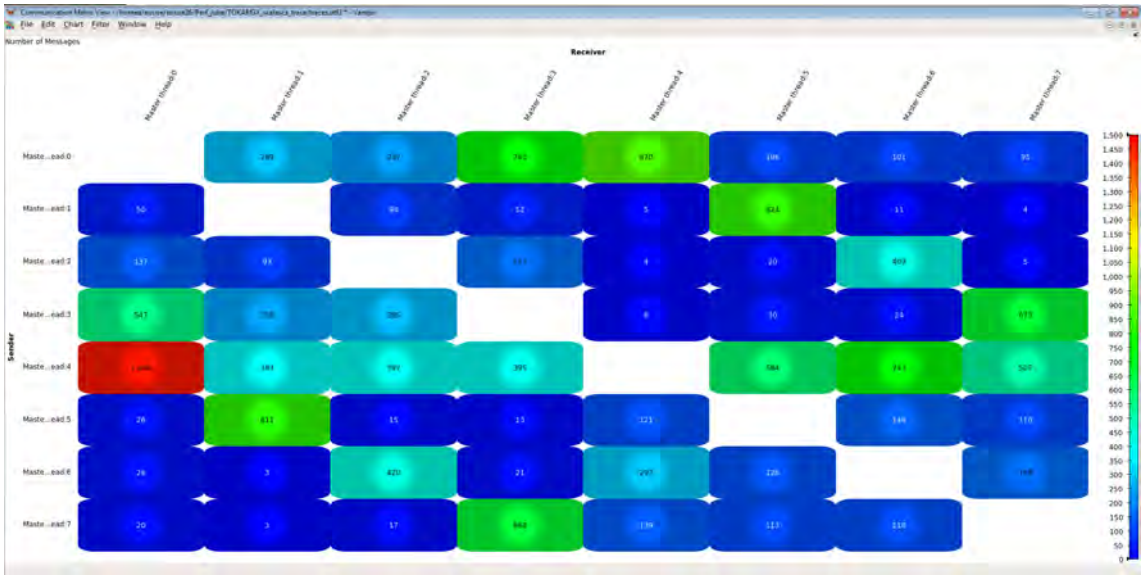


Figure 26: Screenshot from Vampir on our Test Case, matrix TOKAM3X on 8x1 cores, showing the communication matrix corresponding to region2 of the Factorisation phase. The block in (row i, column j) represents the number of messages sent from the process i to the process j.

A.16 Maphys

Code ID card

Code name	Maphys
Scientific domain	Sparse linear algebra
Description	Maphys is a parallel sparse linear algebra solver which couples direct and iterative approaches.
Languages	Fortran 90
Library dependencies	Partitionner: SCOTCH; Direct solver: Mumps or Pastix; Lapack; BLAS.
Programing models	MPI, multithreading supported (pthread or OpenMP depending on the direct solver)
Platforms	• Plafrim 2 • Jureca • Occigen
Scalability results	It has been ported on X86 architectures, scaling results are good up to 24000 cores (with a favorable test case).
Typical production run	Not used in production (yet)
Input / Output requirement	Maphys only reads an input matrix and right-hand side.
Application references	[1] E. Agullo, L. Giraud, S. Nakov, and J. Roman. Hierarchical hybrid sparse linear solver for multicore platforms. Research Report RR-8960, INRIA, Oct 2016. [2] L. Giraud, A. Haidar, and L. T. Watson. Parallel scalability study of hybrid preconditioners in three dimensions. Parallel Computing, 34:363–379, 2008. [3] A. Haidar. On the parallel scalability of hybrid solvers for large 3D problems. Ph.D. dissertation, INPT, June 2008. TH/PA/08/57.
Contact	• Luc Giraud (luc.giraud@inria.fr) • Emmanuel Agullo (emmanuel.agullo@inria.fr) • Gilles Marait (gilles.marait@inria.fr)

Performance metrics

Code team:

- Y. Ould Rouis (MdlS) for WP1
- Matthieu Kuhn (INRIA)and Gilles Marait (INRIA) for code developer

Case1 characteristics:

To use a relevant test case for the workshop we chose the medium matrix from TOKAM3X: `TOKAM3X_mat_inertia_limiter_medium.mtx` with its corresponding right hand side.

Domain size	374400 degrees of freedom
Resources	1 node on Jureca (16 cores)
IO details	Only reading input matrix and right hand side
Type of run	development run

MaPHyS is divided into 4 steps:

- Analysis: this step is sequential at the moment. It consists in reading the input matrix and right hand side from files, performing domain decomposition with the help of a partitionner (SCOTCH) and distributing the entries of the matrix to the other processes. At the end of this step, each process is given a non-overlapping interior partition and an interface shared with its neighbors. At the moment, the number of partitions must be a power of two because the partitionner is using the Nested Dissection method which only works on such cases. This step can be skipped if the user has already computed a domain decomposition and passes it to Maphys distributed interface.
- Factorization: we use a sparse direct solver (Pastix here) to compute the Schur complement matrix on the nodes on the interface. This step is entirely parallel and each process calls the direct solver sequentially. The Schur matrix resulting is generally a dense matrix.
- Preconditionning: here we compute the preconditionner to be used for the iterative method. In our case, we use an Additive Schwarz on the Schur complement.
- Solve: we solve the problem on the Schur complement using an iterative method with the preconditionner. Finally we compute the final solution.

Step	Time (s)	MPI Delay cost (s)
Analysis	72.05	67.20
Factorization	42.10	9.77
Preconditionning	11.52	4.78
Solve	21.06	6.47
Total	146.94	88.25

Table 26: Performance metrics for Maphys on the JURECA HPC system

Performance report

According to Table 26, Maphys spends most of its time in the analysis, sequential part of the algorithm. When looking at the trace we noticed that load balancing was bad so we took a look at MPI cost delay to measure this impact on performance.

Otherwise, the number of instruction per cycle is quite good when computing (between 2 and 3 according to paraver), taking advantage of the efficiency of the direct solver (Pastix here) and matrix computations.

Global results. Those 4 steps are clearly visible on figure 27. The analysis part is sequential and only the first process is working while all the others are waiting.
For the factorization, we can see that the load balancing is not very good and the first process seem to have more work to do.
We have the same problem for the preconditionning, as process 4 et 15 seem to have much more work than the others.
Finally for the iterative part, process 4 is always the slowest while all the other are locked in an `MPI_Allreduce` call.

Analysis. When looking at figure 27 we can see that most of the time is spend in this sequential step. However, in practice, this step is not as critical as it first looks. Users of MaPHyS may do multiple solve and this step does not have to be repeated. Also they have most of the time already performed a domain decomposition on their own (partitionning a mesh for example) and they can pass it to MaPHyS directly using a distributed interface.

We are currently trying to improve this step by developing a pre-treatment tool called Pad-

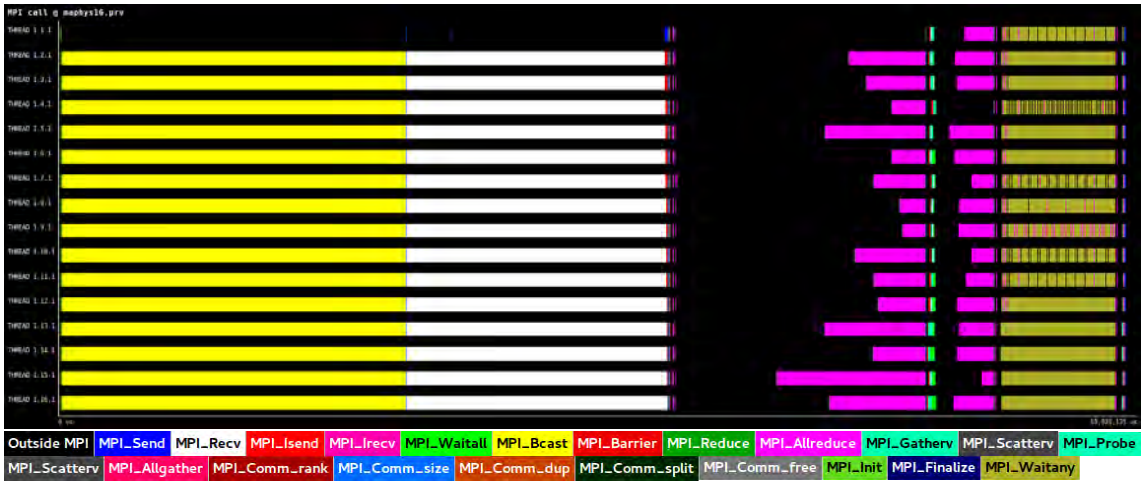


Figure 27: MPI profile of the run with Paraver

dle, which will give us more flexibility and parallelism for this step.

The result of this step is critical for performance because the domain decomposition will set the size of the interior and interface for each domain. They both have to be well balanced among the processes: the factorization step depends on the size of the interior part while the preconditionning and solve depend on the size of the interface.

Factorization. The factorization is entirely parallel. The only way to improve this step other than improving the direct solver is to make sure that the load balancing is good for the interior parts.

Preconditionning. This step is also very parallel. This time we would like the interface parts to be well-balanced so that the size of the tasks are equal for each process.

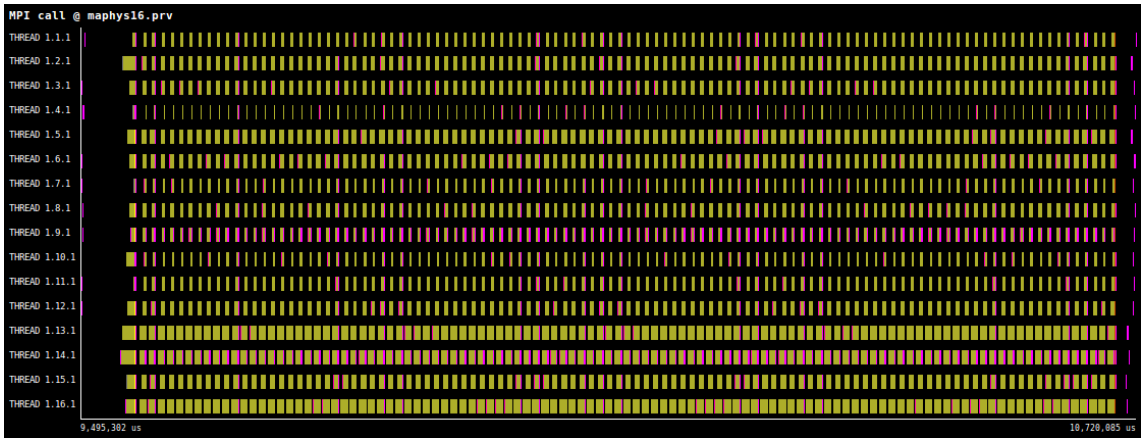


Figure 28: MPI profile of the solve step with Paraver

Solve. This is the iterative part. It is a critical step for some users need to repeat this step many times. A large part of the CPU time is spent in the `MPI_Allreduce` at the end of each iteration, as we can see on figure 28. The load imbalance is quite obvious between process 4 which spends all its time computing whereas process 13 and 16 are almost always waiting. The Paraver table tells us that we spend on average 26 % of the time in `MPI_Allreduce`.

This imbalance can also be seen by looking at the time spent in the lapack function with

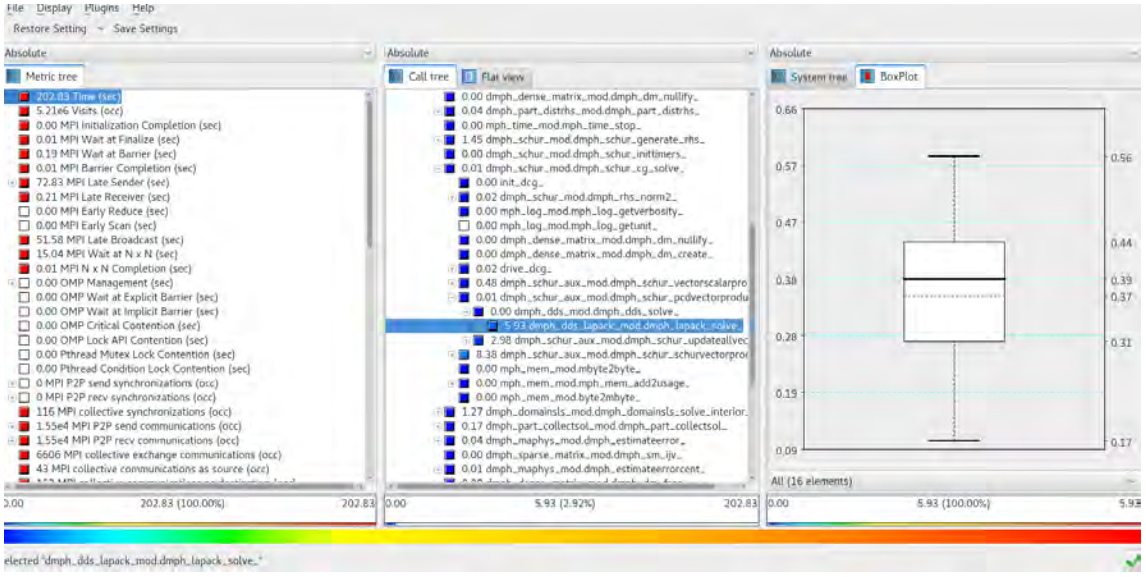


Figure 29: Scalasca time profile for the solve step

scalasca, figure 29. The box plot on the right hand side shows that some processes spend less than 0.28 seconds in this routine when some other spend 0.44 seconds.

Conclusion. As a conclusion, in order to improve Maphys, we would recommend the following roadmap:

1. work on the analysis step. Although it not critical for most of MaPHyS' users who have already computed a domain decomposition, we would like to have more control on the load balancing and the speed of this step for testing and purely algebraic problems. We are already developing a separate library (Paddle) with this purpose.

A.17 DL_MESO

Code ID card

Code name	DL_MESO
Scientific domain	Meso- and Multi-scale modelling
Description	DL_MESO is a general purpose mesoscale simulation package developed by Michael Seaton for CCP5 under a grant provided by EPSRC. It is written in Fortran90 and C++ and supports both Lattice Boltzmann Equation (LBE) and Dissipative Particle Dynamics (DPD) methods. It is supplied with its own Java-based Graphical User Interface (GUI) and is capable of both serial and parallel execution. In this report we will always refer to the DPD method
Languages	Fortran90
Library dependencies	FFTW3
Programing models	MPI and OpenMP
Platforms	• UK Tier0 Archer
Scalability results	It has been ported on X86 architectures, scaling results are good up to 1024#cores
Typical production run	24h on 64 - 512 cores
Input / Output requirement	• Size: 1GB / 24h • Single post-processing output: 100 MB • Single restart output: 100 MB
Application references	DL_MESO: highly scalable mesoscale simulations. MA Seaton (STFC Daresbury Lab.), RL Anderson (STFC Daresbury Lab.), S Metz (STFC Daresbury Lab.)ORCID icon, W Smith (STFC Daresbury Lab.) Mol Simul 39, no. 10 (2013): 796-821
Contact	• Michael Seaton (michael.seaton@stfc.ac.uk)

Performance metrics

Code team:

- R. Halver for WP1
- Jony Castagna, Michael Seaton (STFC, DL) for code developer

Case1 characteristics:

The benchmark consists in the simulation of a plasma made of 300k ions (half positive and half negative charged, so to have an overall neutral electrical field) in a 3D periodic box. This involves the use of special algorithms (like the Ewald summation method) to calculate the long and short electrostatic interactions, which in Molecular Dynamics is usually the highest computational cost. Three test cases have been run for an overall total of 120, 240 and 480 cores.

Domain size	300000 ions
Resources	5,10 and 20 node on Jureca (24 cores)
IO details	Checkpoint written every 10 steps ⇒ equal to production
Type of run	production run

	Metric name	120 cores	240 cores	480 cores
Global	Total Time (s)	80	43	28
	Time IO (s)	0.12	0.22	0.15
	Time MPI (s)	4.42	4.16	7.31
	Memory vs Compute Bound	1.01	1.00	1.12
	Load Imbalance (%)	4.29	6.35	14.95
IO	IO Volume (MB)	32.53	34.15	37.39
	Calls (nb)	609262	618269	636269
	Throughput (MB/s)	280.79	158.22	250.87
	Individual IO Access (kB)	0.11	0.11	0.12
MPI	P2P Calls (nb)	3603	3603	3603
	P2P Calls (s)	1.99	1.79	2.29
	P2P Calls Message Size (kB)	87	65	50
	Collective Calls (nb)	703	703	703
	Collective Calls (s)	1.70	1.14	2.35
	Coll. Calls Message Size (kB)	15680	31361	62723
	Synchro / Wait MPI (s)	3.54	2.83	4.66
	Ratio Synchro / Wait MPI (%)	79.67	67.34	63.50
Node	Time OpenMP (s)	N.A.	N.A.	N.A.
	Ratio OpenMP (%)	N.A.	N.A.	N.A.
	Synchro / Wait OpenMP (s)	N.A.	N.A.	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.	N.A.	N.A.
Mem	Memory Footprint	80912kB	156272kB	281708kB
	Cache Usage Intensity	0.97	0.96	0.97
Core	IPC	0.71	0.68	0.71
	Runtime without vectorisation (s)	99	50	31
	Vectorisation efficiency	1.24	1.16	1.11
	Runtime without FMA (s)	79	42	27
	FMA efficiency	0.99	0.98	0.96

Table 27: Performance metrics for DL_MESO on the JURECA HPC system using a different number of cores

Performance report

A resume of the performance metrics gathered for all 3 test cases, using the Scalasca 2.3.1 tool instrumented with Score-P 3.0-p1, is presented in Table 27.

We first analyse the 120 cores run: the MPI communication + IO represent only the 6% of the total time, the rest (94%) is all spent in computation. The communication is split in P2P calls (2.5%) and Collective calls (2.1%), plus a synchronization efficiency of 80%. The IO volume of data is of 33MB and with a throughput of 281MB/s it represents only the 0.2% of the total time. The vectorization improves the performance of a 20% while the use of Fused Multiply-Add instructions (FMA) does not alter the runtime.

The strong scaling results (240 and 480 cores) show that the MPI communication time increases linearly (of a factor 2) with the number of cores (12% and 22%, respectively). However, the P2P and Collective time is not growing linearly, but actually oscillate around the 120 cores value. Instead, the percentage of the ratio between synchronization and MPI wait time constantly decreases as the number of number of cores increases. This suggests a problem of parallel efficiency in the scalability of the code which requires a deeper analysis:

1. First, we identify which part of the program consume most of the time. This is done looking at a plot graph produced by Cube 4.3.4 tracing program. However, a first tracing profile shows that 6 subroutines are called more than 1 billion times (Figure30) requiring

a a very large tracing file (>130GB). The visiting time of each of those subroutine is very small ($0.1\mu s$) and it represents only 4% of the total time. A filter on those 6 subroutines has then been applied in order to reduce the size of the tracing files. The filtered call tree plot and the box plot are presented in Figure 31. This shows that most 86% of the time is spent in the Ewald module, which is the subroutine solving the long range of the electrostatic forces between particles, and most luckily be the cause of the imbalance.

2. Second, we analyse the communication between cores using the Paraver Tracing 4.6.3 tool after instrumenting the code with Extrae 3.4.3. This will help to understand if there is a reduction of the number of instructions per cycle (IPC) for some cores which will bring to the loss of parallel efficiency.

The analysis of Paraver on the 120 cores shows that the average parallel efficiency is very good (97%). Figure 32 shows the so called "histogram of useful instructions", which is obtained by the product of the number of instructions per cycle and an instruction flag equal to 1 for computing and 0 for communication. It represents the number of instructions spent in computing time only. The plot shows that most of MPI time is no really due to transfer of data but to unbalances and dependencies of the code. Despite the main computation regions show unbalance correlated with the instructions, there are few outliers that are the ones that cause the increase on the MPI time because all the other processes have to wait for them both in the point to point calls as well as on the collectives. These processes have a significantly lower IPC (for a region with 2.7 IPC on these processes it goes down to 2 or 1.7). There is no variation on the cycles per microsecond and they do not execute more instructions, so seems highly correlated with the IPC.

A main cause of this imbalance can be attributed to the clustering of particles which will lead to a different number of particles per core. However, further investigation is needed to understand better the cause of it.

As a conclusion, the full performance analysis suggests that the code has a good performance for a typical production run. However, as the number of cores increases, we would recommend the following roadmap:

1. a review of the code (in particular the Ewald summation module) in order to understand the imbalance in the IPC which lead to a loss of performance

```
Estimated aggregate size of event trace:                   136GB
Estimated requirements for largest trace buffer (max_buf): 1182MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       1184MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=1184MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type    max_buf[B]        visits  time[s] time[%] time/visit[us]  region
         ALL 1,238,509,948 5,578,922,537 10799.71   100.0           1.94  ALL
         USR 1,237,735,174 5,576,529,745  1606.23    14.9           0.29  USR
         MPI       489,572     1,009,796   656.90     6.1         650.53  MPI
         COM       305,734     1,382,996  8536.58    79.0        6172.53  COM

         USR 1,128,162,646 5,081,008,960   472.56     4.4           0.09  numeric_container.erfcdp_
         USR    41,081,144   184,199,898    18.15     0.2           0.10  numeric_container.mtrnd_
         USR    41,013,050   183,899,778    18.56     0.2           0.10  field_module.conservativeforce_
         USR    24,992,318   114,722,651    13.87     0.1           0.12  field_module.loadpart_
         USR     1,783,886     8,185,127    11.91     0.1           1.46  field_module.diff_
         USR       655,200     3,024,000     1.27     0.0           0.42  ewald_module.loadpart_ewald_
         USR       204,282       900,360     0.10     0.0           0.11  numeric_container.duni_
         MPI       160,200       216,000     2.51     0.0          11.64  MPI_Isend
         MPI       160,200       216,000     0.69     0.0           3.19  MPI_Irecv
         MPI        93,600       432,000   218.39     2.0         505.54  MPI_Wait
         MPI        47,804        84,360   319.56     3.0        3788.01  MPI_Allreduce
         COM        46,800       216,000     0.12     0.0           0.57  comms_module.msg_wait_and_size_double_
         COM        46,800       216,000     0.15     0.0           0.68  comms_module.msg_send_unblocked_
         COM        46,800       216,000     0.09     0.0           0.43  comms_module.msg_receive_unblocked_pe_
         COM        46,800       216,000     0.09     0.0           0.43  comms_module.msg_wait_
         USR        46,800       216,000   209.84     1.9         971.48  ewald_module.diff_ewald_
         USR        26,598       122,760     0.03     0.0           0.25  numeric_container.qsort_integer_
         USR        16,198        74,760     0.01     0.0           0.09  numeric_container.bitadd_
         USR        16,198        74,760     0.01     0.0           0.10  numeric_container.bitmult_
         COM        15,600        72,000     7.55     0.1         104.83  domain_module.import_
         COM        15,600        72,000     1.49     0.0          20.75  domain_module.deport_
         COM        15,600        72,000     4.44     0.0          61.69  domain_module.export_
         COM        13,104        60,480     0.02     0.0           0.35  comms_module.mynode_
         MPI        13,104        60,480     0.01     0.0           0.18  MPI_Comm_rank
         COM        13,078        60,360     0.04     0.0           0.63  comms_module.timchk_
         MPI         7,259           238     0.00     0.0           2.69  MPI_Send
         MPI         7,259           238    32.85     0.3      138020.72  MPI_Recv
         COM         5,226        24,120     0.81     0.0          33.47  comms_module.global_sum_dble_
         USR         5,200        24,000     5.53     0.1         230.41  domain_module.parlnk_
         COM         5,200        24,000     0.81     0.0          33.66  integrate_dpd_mdvv.mdvv_nvt_
         COM         3,094           119     0.00     0.0           0.79  comms_module.msg_send_sca_blocked_
         COM         3,094           119     0.00     0.0           0.45  comms_module.msg_receive_blocked_
         USR         2,860        13,200     0.04     0.0           3.04  parse_utils.getword_
         COM         2,626        12,120     0.01     0.0           0.54  comms_module.global_sum_sca_int_
         USR         2,626        12,120     0.01     0.0           0.54  parse_utils.lowercase_
         COM         2,600        12,000     0.65     0.0          53.80  field_module.plcfor_mdvv_
         COM         2,600        12,000     0.02     0.0           1.35  domain_module.deportdata_
         COM         2,600        12,000     0.05     0.0           4.07  domain_module.exportdata_
         COM         2,600        12,000     0.01     0.0           0.72  comms_module.global_sca_max_int_
         USR         2,600        12,000    56.04     0.5        4669.79  field_module.forces_mdvv_
         USR         2,600        12,000     0.00     0.0           0.34  field_module.freeze_beads_
         COM         2,600        12,000     0.00     0.0           0.40  comms_module.global_sca_or_all_
         USR         2,600        12,000   796.19     7.4       66349.40  ewald_module.ewald_real_slater_
         COM         2,600        12,000  8515.31    78.8      709608.88  ewald_module.ewald_reciprocal_
         COM         2,600        12,000     0.00     0.0           0.37  comms_module.global_sca_min_dble_
         COM         2,600        12,000     0.03     0.0           2.25  domain_module.importdata_
         COM         2,600        12,000     0.01     0.0           0.45  comms_module.global_sca_max_dble_
         COM         2,600        12,000     0.06     0.0           5.06  statistics_module.statis_
         USR         1,040         4,800     0.02     0.0           3.39  parse_utils.getdble_
         USR           780         3,600     0.00     0.0           0.44  parse_utils.parsedble_
         USR           572         2,640     0.00     0.0           0.40  parse_utils.parseint_
         USR           572         2,640     0.00     0.0           0.37  parse_utils.getint_
         USR           286            11     0.00     0.0         131.45  statistics_module.printout_
         MPI            68           120     1.89     0.0       15782.35  MPI_Barrier
         MPI            26           120     0.00     0.0           0.41  MPI_Comm_size
         USR            26           120     0.10     0.0         846.68  config_module.zero_
         USR            26           120     0.00     0.0           3.51  config_module.elecgen_
         USR            26           120     0.01     0.0         111.79  domain_module.domain_dimensions_
         USR            26           120     0.00     0.0           0.42  surface_module.surfacenodes_
         COM            26           120     3.65     0.0       30422.92  start_module.start_
         COM            26           120     0.17     0.0        1379.36  start_module.initialize_
         COM            26           120     0.00     0.0          18.60  domain_module.domain_decompose_
         USR            26           120     0.10     0.0         851.59  config_module.read_field_
         USR            26           120     0.05     0.0         448.75  config_module.scan_field_
         COM            26           120     0.00     0.0           1.76  comms_module.global_sum_int_
         COM            26           120     0.10     0.0         853.99  start_module.initialvelocity_
         USR            26           120     0.04     0.0         292.25  config_module.read_control_
         COM            26           120     0.07     0.0         591.90  config_module.sysdef_
         USR            26           120     0.17     0.0        1422.26  ewald_module.ewald_reciprocal_map_
         USR            26           120     0.00     0.0          15.24  start_module.create_local_id_mol_map_
         USR            26           120     0.00     0.0           2.87  numeric_container.quicksort_integer_indexed_
         COM            26           120     0.29     0.0        2430.61  run_module.mdvv_
         USR            26           120     0.02     0.0         188.34  statistics_module.result_
         USR            26           120     0.32     0.0        2684.18  config_module.scan_control_
         COM            26           120     0.00     0.0           2.21  comms_module.numnodes_
         USR            26           120     1.25     0.0       10444.60  statistics_module.exportout_
         COM            26           120     0.00     0.0          14.00  comms_module.initcomms_
         COM            26           120     0.16     0.0        1327.10  MAIN__
         USR            26           120     0.00     0.0          36.28  config_module.free_memory_
         COM            26           120     0.38     0.0        3184.83  comms_module.exitcomms_
         COM            26           120     0.00     0.0           5.76  comms_module.gsync_
         COM            26           119     0.00     0.0           6.83  comms_module.msg_receive_sca_blocked_
         COM            26           119     0.00     0.0           3.86  comms_module.msg_send_blocked_
         MPI            26           120    80.86     0.7      673838.01  MPI_Init
         MPI            26           120     0.13     0.0        1093.40  MPI_Finalize
```

Figure 30: Scalasca profiling for the 120cores test case without filter.

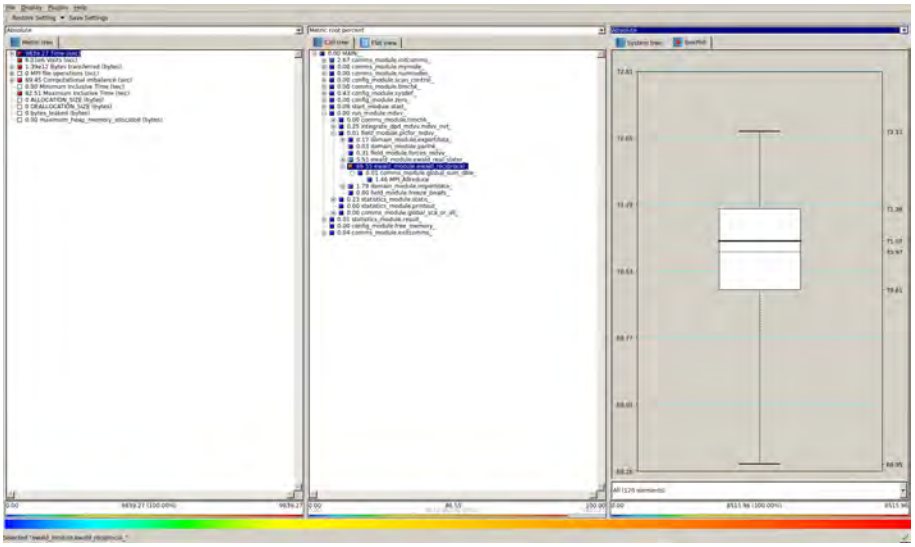


Figure 31: Tree call and box plots for the 120 cores with filter.

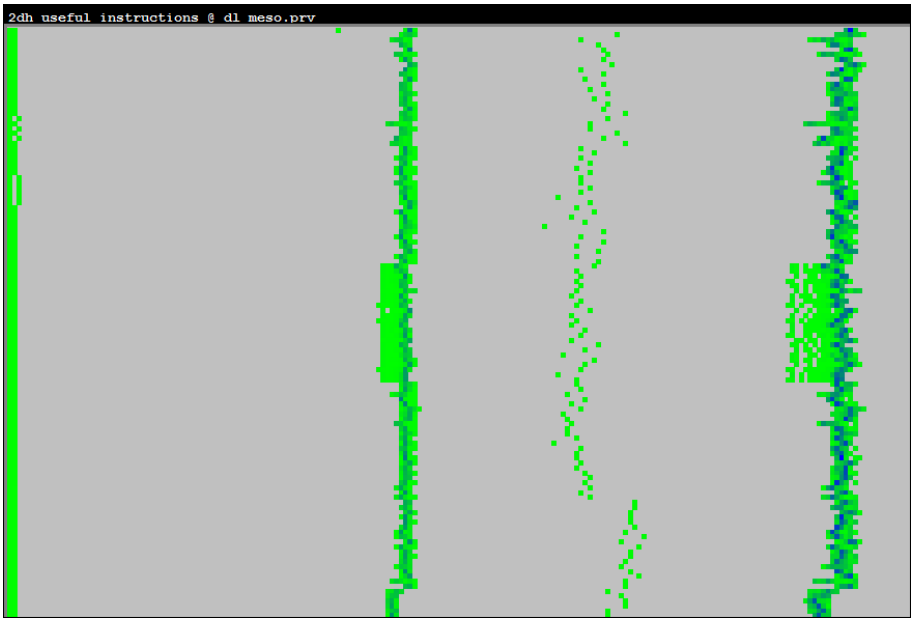


Figure 32: Paraver histogram of instructions for the 120 cores tes case.

A.18 Compass

Code ID card

Code name	ComPASS
Scientific domain	subsurface mass and energy transfers in fractured porous media: application to the modeling of high temperature geothermal systems and reservoirs
Description	Non-isothermal compositional multiphase Darcy flows are simulated on 3D unstructured meshes including networks of fractures. Fracture flows are simulated using a 2D model which is coupled with a 3D model in the matrix. The problem is discretized using a fully implicit time integration combined with the Vertex Approximate Gradient (VAG) finite volume scheme. The fully coupled systems are assembled and solved in parallel with one layer of ghost cells. This strategy allows for a local assembly of the discrete systems. A CPR-AMG preconditioner is implemented to solve the linear systems at each time step and each Newton type iteration of the simulation. The formulation of the compositional model is based on a generic extended Coats' type formulation. It accounts for an arbitrary nonzero number of components in each phase allowing to model immiscible, partially miscible or fully miscible flows. Several equation of states can be implemented. A well model was introduced recently. The well geometry is discretized by a set of edges of the mesh to represent efficiently slanted or multi-branch wells. The connection with the 3D matrix and the 2D fault network is accounted for using Peaceman's approach.
Languages	Fortran 2003, C++ with python interface (using pybind11)
Library dependencies	MPI, PETSc (v3.5, Hypre) and METIS are mandatory for the minimal standalone version, pybind11 is used for the python interface, VTK and HDF5 are optional, a transitory light dependency on boost has been introduced
Programing models	MPI
Platforms	Tested on lunix clusters: *cicada* of University of Nice (Intel Sandy Bridge) and *srv185* of BRGM (AMD Abu Dhabi)
Scalability results	Scaling results are good up to 256 cores on X86 architectures.
Typical production run	24h on 8 - 512 cores
Input / Output requirement	Problem is described through compiled Fortan files or via python scripts. Typical ouput are VTK pvtu files with size: 10 GB / 24h run.
Relevant kernel algorithms	Coat's type formulation for thermal compositional flows, well model, Vertex Approximate Gradient (VAG) finite volume scheme do discretize Darcy-like/gradient flows, *fully implicit* active set Newton-Raphson algorithm relying on a CPR-AMG preconditioner, local assembly with Schur complement to pre-eliminate certain degrees of freedom and Jacobian filling
Software licence	CeCILL v2.1 / GNU GPL V3
Application references	Xing *et al.* (2016), Xing *et al.* (2017)
Contact	s.lopez@brgm.fr, roland.masson@unice.fr

Performance metrics

Performances of the ComPASS code were assessed twice in the framework of two EoCoE hands-on workshop on HPC benchmarking and performance analysis:

- Session 1 was held in spring 2016 at Maison de la Simulation
- Session 2 was held in spring 2017 at Barcelona Supercomputing Center

Code team:

- 2016 session:
  - Matthieu Haefele (MdlS) and Yacine Ould-Rouis (MdlS) for WP1
  - Simon Lopez (BRGM) and Feng Xing (INRIA/BRGM) for external partners
- 2017 session:
  - Abel Marin-Lafleche (MdlS) for WP1
  - Michel Kern (MdlS), Simon Lopez (BRGM) for external partners

Compared to the 2016 version, the 2017 version of the ComPASS code included geothermal well modeling and a preliminary python interface build on top of the previous Fortran code. The performance of the code used through the python interface were only partially tested: Paraver could be used but link problems still need to be solved to use the score-P system.

Case characteristics:

Both sessions used a typical simple production case based on a geothermal doublet exploiting a monophasic aquifer (*cf.* figure 33). The reservoir was discretized using regular grid with homogeneous properties.

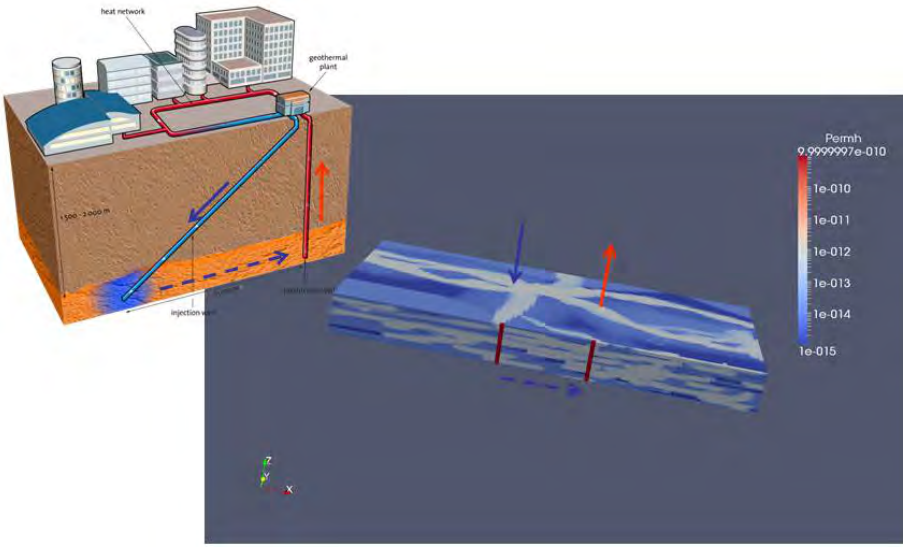


Figure 33: Typical geothermal doublet which consists of a closed loop with one hot production well (red) and one cold injection well (blue). Injection of the cooled brines leads to the progressive and temporary exhaustion of the resource at the local doublet scale.

Performance report

	Metric name	03/01/2016
	Test-case	case1
Golbal	Total Time (s)	43.2
	Time IO (s)	0.3
	Time MPI (s)	12.4
	Memory vs Compute Bound	1.1
IO	IO Volume (MB)	35.8
	Calls (nb)	384000
	Throughput (MB/s)	105.0
	Individual IO Access (kB)	0.1
MPI	P2P Calls (nb)	0
	P2P Calls (s)	0.0
	Collective Calls (nb)	2721
	Collective Calls (s)	0.1
	Synchro / Wait MPI (s)	11.7
	Ratio Synchro / Wait MPI	94.8
	Message Size (kB)	908.4
	Load Imbalance MPI	24.8
Node	Ratio OpenMP	0.0
	Load Imbalance OpenMP	0.0
	Ratio Synchro / Wait OpenMP	0.0
Mem	Memory Footprint (B)	66 mB
	Cache Usage Intensity	N.A.
	RAM Avg Throughput (GB/s)	N.A.
Core	IPC	N.A.
	Runtime without vectorisation (s)	46.5
	Vectorisation efficiency	1.1
	Runtime without FMA (s)	44.6
	FMA efficiency	1.0

Table 28: Performance metrics for Compass on the JURECA HPC system at 2016 MdlS Workshop

Table 28 presents preliminary results obtained with the 2016 version of the code. Unfortunately, we no longer have details on the run parameters. The results are in line with those obtained at the 2017 workshop that are described next.
Table 29 describes a more recent set of experiments. The column labelled `N2500_P04` is a "small" case with a grid size of $61 \times 41 \times 1$ on 4 cores, while the column labelled `N98000_P48` is a larger case with a grid size of $121 \times 81 \times 10$, run on 48 cores. Note that the final simulation time for the small case was larger than for the large case, and as a result the number of time steps was three times larger (and the same is approximately true for the number of Newton iterations).
We also point out that Compass uses the PETSc library for solving the linear system at each Newton iteration, and that PETSc was not instrumented. It was clearly not our purpose to evaluate the performance of PETSc, which we believe to be quite good in any case!
To obtain meaningful numbers, we compare the average runtime per Newton iteration divided by the number of grid points. We obtain 30.5 µs for the small run and 3.06 µs for the larger run. Given that the number of cores is 12 times larger in the latter case, this points to a rather satisfactory scaling behavior.
The load imbalance has increased a lot for the larger run. We believe this is due to (at least) two different causes:

- First, the mesh is read and partitioned on a master processor, then sent to the other processors;
- Second this partitioning process is not perfect (this has been confirmed both by runs with Paraver, and by looking at the shape of the subdomains). The subdomains that touch

	Metric name	N2500_P04	N98000_P48
Global	Total Time (s)	14	27
	Time IO (s)	N.A.	N.A.
	Time MPI (s)	1.95	6.87
	Memory vs Compute Bound	1.00	0.97
	Load Imbalance (%)	7.20	24.95
IO	IO Volume (MB)	N.A.	N.A.
	Calls (nb)	N.A.	N.A.
	Throughput (MB/s)	N.A.	N.A.
	Individual IO Access (kB)	N.A.	N.A.
MPI	P2P Calls (nb)	418977	642828
	P2P Calls (s)	0.87	2.09
	P2P Calls Message Size (kB)	0	0
	Collective Calls (nb)	17957	8981
	Collective Calls (s)	0.86	2.09
	Coll. Calls Message Size (kB)	0	1
	Synchro / Wait MPI (s)	1.07	5.05
	Ratio Synchro / Wait MPI (%)	50.82	72.75
Node	Time OpenMP (s)	N.A.	N.A.
	Ratio OpenMP (%)	N.A.	N.A.
	Synchro / Wait OpenMP (s)	N.A.	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.	N.A.
Mem	Memory Footprint	47812kB	188252kB
	Cache Usage Intensity	0.94	N.A.
Core	IPC	2.34	N.A.
	Runtime without vectorisation (s)	14	27
	Vectorisation efficiency	1.00	1.00
	Runtime without FMA (s)	14	26
	FMA efficiency	1.00	0.96

Table 29: Performance metrics for Compass on Jureca at EoCoE-Pop Barcelona workshop (04 2017)

the boundaries typically have less communication work than those in the middle of the domain. This effect was felt much less on the smaller case, because the domains were more similar (on 4 cores) than for the larger case (on 48 cores).

We have looked in more detail at the Scalasca report for the large test case. This confirms our conclusions for the source of the imbalance: the total imbalance is 7.1s (elapsed time), 1.7s of which is due the mesh partitioning (done sequentially by the master process). Of the 5 remaining seconds, half comes from unbalanced computation (forming the Jacobian, solving the linearized system), and we suspect the the other half may come from creating directories to store the results at each output step (which is again done on the master process).

As a conclusion (and after discussions held during the workshop with the attending experts), in order to improve Compass, we would recommend the following roadmap:

1. Improve the routine that computes the local Jacobian matrix `jacobian_jacbiga_bigsm`, as it account for 247s out of a total CPU time of 1380s. Remember that the solution part, which accounts for another 300s is carried out by calling PETSc - the latest computation times is likely to increase with stiffer problems (e.g. involving the presence of gaz phase) but this remains to be tested;

2. The code shows little, if at all, improvement due to vectorization. The routine mentioned above may be a prime candidate for looking at this issue. As currently written, it has one large loop over the mesh cells, and a large number of small loops over the nodes or faces of each cell. Reversing the order of the loops, and otherwise assisting the compiler, might improve the performance;

3. Reducing the load imbalance may not be easy, given that we have little control on what Metis does for partitioning the mesh. Nevertheless, this is still the dominant cause for performance loss, and deserves further investigation.

A.19 WRF-Solar

Code ID card

Code name	WRF-Solar
Scientific domain	W.P. 6.5: Co-Design activities for exascale Hardware and software
Description	The weather research & forcasting model (WRF) is a numerical weather prediction system design for both atmospheric research and operational forcasting needs. WRF-solar is a specific configuration and augmentation of WRF design for solar energy applications
Languages	C and Fortran
Library dependencies	MPI, NetCDF, NetCDF-Fortran, Jasper, libpng, zlib, flex, Bison and NCO
Programing models	MPI
Platforms	• JURECA (5000 CPUh in 2016) • CYTERA (10000 CPUh in 2016)
Scalability results	It has been ported on X86 architectures and more specifically on the JURECA machine. Scaling performance evaluation results are good up to 384 cores cores (15TFLOPS peak perfomance) .
Typical production run	10-40h - 100-200 cores
Input / Output requirement	• Size: 100 GB / 24h run • Single post-processing output: 200 MB • Single restart output:
Application references	Jimenez et al 2016
Contact	• Constantinos Demetroullas (c.demetroullas@cyi.ac.cy)

Performance metrics

Code team:

- Pedro A. Jimenez (National Center for Atmospheric Research): Code developer
- Constantinos Demetroullas (Computation-based Science and Technology Research Center, The Cyprus Institute): Code optimisation
- Swen Metzger (Computation-based Science and Technology Research Center, The Cyprus Institute): Supervisor

Case1 characteristics:

Domain size	65 x 65 + 51 x 51 grid
Resources	1 node on Jureca (24 cores)
IO details	IO is similar to production run
Run description	A small test case that can be run on a single node on a very short time (10-15 minutes). This test is perfect to test the profiling and tracing tools

Table 30: Performance metrics for WRFSolar on the JURECA HPC system

Benchmark code characteristics:
WRFSolar has been analysed using the Scalasca/Score-P and Darshan performance tools. The results are shown in Table 31. The execution time profile is summarised in Fig 34

	Metric name	Code state Workshop Barcelone April 2017
Global	Total Time (s)	10132
	Time IO (s)	330
	Time MPI (s)	4137
Individual	zolri	482.3
	wrf_message	621.5
	sintb_	52.5
	module_configure.in_use_for_config_	328.5

Table 31: WRFSolar extracted values for case 1

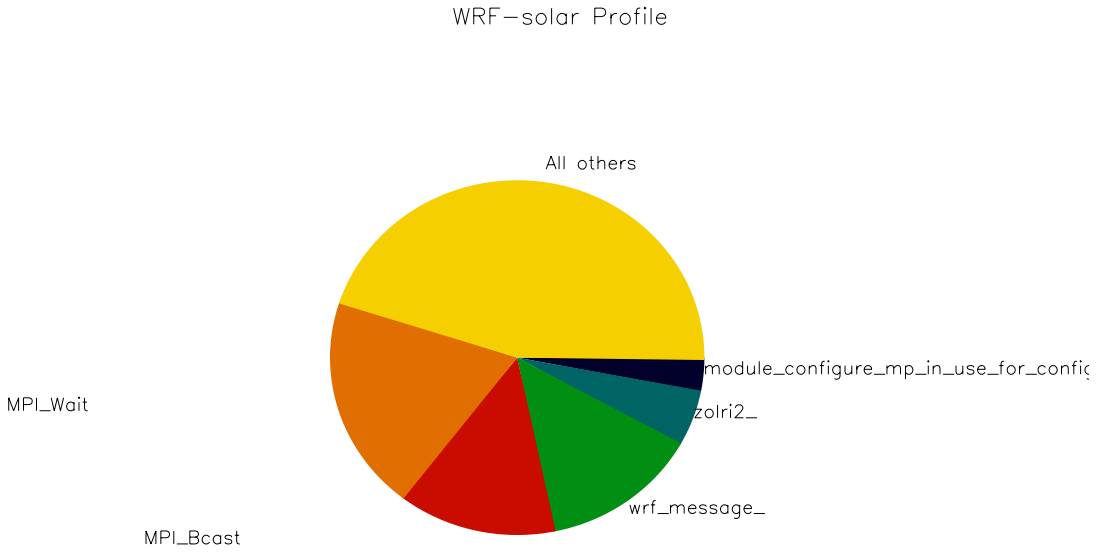


Figure 34: WRFSolar execution time profile for the a pair of grid sizes of 65 x 65 + 51 x 51 and executed on one Jureca machine.

The trace created by Scalasca/Score-P has shown that in this small case is that all functions are very well optimised and that the MPI communication is what slows down the execution of the program. This of course is misleading as we will illustrate in the next case, since the small grid size is the main reason the MPI_Wait is the dominant factor when it comes to computing time.
Case2 characteristics: Increasing the generated maps' grid size will test if the size of the generated arrays' has an effect on the code performance. We therefore test WRFSolar by generating maps 1800 across on 8 nodes (192 cores).

Domain size	55 x 55, 289 x 289, 1801 x 1801 grid
Resources	8 nodes on Jureca (192 cores)
IO details	IO frequency similar to production
Run description	A bigger case but using more nodes still able to be completed within 1-2 hours.

Table 32: Performance metrics for WRFSolar on the JURECA HPC system

Benchmark code characteristics:
WRFSolar has been analysed once again using the Scalasca/Score-P and Darshan performance tools. The results are shown in Table 33 and summarised in Fig 35
The trace created by Scalasca/Score-P has shown that in this case the code performs really well, no process takes a considerable amount of time and MPI communication is only a fraction of the total execution time. Therefore no further optimisation of the code is needed because even if we speedup by 100 times the most time consuming function of the code we will still only get 6-7% increase in computing time.
The code at this point, although it uses the MPI library to split the computations between different cores, it still only uses the head core to read in and write out the data. Using the library PNetCDF we repeat the Case 2 experiment. Using the Darshan performance tool we measure a speedup in total execution time again at ∼7% (total execution time of 5087s). The Darhian performance tool reports show that the average achieved speedup for the independent reads and writes, when using the P-NetCDF library, is 2 and 10 respectively.

Performance report

Conclusions

As a conclusion we find that the WRF-solar code is very well optimised and it does not need any further optimisation. On the other hand using the P-NetCDF library (instructions for downloading/installing/executing can be provided) one can achieve a significant speedup (depending on the amount of reads and writes that instructs the program to perform and on the resolution of the maps) without having to put much effort into it.

	Metric name	Code state Workshop Barcelone April 2017
Global	Total Time (s)	5439
	Time IO (s)	352
	Time MPI (s)	81.5
Individual	zolri	381.2
	wrf_message	489.7
	sintb_	272.5
	module_configure.in_use_for_config_	271.1

Table 33: WRFSolar extracted values for case 2

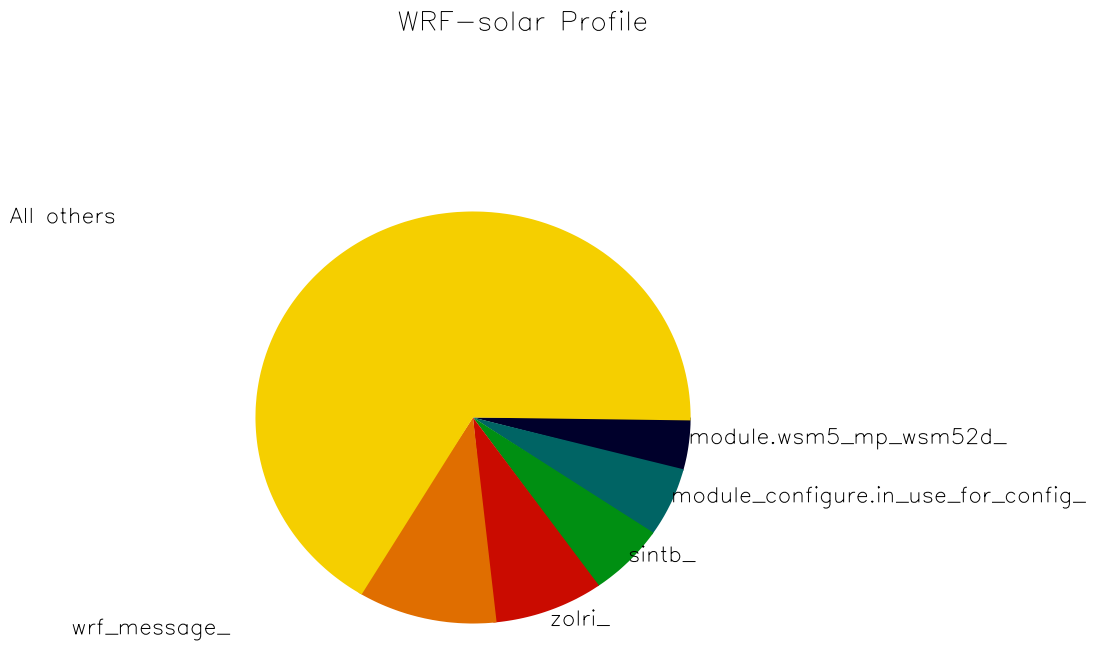


Figure 35: WRFSolar execution time profile for the a pair of grid sizes of 65 x 65 + 51 x 51 and executed on one Jureca machine.

A.20 CP2K
Code ID card

Code name	CP2K
Scientific domain	WP3 Molecular dynamic
Description	CP2K is a quantum chemistry and solid state physics software package that can perform atomistic simulations of solid state, liquid, molecular, periodic, material, crystal, and biological systems. CP2K provides a general framework for different modeling methods such as DFT using the mixed Gaussian and plane waves approaches GPW and GAPW. Supported theory levels include DFTB, LDA, GGA, MP2, RPA, semi-empirical methods (AM1, PM3, PM6, RM1, MNDO, ...), and classical force fields (AMBER, CHARMM, ...).
Languages	Fortran 2003 (> 1 M lines).
Library dependencies	blacs, scalapack, FFTW3, libint, libxc, libgrid, libsmm.
Programing models	MPI, OpenMP and CUDA..
Platforms	• CRESCO • JURECA
Scalability results	Scaling results are good up to 65000 cores (https://www.cp2k.org/performance).
Typical production run	24h on 64 - 512 cores
Input / Output requirement	• Size: 10 GB / 24h run • Single post-processing output: 100MB • Single restart output: 100MB
Software licence	GPL licence.
Application references	Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed, M. Parrinello, T. Chassaing and J. Hutter, Comp. Phys. Comm. 167, 103 (2005). An efficient orbital transformation method for electronic structure calculations, J. VandeVondele and J. Hutter, J. Chem. Phys. 118, 4365 (2003). Auxiliary Density Matrix Methods for Hartree-Fock Exchange Calculations, M. Guidon, J. Hutter, and J. VandeVondele, J. Chem. Theory Comput. 6, 2348 (2010).
Contact	• Massimo Celino (massimo.celino @enea.it) • Michele Gusso (michele.gusso@enea.it)

Performance metrics

Code team:

- Sebastian Lührs (FZJ) for WP1
- Agostino Funel (ENEA) for WP1
- Michele Gusso (ENEA) for WP3

Case1 characteristics:
The benchmark consists of a 1 step of Born-Oppenheimer molecular dynamics simulation of amorfous hydrogenated silicon (a-SiH 512 Si atoms + 64 H atoms in a 22 ang cubic box) using the Quickstep CP2K module. The atomic basis set was TZV2P and the planewave cutoff was 400

Ry. The PBE functional was used for the Exchange-Correlation energy. The initial guess of the electronic density was based on atomic orbitals. The benchmark was aimed at evaluating the Quickstep module. 36 processors were used.

	Metric name	SiH.json
Global	Total Time (s)	180
	Time IO (s)	0.03
	Time MPI (s)	29.19
	Memory vs Compute Bound	1.04
	Load Imbalance (%)	9.33
IO	IO Volume (MB)	859.95
	Calls (nb)	18042
	Throughput (MB/s)	28823.61
	Individual IO Access (kB)	55.90
MPI	P2P Calls (nb)	46380
	P2P Calls (s)	14.12
	P2P Calls Message Size (kB)	716
	Collective Calls (nb)	23844
	Collective Calls (s)	13.38
	Coll. Calls Message Size (kB)	221
	Synchro / Wait MPI (s)	13.32
	Ratio Synchro / Wait MPI (%)	44.84
Node	Time OpenMP (s)	N.A.
	Ratio OpenMP (%)	N.A.
	Synchro / Wait OpenMP (s)	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.
Mem	Memory Footprint	1181460kB
	Cache Usage Intensity	0.97
Core	IPC	0.65
	Runtime without vectorisation (s)	192
	Vectorisation efficiency	1.07
	Runtime without FMA (s)	196
	FMA efficiency	1.09

Table 34: Performance metrics for CP2K on the JURECA HPC system, 512 Si atoms + 64 H atoms, case1

Case2 characteristics:
This is a short molecular dynamics run of 100 time steps in a NPT ensemble at 300K. It consists of 28000 atoms - a 10^3 supercell with 28 atoms of iron silicate (Fe_2SiO_4, also known as Fayalite) per unit cell. The simulation employs a classical potential (Morse with a hard-core repulsive term and 5.5 Å cutoff) with long-range electrostatics using Smoothed Particle Mesh Ewald (SPME) summation. While CP2K does support classical potentials via the Frontiers In Simulation Technology (FIST) module, this is not a typical calculation for CP2K but is included to give an impression of the performance difference between machines for the MM part of a QM/MM calculation. 36 processors were used.
Case3 characteristics:
This is a single-point energy calculation using Quickstep GAPW (Gaussian and Augmented Plane-Waves) with hybrid Hartree-Fock exchange. It consists of an isolated cluster of 54 Si atoms in a 20x20x20 $Å^3$ cubic cell. These types of calculations are generally around one hundred times the computational cost of a standard local DFT calculation, although this can be reduced using the Auxiliary Density Matrix Method (ADMM) (as in this example). Using OpenMP is of particular benefit here as the HFX implementation requires a large amount of memory to store partial integrals. By using several threads, fewer MPI processes share the available memory on the node and thus enough memory is available to avoid recomputing any integrals on-the-fly, improving performance. In this test pure MPI was used. 36 processors were used. In the Scalasca analysis only the

	Metric name	fayalite.json	fayalite_io.json
Global	Total Time (s)	25	19
	Time IO (s)	0.32	0.13
	Time MPI (s)	23.20	15.03
	Memory vs Compute Bound	1.00	1.06
	Load Imbalance (%)	55.56	43.88
IO	IO Volume (MB)	197.73	197.73
	Calls (nb)	3113003	27519
	Throughput (MB/s)	616.35	1571.13
	Individual IO Access (kB)	0.06	7.70
MPI	P2P Calls (nb)	15352	15352
	P2P Calls (s)	0.34	0.33
	P2P Calls Message Size (kB)	0	0
	Collective Calls (nb)	8062	8062
	Collective Calls (s)	21.94	13.88
	Coll. Calls Message Size (kB)	2357	2357
	Synchro / Wait MPI (s)	21.79	13.72
	Ratio Synchro / Wait MPI (%)	92.74	90.32
Node	Time OpenMP (s)	N.A.	N.A.
	Ratio OpenMP (%)	N.A.	N.A.
	Synchro / Wait OpenMP (s)	N.A.	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.	N.A.
Mem	Memory Footprint	148336kB	122632kB
	Cache Usage Intensity	0.97	0.97
Core	IPC	0.66	0.66
	Runtime without vectorisation (s)	25	20
	Vectorisation efficiency	1.00	1.05
	Runtime without FMA (s)	25	20
	FMA efficiency	1.00	1.05

Table 35: Performance metrics for CP2K on the JURECA HPC system, Fayalite benchmark, case2, buffered I/O performance in contrast to default I/O behavior

MPI and hybrid potentials parts of the code were scanned.

Performance report

All benchmarking runs were executed on the JURECA system.
According to Table 34 CP2K, the given configuration spends lot of its MPI time within waiting procedures (nearly 40% of the MPI time). In the specific case most of these delays are created by `MPI_Waitany` and `MPI_Waitall` commands and collective communication like `MPI_Allreduce` and `MPI_Alltoallv` mostly due to load balancing problems.
In some situations process zero is doing extra work, like performing I/O as shown in the `Vampir` overview in Figure 36. This problem is also seen in the second test case Table 35. Here the `Ratio Synchro / Wait MPI` is even worse, as 90% of the MPI time is spend for waiting and 90% of the whole program time is MPI time. This behavior was mostly stressed by the benchmark case, were data is written at each time step to investigate this particular case. The number of I/O calls is quite high while the individual access size is very low. This shows a big bottleneck on the I/O site. The I/O time seems to be quite low, but the problem here is the master worker writing scheme, were only the master is writing the data to disk. All other processes are waiting within pending collective operations as shown in Figure 37. The master process is using ASCII output and is only writing small chunks of data. Enabling the Fortran I/O buffer by setting `FORT_BUFFERED=true` could already lower the number of I/O calls and reduces the runtime by 30%. This is shown in the last column of Table 35. A more effective solution would be to switch to a different output format, a more effective writing procedure or asynchronous I/O.

	Metric name	hybrid.json
Global	Total Time (s)	25
	Time IO (s)	0.00
	Time MPI (s)	11.11
	Memory vs Compute Bound	1.00
	Load Imbalance (%)	10.80
IO	IO Volume (MB)	2.54
	Calls (nb)	3506
	Throughput (MB/s)	3006.25
	Individual IO Access (kB)	1.15
MPI	P2P Calls (nb)	48776
	P2P Calls (s)	1.31
	P2P Calls Message Size (kB)	35
	Collective Calls (nb)	21173
	Collective Calls (s)	8.21
	Coll. Calls Message Size (kB)	13
	Synchro / Wait MPI (s)	9.11
	Ratio Synchro / Wait MPI (%)	80.72
Node	Time OpenMP (s)	N.A.
	Ratio OpenMP (%)	N.A.
	Synchro / Wait OpenMP (s)	N.A.
	Ratio Synchro / Wait OpenMP (%)	N.A.
Mem	Memory Footprint	384712kB
	Cache Usage Intensity	0.77
Core	IPC	2.25
	Runtime without vectorisation (s)	23
	Vectorisation efficiency	0.92
	Runtime without FMA (s)	25
	FMA efficiency	1.00

Table 36: Performance metrics for CP2K on the JURECA HPC system, HYBRID XC-potential benchmark, case3

In the SiH benchmark there are also program parts which only instrument a subset of the available processes (Figure 38). This setting based on a process reduction in the `cp_fm_diag.cp_fm_syevd` routine where the number of active processes is lowered (in this case to 16 processes instead of 36) based on the used problem size to avoid additional communication.
Lots of time is also spend in the creation of MPI communicators as there are a lot of communicators involved. There might be some room for improvements to keep communicators as long as possible to avoid recreation (which also needs a collective operation every time).
The simple vectorization test only shows basic vectorization efficiency, which is influenced by the large amount of MPI time in contrast of the total program execution time.
The last benchmark case in Table 36 also highlights the MPI delays. Here the problem is mostly triggered by a load imbalance in the calculation part of the code. The largest load imbalance problem is seen in Figure 39 in the function `integrate_four_center`.

Figure 36: MPI delay due to I/O on master for SiH benchmark case

Figure 37: MPI delay due to I/O on master for fayalite benchmark case

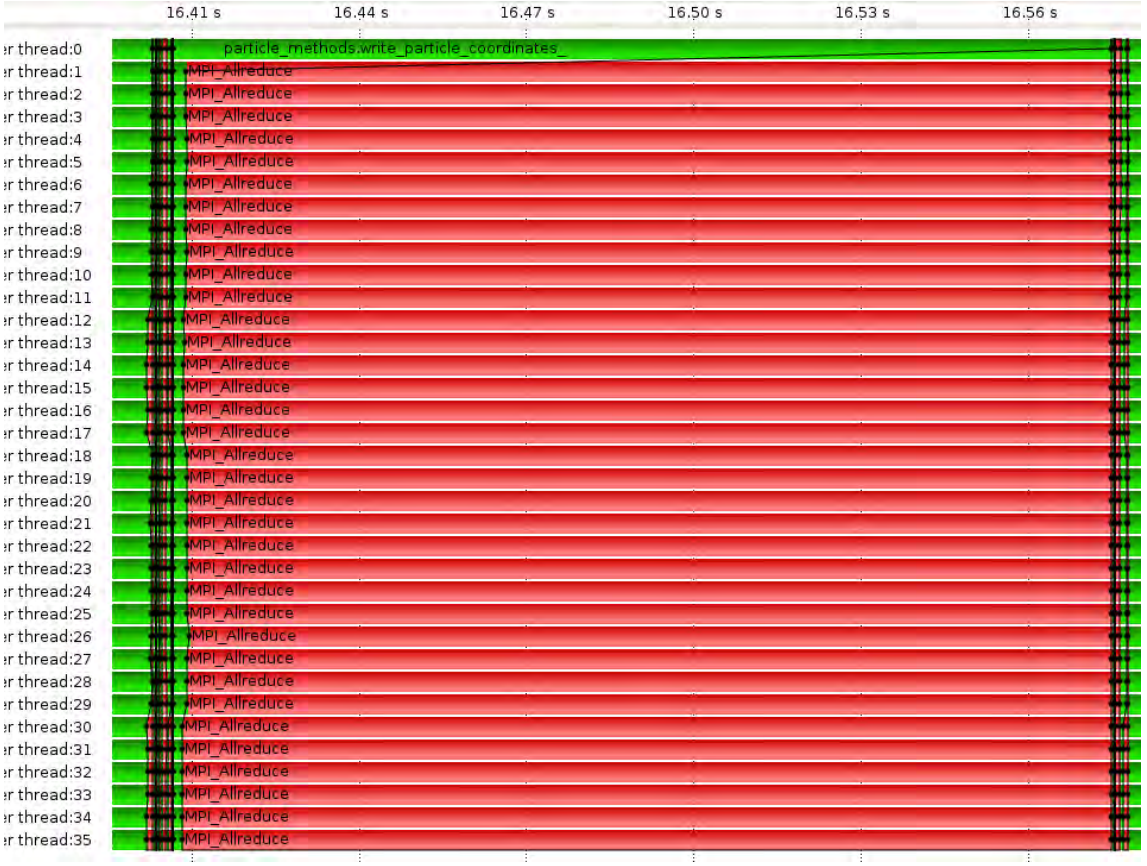


Figure 38: Processes used within the SiH benchmark case

Figure 39: Load imbalance problem in the HYBRID XC-potential benchmark case

A.21 DIVA
Code ID card

Code name	DIVA
Scientific domain	Seismic Wave Propagation
Description	DIVA (Depth Imaging & Velocity Analysis) is a seismic wave propagation code that implements RTM and FWI algorithms using finite differences method. It implements multiple propagators associated to different wave equations (Acoustic/Elastic, variable density...). Here we will only work on modelling which computes the signal received by a set of receivers following a signal sent by a source and taking into account the ground physics (propagation velocity, pressure, anisotropy...).
Languages	Fortran90 (200k lines, 1K lines for modeling)
Library dependencies	MPI, OpenMP.
Programing models	MPI, OpenMP.
Platforms	Pangea – Total Exploration and Production (cf. TOP500)
Scalability results	It has been ported on X86 architectures, scaling results are good up to 2000 cores per shot.
Typical production run	1 week on 20k cores for production.
Input / Output requirement	• Size: 10 GB / 24h run • Single post-processing output: 50 MB • Single restart output: 50 MB
Application references	
Contact	• Elies Bergounioux (Elies.Bergounioux@total.com) • Xavier Lacoste (Xavier.Lacoste@total.com)
Main bottleneck	Memory access
Relevant kernel algorithm	Finite differences
Software Licence	none

Performance metrics

Code team:

- Xavier Lacoste (Total) for code developer
- Abel Marin-Laflèche for WP1

Case1 characteristics:

Domain size	382 x 381 x 321 regular grid for each shot.
Resources	1 node on Jureca (24 cores)
IO details	No checkpoint written.
Type of run	development run.

Case2 characteristics:

Domain size	952 x 951 x 801 regular grid for each shot.
Resources	8 node on Jureca (128 cores (8x16 because the load balancing should perform better on power of 2 number of processes : Processors' grid 2x8x8))
IO details	No checkpoint written.
Type of run	development run.

	Metric name	Case 1	Case 2
Global	Total Time (s)	141	949
	Time MPI (s)	57.23	527.40
	Memory vs Compute Bound	1.38	1.24
	Load Imbalance (%)	35.68	53.09
MPI	P2P Calls (nb)	60565	209390
	P2P Calls (s)	52.62	512.84
	P2P Calls Message Size (kB)	295	574
	Collective Calls (nb)	862	683
	Collective Calls (s)	4.19	9.26
	Coll. Calls Message Size (kB)	5470	1183
	Synchro / Wait MPI (s)	48.68	489.27
	Ratio Synchro / Wait MPI (%)	77.36	90.31
Mem	Memory Footprint	408000kB	728480kB
	Cache Usage Intensity	0.74	0.84
Core	IPC	1.81	1.91
	Runtime without vectorisation (s)	207	1561
	Vectorisation efficiency	1.47	1.64
	Runtime without FMA (s)	141	970
	FMA efficiency	1.00	1.02

Table 37: Performance metrics for DIVA on the JURECA HPC system

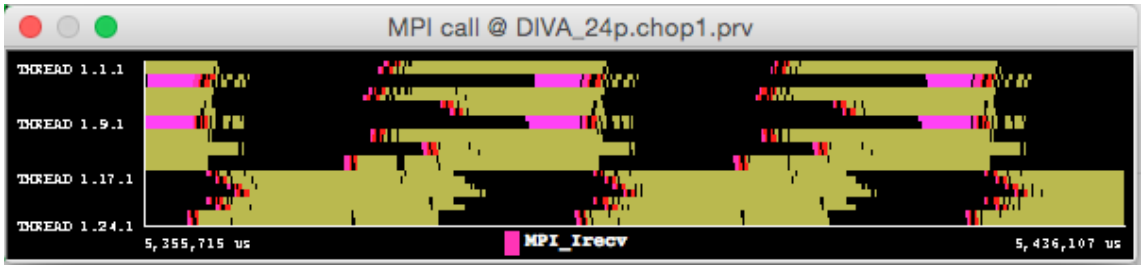
Performance report

According to Table 37, DIVA suffers from an important load imbalance on the tested runs.
When looking at the trace (Figure 40) we confirmed that load impalance. This can be explained by the fact that the border domains include PML computations which imply a heavier workload than in the inner domain.
DIVA includes a load balancing grid recalculation that fails with this relativly small grid.
The MPI Call profile on Figure 41 shows that a lot of time is spent in communications except for 4 processes.
The load imbalance is confirmed on Figure 42. We can see that 2 groups of 4 processes do more work than the others. These must correspond to the top and bottoms domain. Indeed the processors grid id 2x2x6. And thus, the 4 processes at top and bottom have more work than the others.
Using Scalasca (Figure 43), one can confirm that nearly half of the time is spent waiting for communications, in `MPI_Wait_any()` function, and that 4 processes spend less time in this function. The other part of the time is spent in the computational kernel.
As a conclusion, in order to improve DIVA, we would recommend the following roadmap:

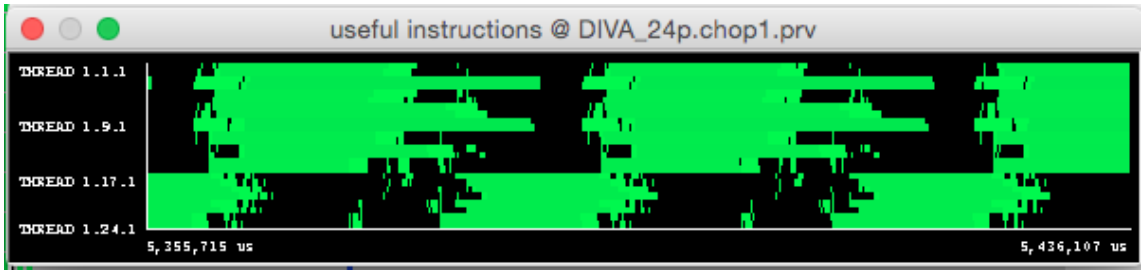
1. Create more domains than MPI processes to help handling load imbalance more easily.

2. Separate inner computation from each domain from the ghost computation. The inner computation can be computed during communications and thus hide them.

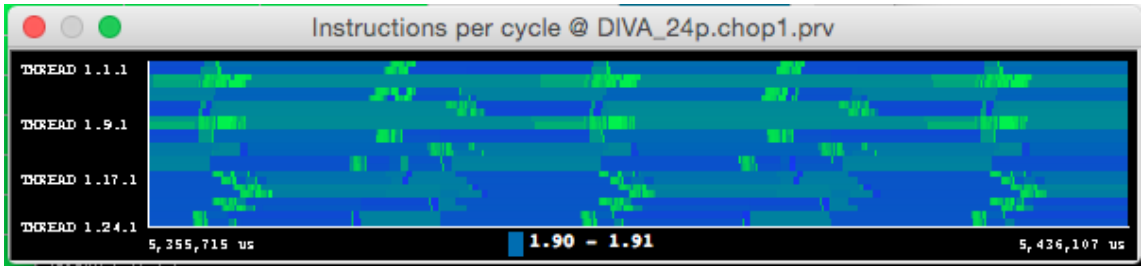
3. Add and OpenMP parallelisation, either in the kernel or on subdomains to improve load-balance inside a node.



(a) MPI Call



(b) Usefull instructions



(c) Instructions per cycle

Figure 40: Paraver trace on 1 node

MPI call profile @ DIVA_24p.chop1.prv

	Outside MPI	MPI_Isend	MPI_Irecv	MPI_Reduce	MPI_Allreduce	MPI_Waitany
THREAD 1.1.1	49.17 %	0.25 %	0.16 %	0.14 %	-	50.28 %
THREAD 1.2.1	48.62 %	0.21 %	0.17 %	0.00 %	-	51.00 %
THREAD 1.3.1	84.78 %	0.34 %	1.26 %	0.00 %	11.74 %	1.89 %
THREAD 1.4.1	85.51 %	0.30 %	1.23 %	0.00 %	10.97 %	1.99 %
THREAD 1.5.1	48.08 %	0.33 %	0.21 %	0.05 %	-	51.33 %
THREAD 1.6.1	48.05 %	0.36 %	0.24 %	0.00 %	-	51.36 %
THREAD 1.7.1	66.51 %	0.37 %	0.23 %	0.00 %	-	32.89 %
THREAD 1.8.1	67.66 %	0.35 %	0.23 %	0.00 %	-	31.76 %
THREAD 1.9.1	83.05 %	0.28 %	0.69 %	0.00 %	13.18 %	2.79 %
THREAD 1.10.1	83.97 %	0.29 %	0.69 %	0.00 %	12.32 %	2.74 %
THREAD 1.11.1	48.09 %	0.24 %	0.16 %	0.00 %	-	51.52 %
THREAD 1.12.1	48.03 %	0.24 %	0.17 %	0.00 %	-	51.55 %
THREAD 1.13.1	53.40 %	0.36 %	0.72 %	0.00 %	-	45.51 %
THREAD 1.14.1	53.69 %	0.36 %	0.70 %	0.00 %	-	45.24 %
THREAD 1.15.1	40.77 %	0.24 %	0.68 %	0.00 %	-	58.31 %
THREAD 1.16.1	40.87 %	0.24 %	0.66 %	0.00 %	-	58.24 %
THREAD 1.17.1	42.36 %	0.22 %	0.15 %	0.02 %	-	57.26 %
THREAD 1.18.1	42.01 %	0.19 %	0.16 %	0.00 %	-	57.63 %
THREAD 1.19.1	42.66 %	0.28 %	0.19 %	0.01 %	-	56.87 %
THREAD 1.20.1	42.52 %	0.27 %	0.17 %	0.00 %	-	57.04 %
THREAD 1.21.1	43.62 %	0.26 %	0.18 %	0.01 %	-	55.94 %
THREAD 1.22.1	43.48 %	0.26 %	0.19 %	0.00 %	-	56.07 %
THREAD 1.23.1	41.39 %	0.18 %	0.16 %	0.00 %	-	58.28 %
THREAD 1.24.1	41.45 %	0.17 %	0.15 %	0.00 %	-	58.24 %
Total	1,289.73 %	6.56 %	9.55 %	0.23 %	48.21 %	1,045.73 %
Average	53.74 %	0.27 %	0.40 %	0.01 %	12.05 %	43.57 %
Maximum	85.51 %	0.37 %	1.26 %	0.14 %	13.18 %	58.31 %
Minimum	40.77 %	0.17 %	0.15 %	0.00 %	10.97 %	1.89 %
StDev	15.33 %	0.06 %	0.34 %	0.03 %	0.81 %	19.71 %
Avg/Max	0.63	0.74	0.32	0.07	0.91	0.75

Outside MPI

Figure 41: Paraver : MPI Call Profile on 1 node

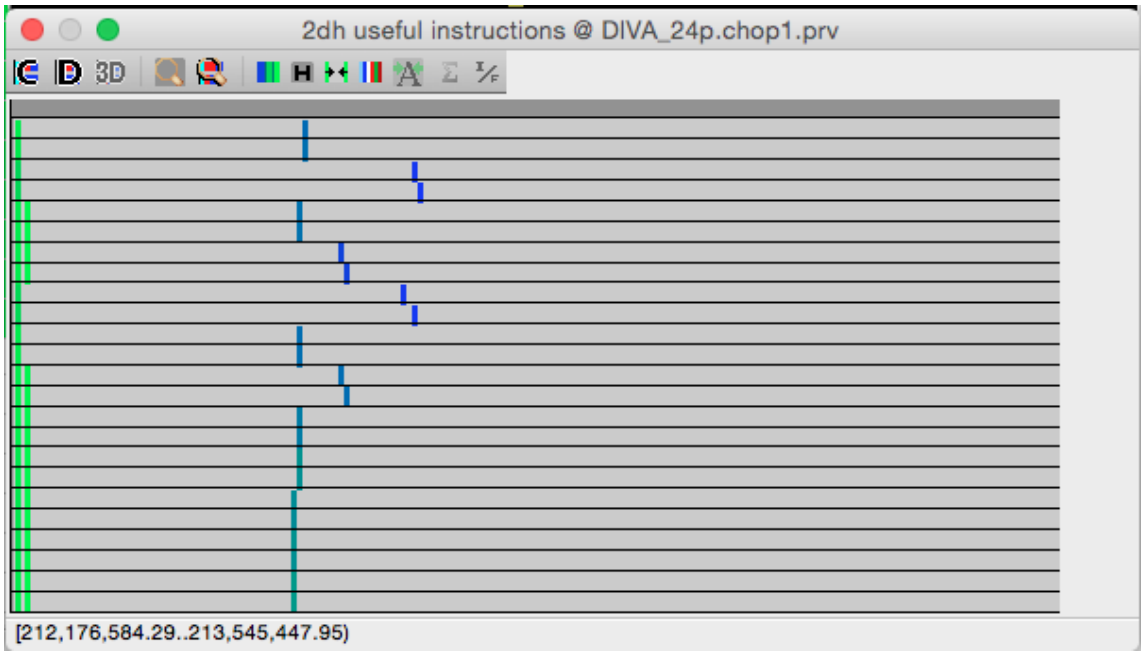


Figure 42: Paraver : Usefull Instruction diagramm on 1 node

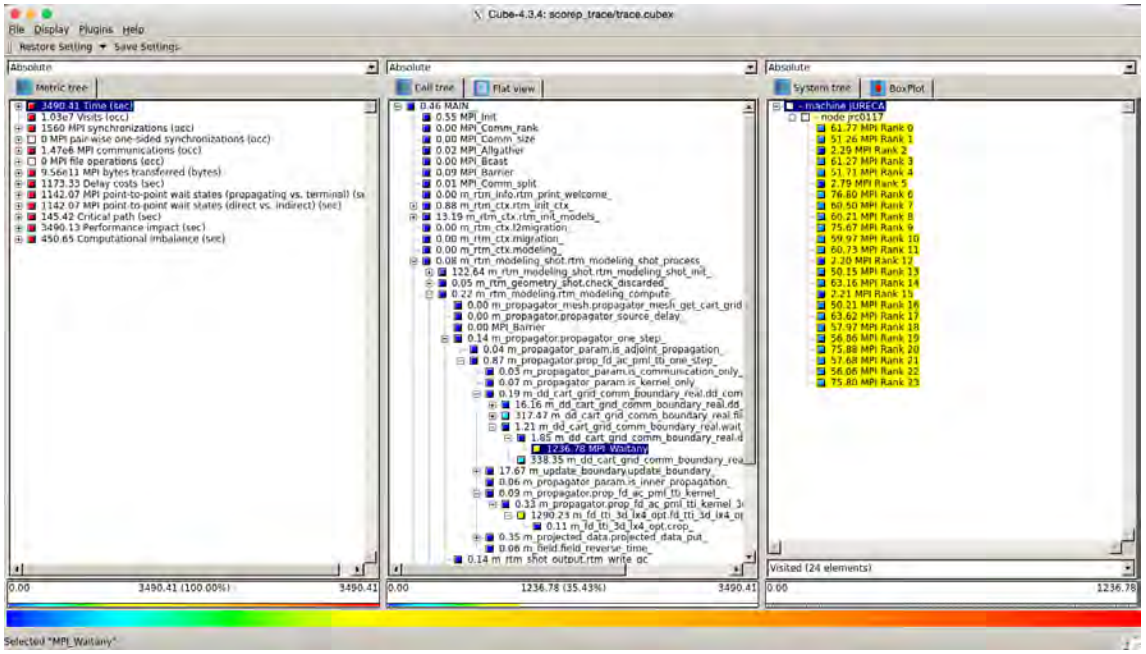


Figure 43: Scalasca : calling tree on 1 node

References

[1] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] Patrick R Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing*, 32(2):136–156, 2006.

[3] P Tamain, Hugo Bufferand, Guido Ciraolo, C Colin, D Galassi, Ph Ghendrih, Frédéric Schwander, and Eric Serre. The tokam3x code for edge turbulence fluid simulations of tokamak plasmas in versatile magnetic geometries. *Journal of Computational Physics*, 321:606–623, 2016.