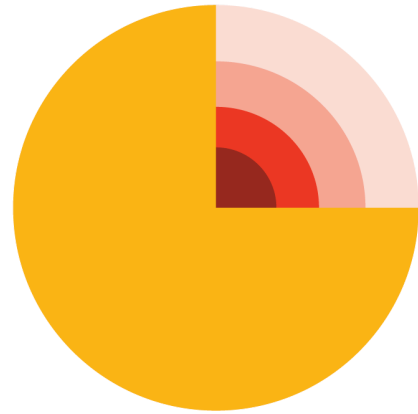




ChEESE

Deliverable D2.1 First code audit report



ChEESE-2P Deliverable

D2.1 First code audit report

Document Information

Contract Number	101093038
Project Website	https://cheese2.eu
Contractual Deadline	31/12/2023 (M12)
Activity	WP2
Dissemination Level	PUBLIC
Nature	R

I. Copyright

Copyright Members of ChEESE collaboration, 2023/2026.

II. Delivery Slip

Role	Name	Partner/Activity
Author, editor	José Gracia	USTUTT-HLRS
Code chapter authors	All code owners	Various
Reviewer 1	Piero Lanucara	CINECA
Reviewer 2	Michael Bader	TUM

III. Version control

Version	Date	Description of change	Author/Partner
V2	31/01/2024	Submitted to EC	

IV. Table of Contents

1. Executive summary.....	6
2. Ambition and lessons learnt.....	7
2.1. Access to EuroHPC systems.....	8
2.2. Availability of performance analysis tools.....	9
2.3. Trace volume.....	10
3. Code Elmer.....	11
3.1. Brief description of code and use-case.....	11
3.2. Scaling behavior.....	11
3.3. Performance data acquisition.....	13
3.4. POP performance efficiency metrics.....	14
3.5. Communication.....	15
3.6. Computational scalability.....	17
3.7. Observations and recommendations.....	17
3.8. References.....	17
4. Code SPECFEM3D.....	18
4.1. Brief description of code and use-case.....	18
4.2. Scaling behavior.....	18
4.3. Performance data acquisition.....	20
4.4. POP performance efficiency metrics.....	20
4.5. Communication.....	21
4.6. Computational scalability.....	23
4.7. Observations and recommendations.....	24
5. Code tandem.....	25
5.1. Brief description of code and use-case.....	25
5.2. Scaling behavior.....	25
5.3. Performance data acquisition.....	27
5.4. POP performance efficiency metrics.....	28
5.5. Communication.....	28
5.6. Computational scalability.....	28
5.7. Observations and recommendations.....	29
5.8. References.....	30
6. Code SeisSol.....	31
6.1. Brief description of code and use-case.....	31
6.2. Scaling behavior.....	31
6.3. Performance data acquisition.....	34
6.4. POP performance efficiency metrics.....	34

7. Code Tsunami-HySEA.....	36
7.1. Brief description of code and use-case.....	36
7.2. Scaling behavior.....	37
7.3. Performance data acquisition.....	38
7.4. POP performance efficiency metrics.....	38
7.5. Communication.....	39
7.6. Computational scalability.....	40
7.7. Observations and recommendations.....	40
8. Code LaMEM.....	42
8.1. Brief description of code and use-case.....	42
8.2. Scaling behavior.....	43
8.2.1. Model setup.....	43
8.2.2. Weak scalability.....	44
8.2.3. Strong scaling.....	47
8.2.4. Discussion.....	49
8.3. Performance data acquisition.....	51
8.4. POP performance efficiency metrics.....	52
9. Code OpenPDAC.....	54
9.1. Brief description of code and use-case.....	54
9.2. Scaling behavior.....	55
9.3. Performance data acquisition.....	56
9.4. POP performance efficiency metrics.....	57
10. Code FALL3D.....	58
10.1 Brief description of code and use-case.....	58
10.1.1. Use case scaling.....	59
10.1.2. Use case audit.....	60
10.2. Scaling behavior.....	62
10.3. Performance data acquisition.....	64
10.4. POP performance efficiency metrics.....	65
10.5. Communication.....	67
10.6. Computational scalability.....	68
10.7. Observations and recommendations.....	69
11. Code pTatin3D.....	71
11.1. Brief description of code and use-case.....	71
11.2. Scaling behavior.....	72
11.3. Performance data acquisition.....	72
11.4. POP performance efficiency metrics.....	73
12. Code ExaHyPE 2.....	75
12.1. Brief description of code and use-case.....	75

12.2. Scaling behavior.....	75
12.3. Performance data acquisition.....	77
12.4. POP performance efficiency metrics.....	78
12.5. Communication.....	79
12.6. Computational scalability.....	79
12.7. Observations and recommendations.....	79
Appendix: POP performance efficiency metrics.....	80
A.1. Pure-MPI metrics.....	80
A.2. Hybrid environment metrics.....	82
A.2.1. Multiplicative approach.....	82
A.2.2. Additive approach.....	83

1. Executive summary

This document describes the result of the first performance assessment campaign of the project ChEESE-2P. The campaign serves three purposes: (1) identify performance bottlenecks to guide optimisation work, (2) set a performance baseline to monitor code evolution throughout the project duration, and (3) produce and document key performance indicators (KPI) related to the progress on flagship codes.

Conducting such a performance assessment campaign for all of ChEESE's 11 flagship codes has been quite challenging. In chapter 2, we outline from a high-level perspective our original ambition, issues encountered, and lessons learnt.

In collaboration with partners from the POP collaboration¹, performance assessments have been started for all 11 flagship codes. Doing such assessments requires a number of steps: deploy performance analysis tools, instrument the code to extract performance data (traces), prepare use-cases, do actual code runs, analyze the performance traces, and report. Some of these steps might need to be iterated to satisfaction. All in all, this can be a very time-consuming process.

At the time of writing this document, 10 of those assessments have advanced sufficiently to produce at least some performance metrics. The results of these assessments are reported in chapters 3 to 12, one for each code. Out of those 10, 6 assessments can be considered completed, and 4 are preliminary in the sense that only a subset of metrics has been produced or some part of the analysis is still underway. Therefore, the amount of detail and extent of the material covered varies across chapters.

Finally, the last chapter 13 briefly summarizes the performance metrics which have been used throughout this document.

¹ <https://pop-coe.eu/>

2. Ambition and lessons learnt

Unlike other CoEs, ChEESE-2P relies on a relatively large number of so-called flagship codes. The objective of work package 2 "Exascale technical challenges in flagship codes" is, amongst others, to port 11 flagship codes to EuroHPC (pre-)exascale machines, as well as to optimize the codes on these platforms according to the requirements defined by pilot demonstrators (WP4) and scientific grand challenges (WP5).

Following best-practices, code optimisations in ChEESE-2P will be guided by quantitative assessment of the performance of the particular use-cases of interest on the actual target platform. In the scope of implementing a project like ChEESE-2P, these performance assessments serve three purposes: (1) identify performance bottlenecks to guide optimisation work, (2) set a performance baseline to monitor code evolution throughout the project duration, and (3) produce and document key performance indicators (KPI) related to the progress on flagship codes.

In order to compare the performance of different codes versions objectively and quantitatively, we have chosen to employ a uniform methodology for all flagship codes. In particular, our approach consists of:

1. Conducting two performance assessments for each code:
 - a. First assessment early in the project with the initial version of the code to determine baseline performance and major bottlenecks.
 - b. Second assessment towards the end of the project with the final version of the code to determine KPI related to code improvement.
2. Using a quantitative, well-defined set of performance metrics, such as the POP Standard Metrics for Performance Analysis of Hybrid Parallel Applications² (POP metrics for short). See Chapter 13 for a brief summary of POP metrics.
3. Doing the assessments on the target EuroHPC system at scale whenever possible, rather than on small institutional clusters.
4. Doing first and second assessment with the same scientific use case on the same EuroHPC system. Note, that it is admissible and even desired to change number of nodes, etc., and even switch between CPU and GPU version of a code between assessments.

2

<https://pop-coe.eu/further-information/learning-material/pop-standard-hybrid-metrics-for-parallel-performance-analysis>

For reasons of project management, we have decided to do performance assessment *campaigns*, i.e., assess the performance of all codes more or less at the same time, as opposed to allow each code owner to follow her own pace.

In hindsight, organizing and implementing such performance assessment campaigns for 11 flagship codes has been quite challenging, in particular due to

1. difficulties in managing access to EuroHPC resources,
2. the availability and fragility of development and performance analysis tool chains,
3. the sheer volume of performance analysis data.

2.1. Access to EuroHPC systems

Ideally, ChEESE-2P as a whole would apply for compute resources to all EuroHPC systems and distribute the allocated budget across work packages and participants across the duration of the project. In reality, each flagship code owner had to apply individually for access which was then granted for a limited time only. In many cases, the duration of these allocations was not sufficient to port the code, setup assessment use-cases, and prepare the code for integration with the performance analysis tools, and finally do the actual performance assessment runs.

In addition, at the time of writing the proposal, ChEESE-2P had planned to focus on three systems: Leonardo (classical GPUs with CUDA), LUMI (AMD APUs), and Mare Nostrum 5 (classical GPUs). Leonardo was of particular interest, as CUDA was widely available and porting of the codes relatively well understood. On the other hand, LUMI's architecture (AMD APUs) was not widely available and requires significant porting effort. At the start of project, only LUMI was available. Leonardo has become fully operational and accessible through the PRACE calls website³ at the beginning of the second half of the year. Finally, Mare Nostrum is still not available. Also, both Leonardo and LUMI had prolonged down-times in the second half of the year which collided with the assessment campaign planning.

³ <https://pracecalls.eu/>

As a result, access to EuroHPC caused additional effort and serious delays for this performance assessment task. In hindsight, we would plan with significant longer delays from third parties. In particular, we will reconsider conducting the campaign in lock-step across all codes. Conducting the campaign asynchronously and for each code independently from the start might have reduced the management effort significantly.

2.2. Availability of performance analysis tools

Performance analysis relies on the availability of a number of tools. Due to the expertise amongst the ChEESE partners, we have chosen to employ one or more of mpiP⁴, Extrae+Paraver⁵, Score-P+Scalasca^{6,7} and Talp/DLB⁸. None of the EuroHPC systems offered a reasonable subset of these. On the other side, some of these tools are highly specific to the particular architecture, the available parallel programming libraries, and compilers. In response, ChEESE staff deployed customized installations of the tools on Leonardo. On LUMI, it was not possible to share disk space across different compute projects, and we provided detailed step-by-step instructions on installing mpiP, Extrae and Talp/DLB.

However, we had to learn that merely succeeding to install a copy of the tools on a given machine is not enough. In practice, we discovered (often after long time-consuming trial-and-error) that many combinations of compiler, flagship code, GPU libraries, etc, do not work reliably for the programming models of interest. For instance, one of our codes uses Pthreads, OpenMP target offloading and CUDA graphs through the NVIDIA compiler all knit together by modern C++23. None of these requirements is a problem for the tools in principle when used in isolation. However, in its combination, we still did not manage to find a robust tool chain which allowed us to instrument the code and record performance analysis traces.

The lesson learnt is that development tool chains are fragile and break in many unexpected ways. One cannot assume that a particular combination is portable to other systems and environments.

⁴ <https://github.com/LLNL/mpiP>

⁵ <https://tools.bsc.es/>

⁶ <https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-4.1/html/>

⁷ <https://www.scalasca.org/scalasca/software/scalasca-2.x/>

⁸ <https://pm.bsc.es/dlb>

2.3. Trace volume

While the two former challenges have a logistical or organizational nature, the biggest challenge for the performance analysis as such has been the sheer size of the performance analysis data (commonly referred to as performance *traces*). Roughly half of the assessment use cases initially produced traces in the order of many tens or even hundreds of Gigabytes. Determining the full set of performance metrics for such large traces can be prohibitively expensive and require more compute time than the actual code execution run.

Where possible, we have employed various techniques to reduce the volume of traces, but in some cases had to sacrifice detail in the performance analysis (as for instance pruning a subset of performance metrics). In other cases, the analysis did not complete in time for this deliverable and we present only preliminary results or result pertaining to a limited set of aspects.

One of the most efficient techniques to reduce trace volume is to mark regions of interest in the code and record traces only for those regions. However, this approach fails if the code uses external numerical libraries and frameworks such as PETSc or OpenFOAM. In contrast to libraries such as BLAS, which offer low-level operations of short duration and small computational loads, PETSc and OpenFOAM (and many others) offer high-level interfaces which drive whole numerical algorithms and solvers. The user code may consist of a few calls into the numerical library which then do essentially all computation and communication without being visible to the developer/user; essentially the framework is a black box. We have seen cases where a single call to a PETSc routine resulted in driving whole multi-grid loop cycles producing millions of MPI operations in hundreds of thousands of repetitive sub steps.

In this situation, marking the entry and exit points to the library calls will not reduce trace volume. We would rather need ways to mark regions of interest inside the libraries in a way that only one (or a few) instances of repetitive operations are recorded.

3. Code Elmer

3.1. Brief description of code and use-case

Elmer is an open source multiphysical simulation software which includes physical models of fluid dynamics, structural mechanics, electromagnetics, heat transfer and acoustics, and beyond. These are described by partial differential equations which Elmer solves by the Finite Element Method (FEM).

This use case is based on a test-suite⁹ that was run in summer 2023 in order to inquire optimal combination of pre-conditioners and Krylov methods for a Poisson-type of problem of about 6 Mio DOFs (applied mesh multiplication level of 2), ranging from as little as 6000 DOFs/MPI rank (at 1024 ranks on 8 nodes) to 47000 (at 128 ranks on single node). The example uses a solver¹⁰ that is already optimized for modern CPUs utilizing the vector units via OpenMP SIMD. It utilizes mesh-coloring in order to avoid race conditions for SIMD operations (Byckling et al., 2017) which will be utilized in the planned OpenMP target offload implementation of the solver. The goal in ChEESE is not to optimize this solver further for CPU (as turned out in the audit, there is little to nothing to do) but rather optimize it by utilization of accelerators in both the assembly and the linear solution step. Based on this prototype solver, we will strive to implement OpenMP target offload to all needed components to operate the pilot demonstrator (PD9) in ChEESE completely on LUMI-G nodes.

3.2. Scaling behavior

Elmer provides execution times for solvers (wall clock and CPU). Looking at those (Figure 3-1), we see a more or less linearly falling value from 128 to 512 MPI tasks, but more or less a level-in from 512 to 1024 MPI tasks. Perhaps better visualization (Figure 3-2) is by plotting the CPU-time times the number of tasks (which should be a constant value in case of linear scaling).

⁹ <https://github.com/ElmerCSC/elmer-linsys>

¹⁰

https://github.com/ElmerCSC/elmer-linsys/blob/cheese/Poisson/WinkelUnstructured-ChEESE/ModelPDEevol_fixedparams.F90

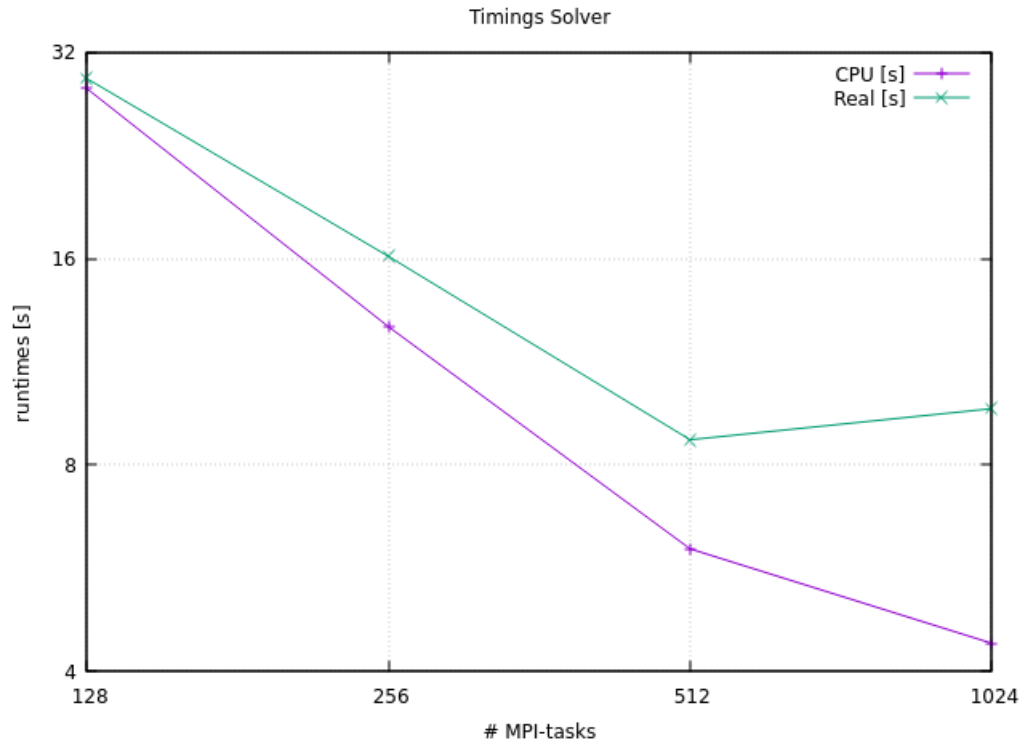


Figure 3-1 Runtime of the solver (wall-clock time in green, CPU time in red) in relation to the amount of MPI ranks

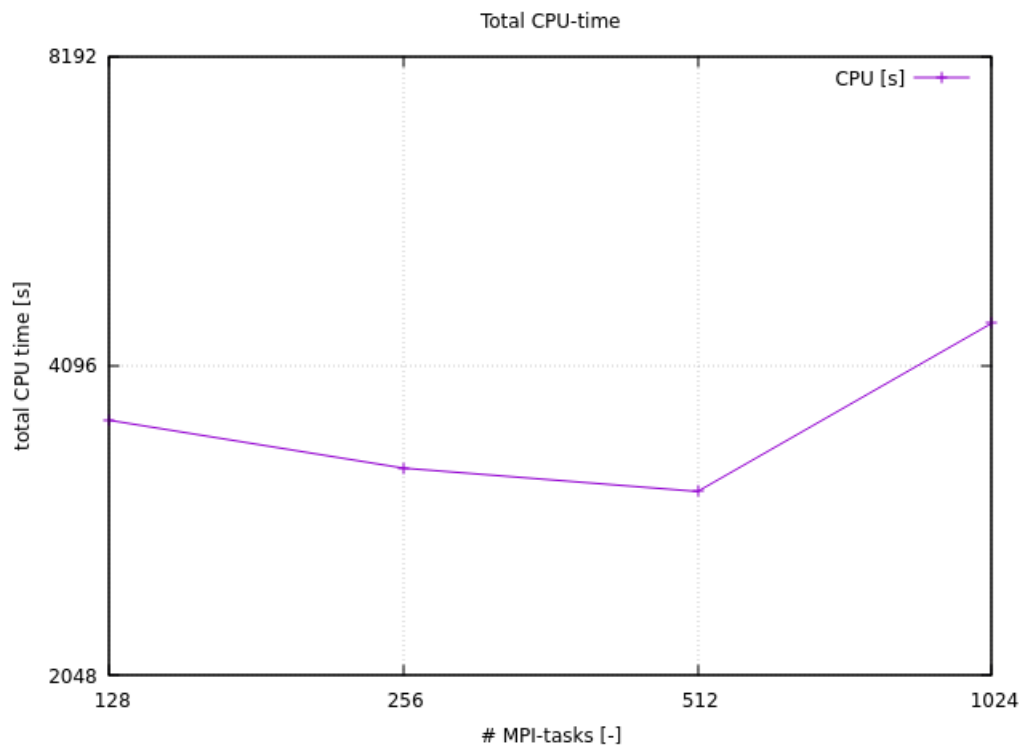


Figure 3-2 Solver CPU time multiplied with the amount of tasks (i.e., total CPU time) spent in solution process over MPI ranks

From Figure 3-2 one can see that there is a slight super-linear scaling between 128 and 512 tasks, but beyond that the total CPU time is increasing. This most likely is due to an increasing imbalance between computation and communication (i.e., would kick in later with a larger problem size), hence should not impose too much of a problem for weak scalability.

From the total amount of needed iterations (Figure 3-3) for the applied solver method, one can see that generally there is a trend of larger iteration number needed when distributing the work to an increased number of MPI-tasks.

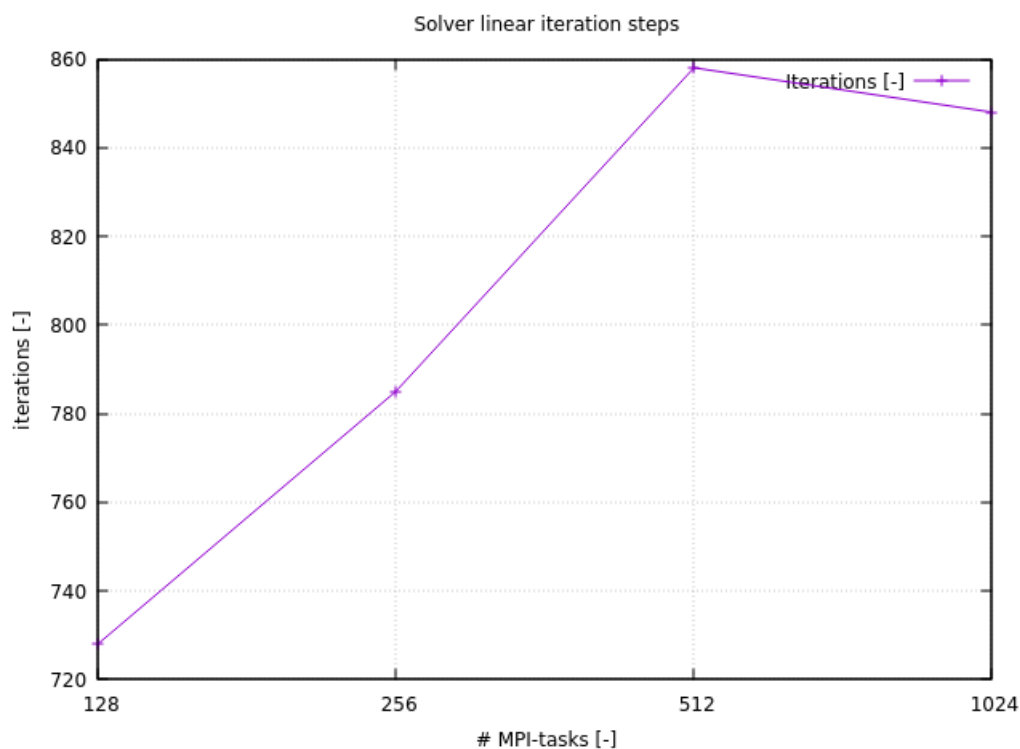


Figure 3-3 Needed iterations for the IDRS Krylov solution in relation to the number of MPI ranks

3.3. Performance data acquisition

The code owners have provided Extrae traces for runs with 128, 256, 512, and 1024 MPI ranks respectively. The traces were recorded with a circular buffer to limit their size. The traces do not contain hardware counters.

A first inspection of the traces showed a strictly repetitive pattern which we will refer to as *iteration*. These iterations are clearly separated from

each other by an *MPI_Barrier* operation. All iterations seem to have the same duration.

The quantitative analysis presented below is limited to a *Focus-of-analysis* (FoA) consisting of 40 such iterations (see illustration in Figure 3-4 below). For certain aspects we had to further limit trace volume to a single iteration (FoA-1IT).

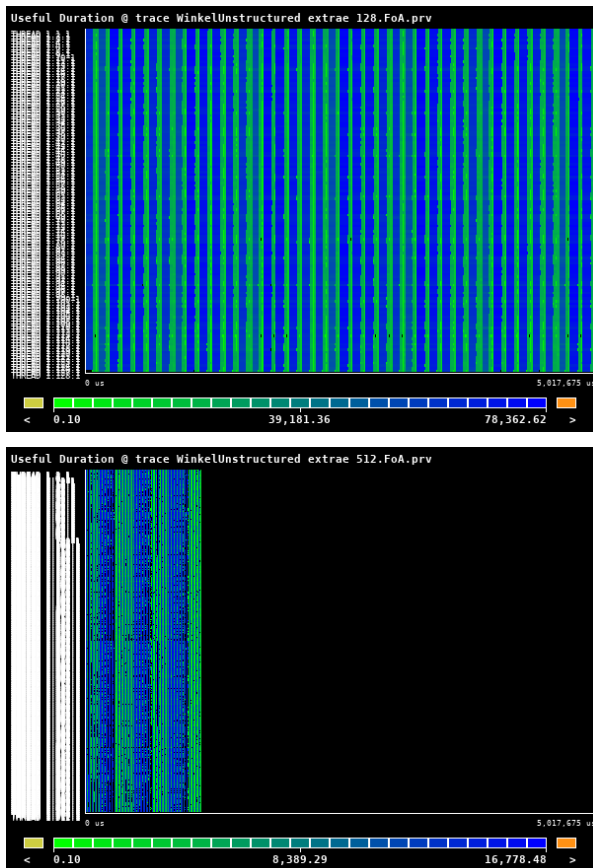


Figure 3-4 FoA of a run with 128 ranks and 512 ranks respectively. The colors indicate the duration of compute bursts between two MPI operations. The repetitive behavior is clearly visible.

3.4. POP performance efficiency metrics

We have calculated POP performance efficiency metrics using the simple (MPI-only) model (Table 3-1). Note, that most metrics are presented for the FoA. The metrics serialization efficiency and transfer efficiency had to be estimated from FoA-1IT; we have done this only for 128 and 1024 MPI processes.

Table 3-1 POP performance efficiency metrics

	MPI Processes			
	128	256	512	1024
Global efficiency	93%	98%	103%	111%
Parallel efficiency	93%	91%	89%	81%
Load balance	95%	93%	92%	87%
Communication efficiency	98%	98%	97%	94%
Serialization efficiency	99%			98%
Transfer efficiency	99%			95%
Computational Scalability	100%	108%	116%	137%

At 128 ranks, all metrics are above 90% and therefore very good.

Parallel efficiency is mainly limited by load balance (95%) while communication efficiency is excellent at 98%. While increasing the number of MPI ranks, load balance drops gradually to 87% at 1024 ranks, which is still a good value. At the same time communication efficiency drops to 94% which is still very good.

Due to the high communication efficiency, it cannot be clearly separated into either one of its components, i.e., serialization or transfer efficiency. At the highest scale transfer efficiency seems to be lower than serialization efficiency.

Interestingly, computational scalability increases with the number of ranks, i.e., actual computations (as opposed to MPI communication) take less and less time. Such behavior is most likely due to a better cache utilization due a decreasing problem size. The effect is so large, that the decrease in parallel efficiency is overcompensated and we observe super-linear scaling of the code for this particular problem and range of MPI ranks.

3.5. Communication

In general, MPI communication is not a performance issue with the application. Communication efficiency is high even at the largest scales. Figure 3-5 below illustrates the structure of MPI calls of a single iteration FoA-1IT. The amount of time spent in MPI operations is shown in Table 3-2 below.

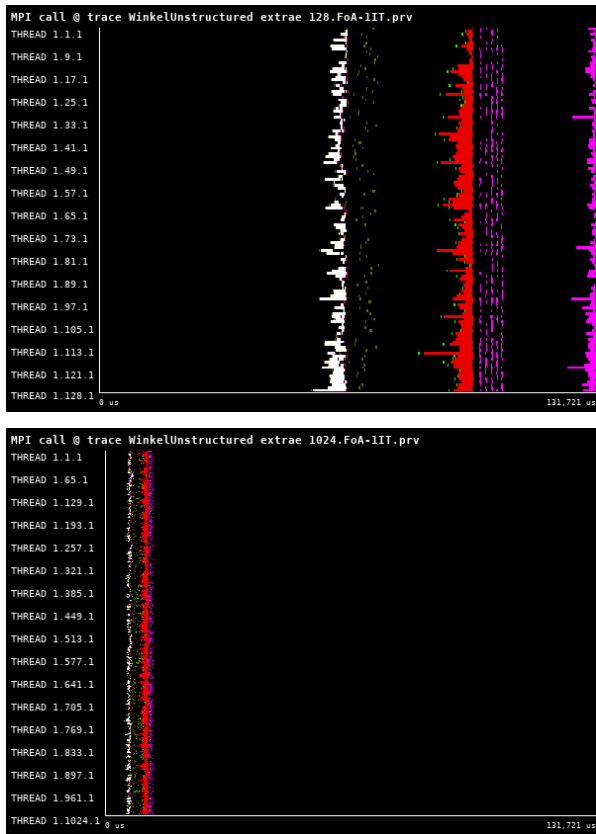


Figure 3-5 MPI call structure of FoA-1IT for a run with 128 ranks and 1024 ranks, respectively. MPI operations are shown in color: MPI_Recv in white, MPI_Barrier in red, MPI_Allreduce in purple.

Table 3-2 Fraction of total execution time of the FoA spent in MPI operations

	MPI Processes	
	128	1024
Outside MPI	93.2%	81.3%
MPI_Barrier	3.0%	10.2%
MPI_Recv	2.1%	3.9%
MPI_Allreduce	1.4%	3.0%
MPI_Bsend	0.3%	1.1%
MPI_Waitall	0.1%	0.4%
MPI_Irecv	0.0%	0.1%

For the FoA, most of the time is spent in *MPI_Barrier*, followed by *MPI_Recv* and *MPI_Allreduce* with similar fraction, and finally *MPI_Bsend* with a much smaller fraction. The fraction of time spent in MPI operations increases with the number of MPI ranks. In particular, the time spent in *MPI_Barrier* increases from 3% at 128 ranks to 10% of the total execution time at 1024 ranks.

There are also non-blocking operations *MPI_Irecv* and *MPI_Waitall* which however take negligible time. This means that progress in non-blocking operations is good.

3.6. Computational scalability

As seen in Section 3.4 above, the computational scalability of the code is above 100% even at the largest scale, i.e. actual computations (as opposed to MPI communication) takes less and less time. Such super-linear computational scaling is most likely due to a better cache utilization as the problem size decreases with increasing number of cores (strong scaling). However, we cannot prove this hypothesis since the traces did not record hardware counters such as the number of instructions. We have decided not to repeat tracing to obtain hardware counters, because computational scalability is not a performance bottleneck at all.

3.7. Observations and recommendations

The code Elmer is very efficient for the use-case under consideration, both in terms of parallelization as well as computational scalability. At 1024 MPI ranks, the biggest issue is a small load imbalance. The communication pattern (as measured by the serialization efficiency) is very good and will not pose a problem even at larger scales. Transfer efficiency starts to increase at larger scales due to the impact of an *MPI_Barrier* and several *MPI_Allreduce* operations.

Recommendations:

- Investigate and address load imbalance, probably the domain decomposition level.
- Investigate if the barrier operation is really necessary.

In both cases it might be beneficial to do further tracing (at small scales) which includes events for source code information to identify the location of the problems at the source code level.

3.8. References

Byckling, Mikko & Kataja, Juhani & Klemm, Michael & Zwinger, Thomas. (2017). OpenMP* SIMD Vectorization and Threading of the Elmer Finite Element Software. 123-137. 10.1007/978-3-319-65578-9_9.

4. Code SPECFEM3D

4.1. Brief description of code and use-case

SPECFEM3D simulates acoustic (fluid), elastic (solid) coupled acoustic-elastic poroelastic or seismic wave propagation in any type of conforming mesh of hexaedro (structured or not). It can, for instance, model seismic wave propagation in sedimentary basins or any other regional geological model following earthquakes. It can also be used for non-destructive testing or for ocean acoustics.

The Specfem3D code utilized for this performance analysis is version 4.0.0, specifically, the CUDA version. The code supports multiple GPUs through support for Message Passing Interface (MPI) parallelism. The MPI parallelism strategy is employed for domain decomposition, where each MPI domain is associated with a dedicated GPU. The performance analysis focused on the isotropic elastic core, a component widely used for seismic tomography (pilot demonstrator PD2) and seismo-acoustic simulation (PD1) as part of the ChEESE-2P project. Strong and weak scaling experiments were carried out to evaluate the effectiveness of the Specfem3D code on Leonardo booster (accelerated) partition with Nvidia A100 GPUs at CINECA. The tests were carried out on a range of mesh sizes, from 1441792 to 23068672 elements. Using degree 4 polynomial interpolation, each element includes 125 interpolation points. The mesh used is produced by the internal meshfem3d mesher, which provides a structured mesh and perfect load balance.

4.2. Scaling behavior

Specfem3d provides the execution time of the computation kernel in the main time loop. The rest is dedicated to reading and writing the mesh and the solution to the disk. Since our aim is to optimize the computational kernel used in the time loop, we used this metric to carry out our scalability tests.

Figure 4-1 shows the weak scaling done for a number of GPUs of 4, 8, 16, 32 and 64. For mesh sizes: 256x256x22, 512x256x22, 512x512x22, 1024x512x22 and 1024x1024x22 elements respectively.

Figure 4-2 shows the strong scaling done for a mesh of 1024x1024x22 elements with 8, 64, 128, 256 and 512 GPUs respectively. It shows the increase in execution speed as a function of the number of GPUs used.

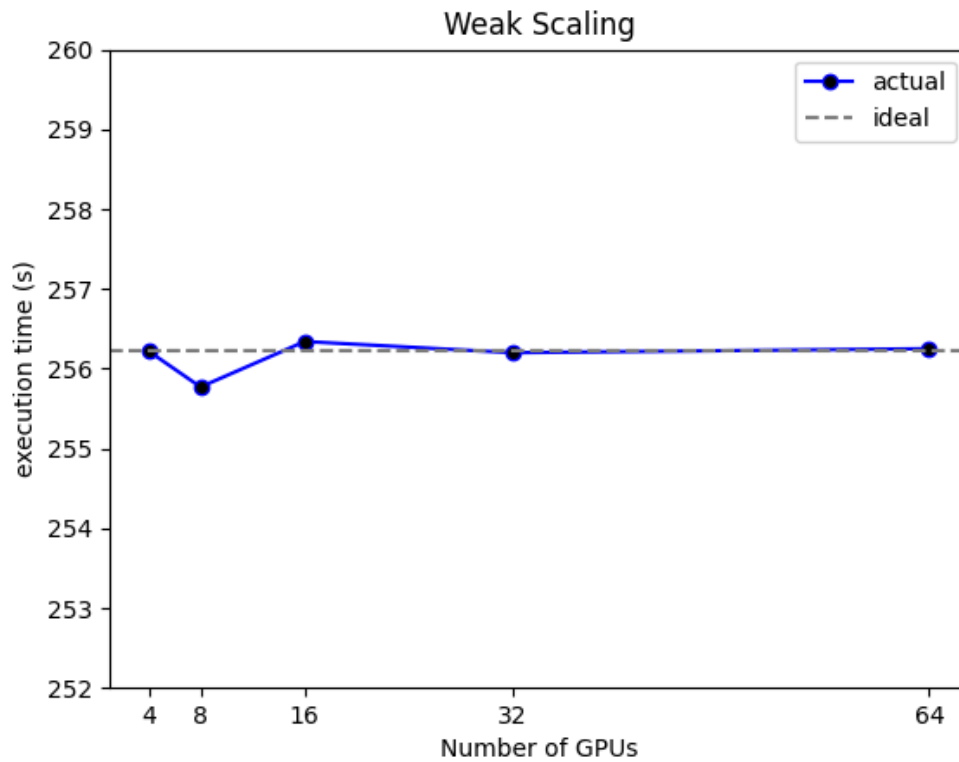


Figure 4-1 Weak scaling: execution time with respect to number of GPUs (one GPU per MPI rank).

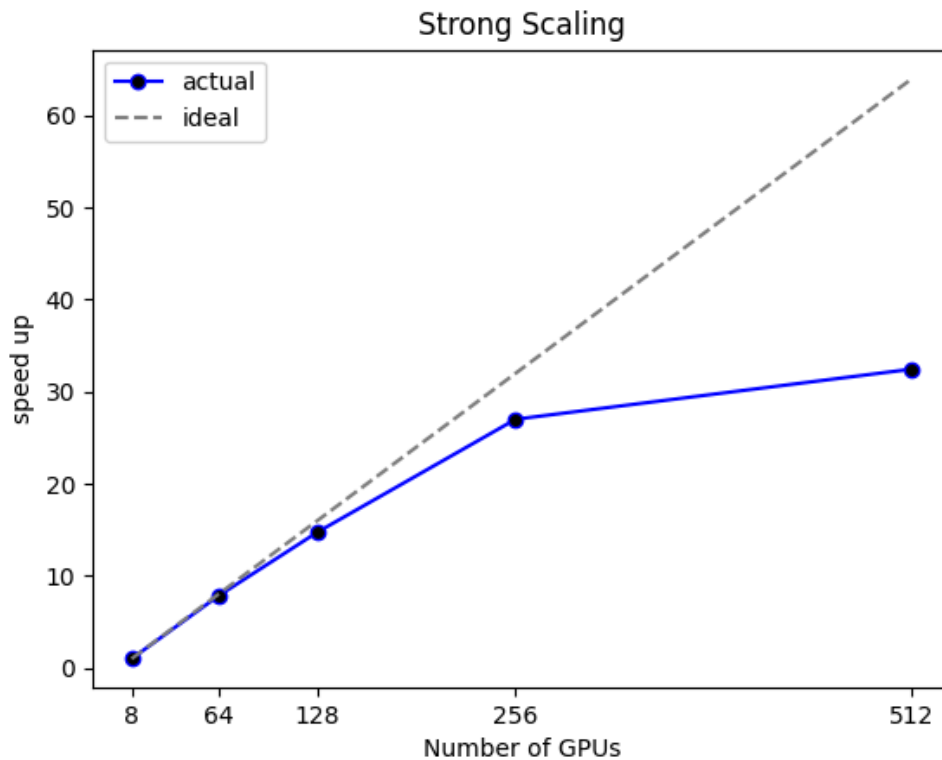


Figure 4-2 Strong scaling, speed up with respect to the number of GPUs used. The reference run is for 8 GPUs.

4.3. Performance data acquisition

Measurements were made using Scalasca/2.6.1 with Score-P/8.3 configured with Intel oneAPI 2023.0.0 compilers and 2021.7.1 MPI and CUDA/11.8. A total of 9 cases were run on 4, 8, 16, 32; 64, 128, 256 and 512 GPUs (1 to 64 nodes). The code region *xspeccfem3D/iterate_time* containing the 48000 iterations was chosen as focus-of-analysis.

4.4. POP performance efficiency metrics

Execution and scaling efficiencies in the following tables were calculated from the GPU perspective (neglecting the associated CPUs). CPU efficiency is very poor but not a significant factor overall. Orchestration efficiency includes MPI communication, kernel offload to accelerator devices, and file I/O (all done exclusively by CPUs).

Weak scaling is excellent to 16 compute nodes and expected to continue well beyond. Strong scaling remains acceptably good (above 0.8) only to 64 nodes before deteriorating significantly to 0.5 for 128 nodes.

Table 4-1 POP performance efficiency metrics for weak scaling

Weak scaling	MPI Processes				
	4	8	16	32	64
Global scaling efficiency	0.986	0.987	0.985	0.986	0.986
Computational time scaling	1.0	1.0	1.0	1.0	1.0
Parallel efficiency	0.986	0.987	0.985	0.986	0.986
Load balance efficiency	1.0	0.999	0.997	0.998	0.998
Orchestration efficiency	0.986	0.988	0.988	0.988	0.988

Table 4-2 POP performance efficiency metrics for strong scaling

Strong scaling	MPI Processes				
	8	64	128	256	512
Global scaling efficiency	0.995	0.973	0.919	0.843	0.507
Computational time scaling	1.0	0.987	0.972	0.950	0.937
Parallel efficiency	0.995	0.986	0.945	0.887	0.541
Load balance efficiency	1.0	0.998	0.996	0.994	0.989
Orchestration efficiency	0.995	0.988	0.948	0.892	0.547

4.5. Communication

Peer-to-peer communication of the acceleration field in the multi-GPU configuration takes place via the CPU memory. The communication process comprises a series of steps:

- 1. Device to Host Transfer:** Data is transferred asynchronously from the GPU device to the host CPU memory.
- 2. Non-blocking MPI Communication Between Hosts:** MPI communication takes place between different hosts, facilitating efficient exchange of information.
- 3. Host to Device Transfer:** Following MPI communication, the data is asynchronously transferred from the host back to the GPU device memory.

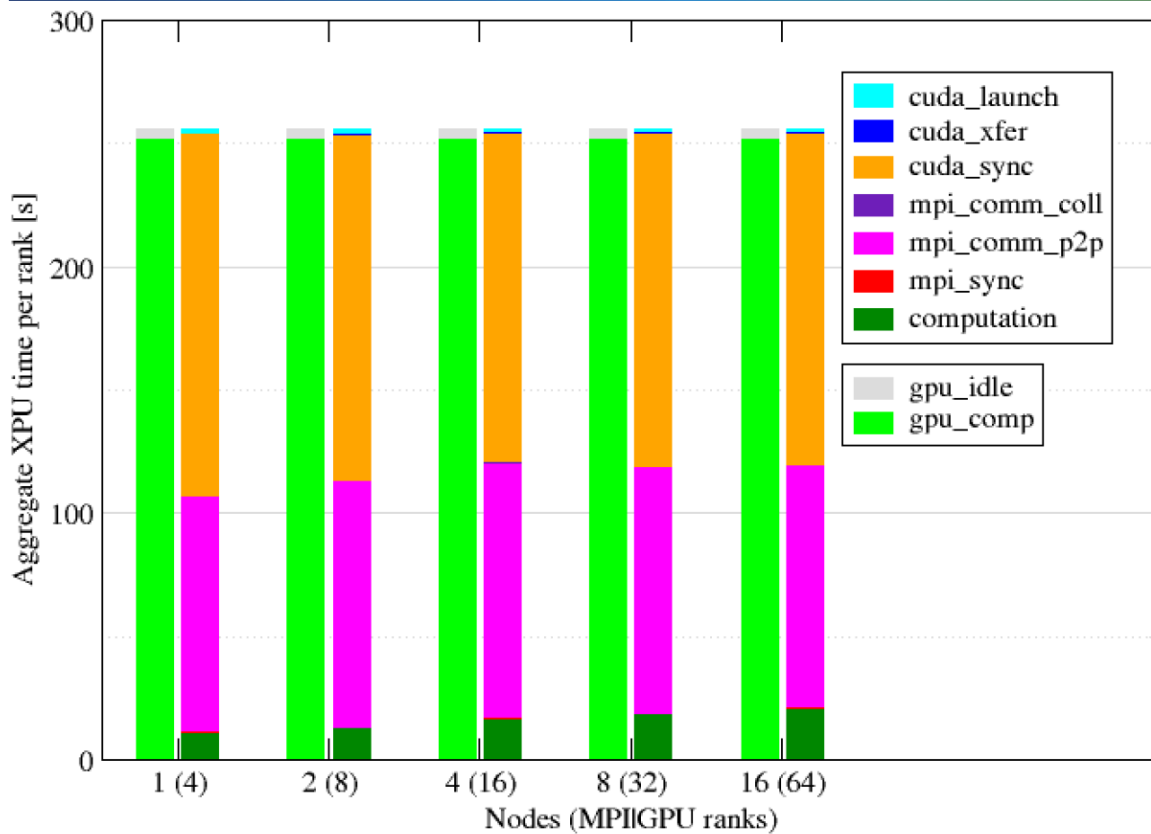


Figure 4-3 Fraction of total execution time of the FoA spent in cuda and MPI operations for weak scaling runs.

This entire communication process is orchestrated using asynchronous operations, leveraging MPI functionalities such as *mpi_isend* and *mpi_irecv* for communication between MPI domains. Data transfers between the host and the device memory are carried out asynchronously using *cudaMemcpyAsync*.

Figure 4-3 highlights the good weak scaling behavior up to 64 GPUs and potentially beyond. Computation is well balanced between GPUs, supported by excellent GPU utilization. Although the MPI point-to-point (P2P) communication time increases with scale, it is efficiently overlapped with the GPU compute kernels.

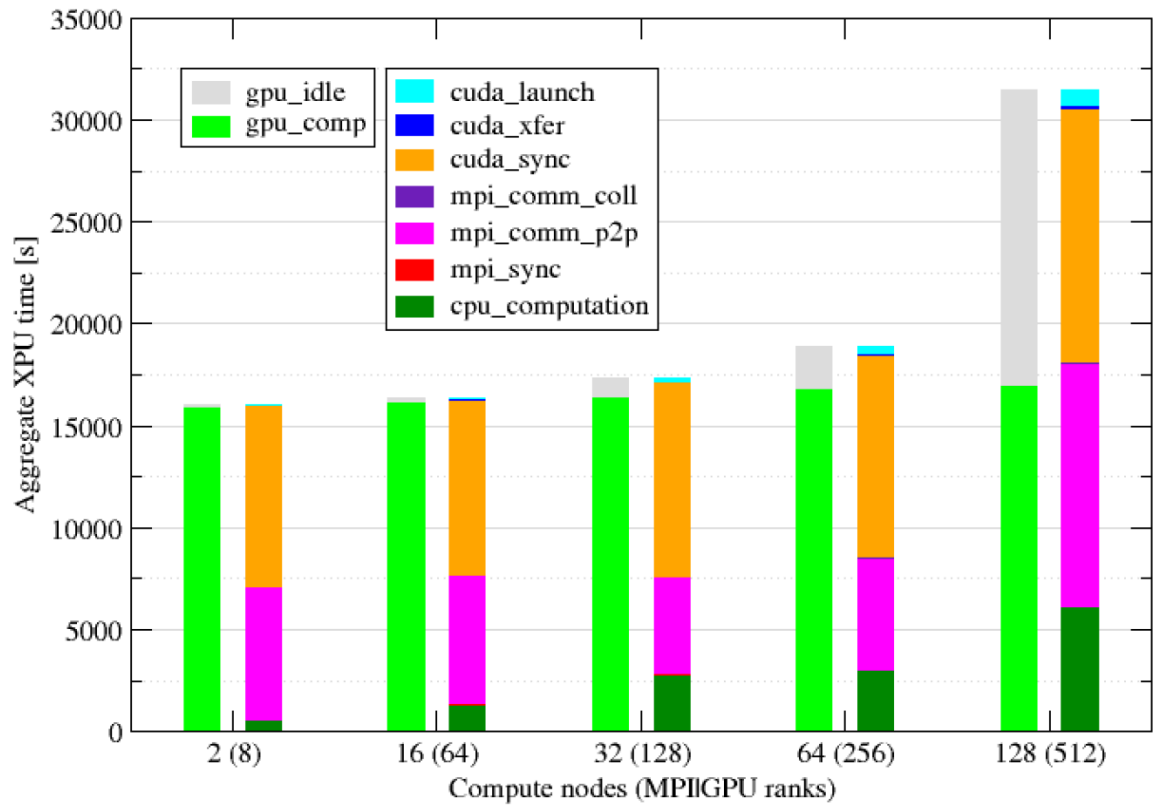


Figure 4-4 Fraction of total execution time of the FoA spent in cuda and MPI operations for strong scaling runs

In Figure 4-4, the analysis framework (FoA) demonstrates good strong scaling with efficient speedup to 256 GPUs. The computation keeps a good balance between GPUs. However, orchestration efficiency gradually decreases, accompanied by a noticeable increase in MPI point-to-point (P2P) communication time, which eventually ceases to be fully overlapped by GPU compute kernels.

4.6. Computational scalability

The system demonstrates excellent weak scaling, with minimal GPU idle time and efficient overlap of MPI communication with GPU kernel computation. Strong scaling is effective up to 256 GPUs, with manageable GPU idle time. However, as the number of GPUs increases to 512, GPU idle time becomes more pronounced, especially in the case of 512 GPUs, where MPI communication no longer achieves full overlap with GPU kernel computation.

4.7. Observations and recommendations

The solver chosen as focus of analysis demonstrates excellent weak scaling up to 16 nodes (64 GPUs) and is expected to continue well beyond as computation is very well balanced over GPUs and growing MPI communication is effectively overlapped with asynchronous GPU computation kernels. Although strong scaling is effective up to 64 nodes (256 GPUs), orchestration efficiency progressively diminishes as growing MPI communication no longer fully overlaps with computation kernels.

5. Code tandem

5.1. Brief description of code and use-case

Tandem is a scalable discontinuous Galerkin code on unstructured curvilinear grids for linear elasticity problems and sequences of earthquakes and aseismic slip (Uphoff et al., 2023). Tandem features a fully parallel implementation, including mesh loading, solution stage, output, and visualization. Tandem uses the Portable, Extensible Toolkit for Scientific Computation (PETSc), therefore benefiting from state-of-the-art solvers, pre-conditioners, and time integrators.

The use-case is a 3-D elastostatic model defining a kinematic scenario of instantaneous deformation inspired by the 2019 Ridgecrest earthquake sequence. In this use-case distinct “strike-slip” fault displacements are imposed across six geometrically complex and intersecting fault segments. The mesh (1017796 volume elements) and the chosen polynomial degree (4) are chosen to be representative of a production run of tandem. We use the multigrid pre-conditioner of PETSc to reduce the complexity of the problem.

We note that the use-case is not based on the full application (modeling sequences of earthquakes and aseismic slip), because such a use-case would require either enormous setup time and or storage to be representative. Instead, the use case is based on the static mini-app, and is equivalent to executing one time step of tandem, which encapsulates the most computationally-intensive part of tandem.

Improving the scalability and time-to-solution of tandem should participate in enhancing our fundamental understanding of earthquake faulting on different space-time scales. Within the project, tandem is envisioned to be integrated with other earthquake modeling and seismic wave propagation CHEESE-2P flagships in digital twin components, such as those developed in DT-GEO, and specific models and data will be prepared to be shared via EPOS, utilizing synergies with Geo-INQUIRE, and inform rapid response simulations.

5.2. Scaling behavior

We use the total run time as a performance metric and strong scaling (Figure 5-1) on LUMI-C. The (instrumented) use-case requires about 2800 GB of memory, and cannot be run on less than 12 nodes on the standard

partition, but can be run on the large mem partition on 1 node. Therefore we run the application on 1, 16, 32, and 64 nodes of LUMI-C, each with 128 ranks per node, and on 16 nodes with 32 ranks per node. Our scaling tests on Frontera suggest that tandem fully scales up to 1000 elements per rank. Because most of the tested configurations (except the one on one node, and the one with 16 nodes, 32 ranks per node) are below this limit the parallel efficiency is reduced. Figure 5-2 shows the same figure using the solve time only, that is without the initialization and finalization phase. The trends are relatively similar, but the efficiency using higher rank numbers has increased.

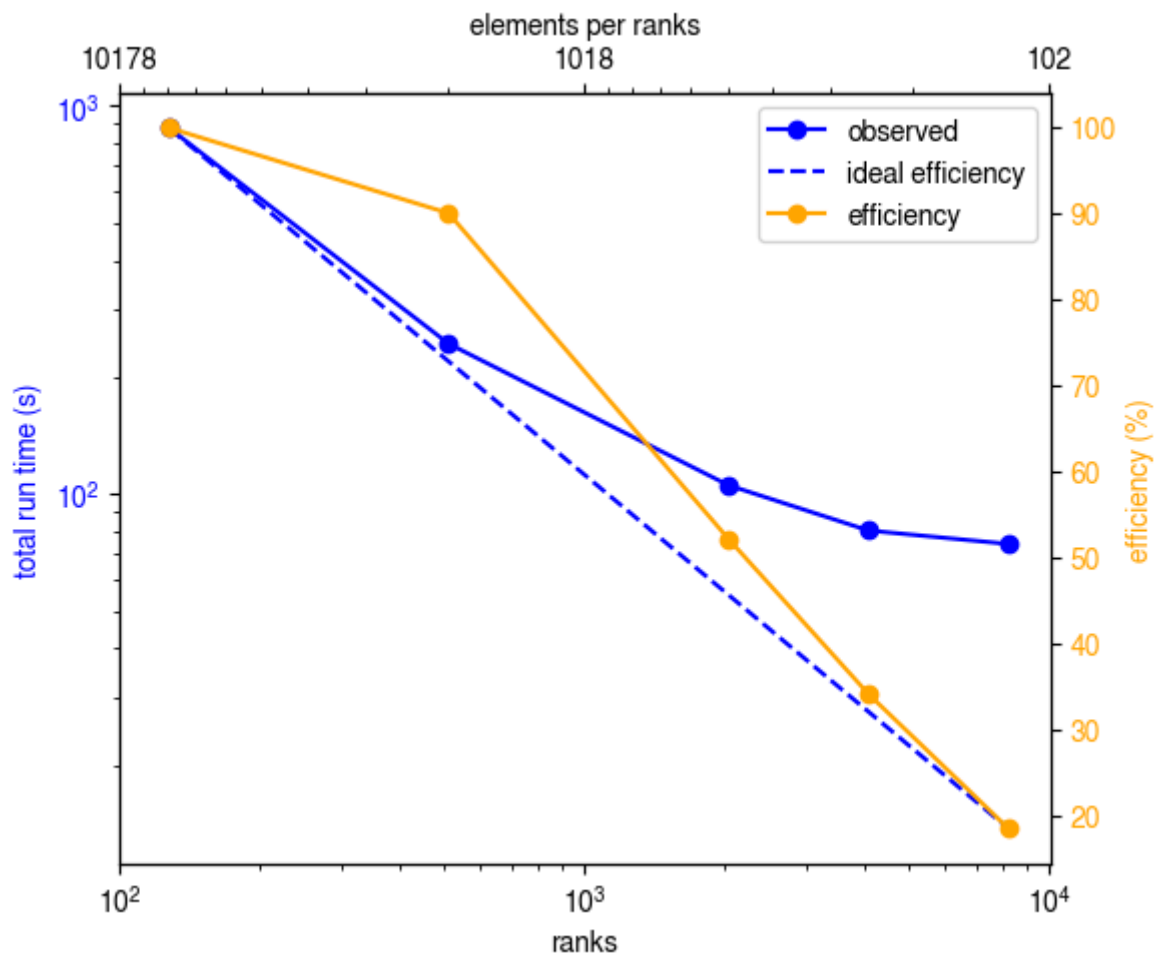


Figure 5-1 Strong scaling of the mini-app static of the CHEESE flagship code tandem.

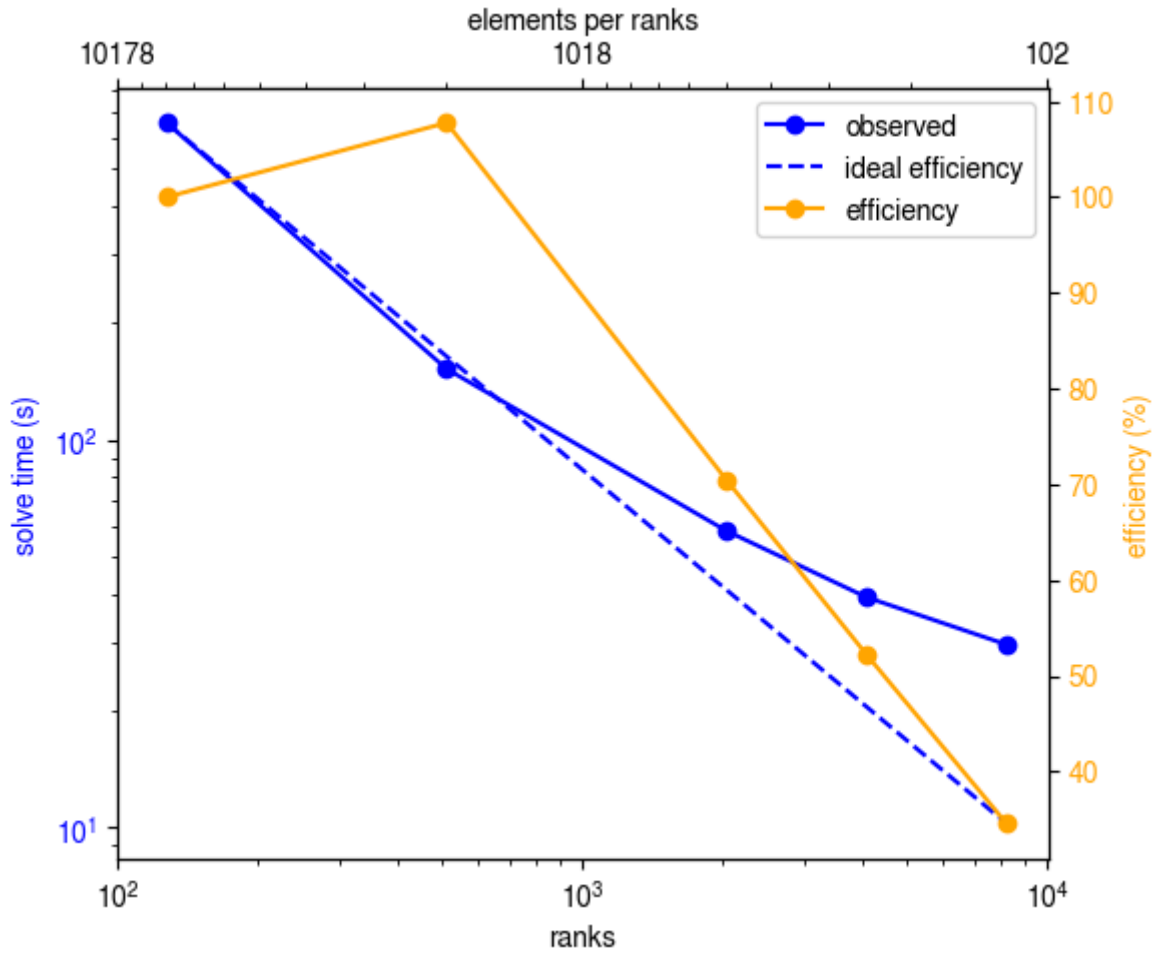


Figure 5-2 Strong scaling of the solve step in the mini-app static of the CHEESE flagship code tandem.

5.3. Performance data acquisition

Tandem was instrumented with Score-P/8.3 (configured with SCOREP_MPI_NO_MINI) for PrgEnv-gnu/8.4.0 (gcc/12.2.0) on LUMI-C, using SCOREP_WRAPPER_INSTRUMENTER_FLAGS="--nomemory -noopenmp -thread=none" (to avoid unnecessary instrumentation of threads and memory management). Although measurement dilation was small, execution measurement was done specifying a filter file that removed many inconsequential routines to have a more convenient profile for analysis. As focus-of-analysis the *static_problem* routine/callpath was chosen (incorporating assembly, warmup & solve phases) to avoid initialization (*MPI_Init_thread*) & finalization (*MPI_Finalize*).

5.4. POP performance efficiency metrics

The POP MPI efficiency model was used for Score-P profile summaries.

MPI ranks	128	512	2048	4096	8192
MPI configuration	n1p128	n16p32	n16p128	n32p128	n64p128
Solve iterations	18	18	19	20	20
Wall-clock time [s]	850.78	216.77	74.76	50.73	39.42
Computation time [s]	95671.33	74046.30	76451.98	78419.05	78618.84
Global scaling efficiency	0.879	0.862	0.624	0.461	0.296
- Computation time scaling	1.000	1.292	1.251	1.221	1.217
- Parallel efficiency	0.879	0.667	0.499	0.377	0.243
- - Load balance efficiency	0.904	0.882	0.735	0.783	0.638
- - Communication efficiency	0.971	0.756	0.679	0.482	0.382

The single-node execution with 128 MPI processes has excellent communication efficiency and good load balance, resulting in good parallel efficiency of 0.88. This execution configuration is used as the baseline for strong scaling up to 64 nodes, where the global scaling efficiency progressively deteriorates and is no longer acceptable (below 0.8) beyond 512 MPI processes.

5.5. Communication

The excellent communication efficiency of the single-node configuration deteriorates rapidly as more compute nodes are employed and shared-memory data transfers are replaced with SlingShot interconnect communication. Tandem uses the PETSc library which employs copious (primarily non-blocking) point-to-point communication as well as a wide variety of collective MPI communication (involving all processes equally). *Waitall* consumes the most pt2pt time, followed by *Iprobe* and *Test(all)*. While *Allgather(v)* is most time consuming during warmup, *Allreduce* is dominant during the solve phase. Further investigation would be required to quantify variability and impact of interference from concurrent jobs.

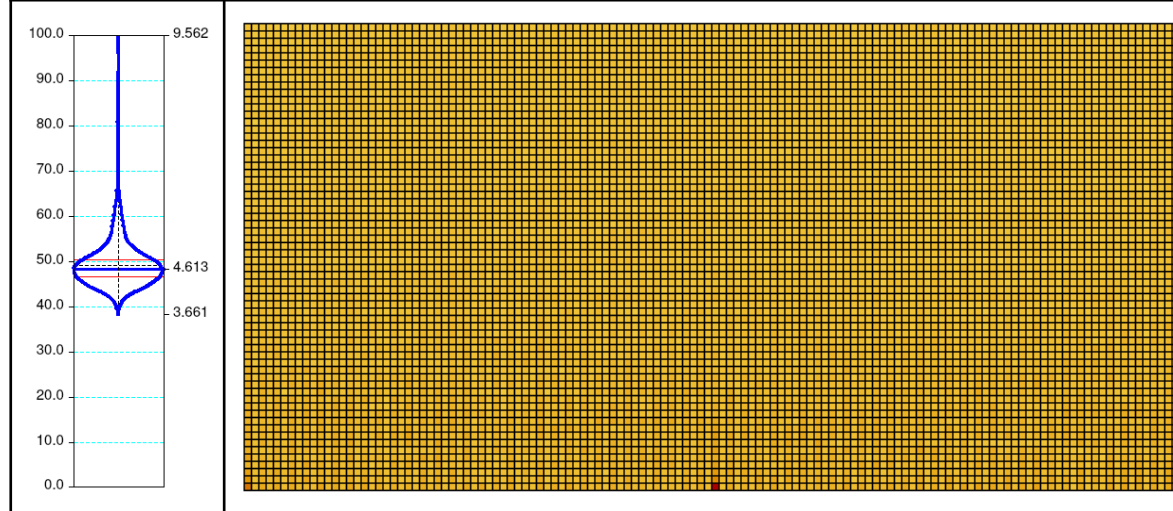
5.6. Computational scalability

Executions with 16 and more nodes show superlinear scaling of the aggregate computation time. This suggests that the very large working set fits better into cache when partitioned than for the single node configuration. Additional measurements including hardware counters for instructions and cycles could be taken to calculate IPC to confirm this.

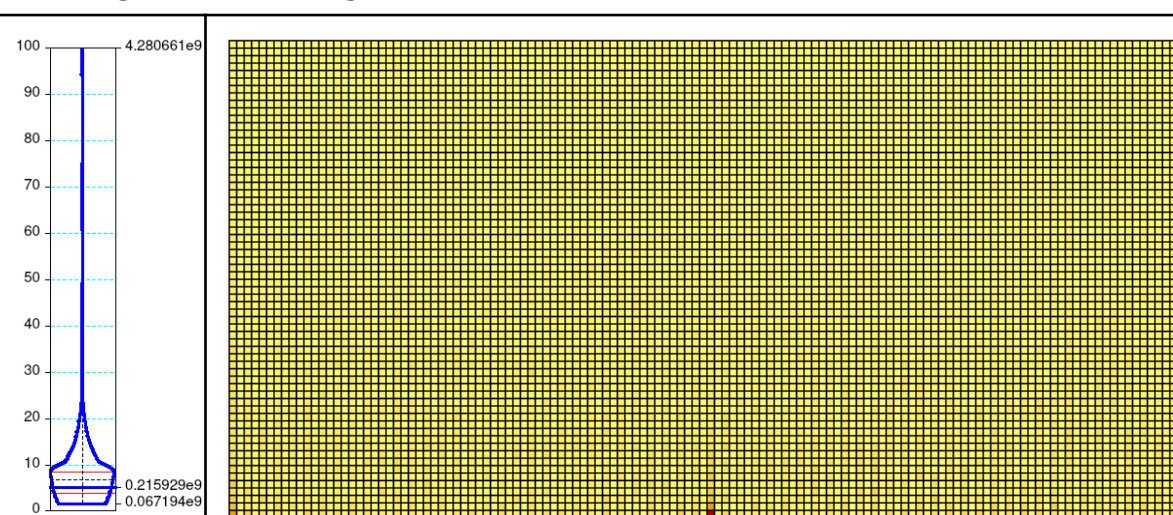
5.7. Observations and recommendations

Computation load balance varies erratically, starting relatively good for up to 512 processes and becoming noticeably poorer beyond that. With 8192 processes, one process takes much longer than the others for its computation, however, all processes on the last node (with the highest MPI ranks) generally are found to be slower.

Computation time distribution for *static_problem* (solve) with 8192 MPI processes, showing rank 8128 requiring notably longer than the others.



Distribution of bytes transferred with MPI point-to-point communication in *static_problem* with 8192 processes, showing bottleneck rank 8128 sending and receiving much more data than the others.



Execution configurations with larger numbers of MPI processes require additional solver iterations to reach the required solution accuracy, and this naturally negatively impacts their efficiency.

Score-P execution traces could be collected and analyzed with Scalasca for additional insight into communication efficiency, however, these are prohibitively large due to the copious use of *Iprobe* & *Test* events (currently recorded for multiple failures as well as valuable successes).

While Tandem's assembly and warmup phases are already distinguishable in Score-P measurements, explicit instrumentation annotation of the solve phase will be required to reliably distinguish it from the rest of *static_problem* (which includes optional file writing).

5.8. References

Uphoff, C., May, D. A., & Gabriel, A. A. (2023). A discontinuous Galerkin method for sequences of earthquakes and aseismic slip on multiple faults using unstructured curvilinear grids. *Geophysical Journal International*, 233(1), 586-626.

6. Code SeisSol

6.1. Brief description of code and use-case

SeisSol¹¹ (developed together by the groups of TUM and LMU) is a comprehensive package for earthquake simulation and related seismic wave propagation problems. It solves the seismic wave equations (in elastic, elastic-acoustic, viscoelastic or poroelastic material) using a high-order discontinuous Galerkin discretization on unstructured static-adaptive tetrahedral grids. It provides state-of-the-art rupture laws for dynamic rupture earthquake simulations. In ChEESE-1P a novel elastic-acoustic model (with gravity boundary condition) for fully-coupled earthquake—tsunami simulation has been developed.

SeisSol is used in Pilot Demonstrators PD1 (Extreme-scale modeling of seismic hazards) and PD4 (Complex multi-source tsunami modeling). We target both capability runs using extreme-scale models for earthquake and earthquake-tsunami events, as well as capacity problems, where SeisSol is used in workflows for uncertainty quantification and probabilistic seismic hazard assessment.

For the performance audit, we consider the TPV5 benchmark, a community benchmark to verify correctness of dynamic rupture simulation codes. TPV5 models spontaneous rupture on a vertical strike-slip fault with heterogeneous initial stress conditions. We use meshes with up to 17 million grid cells, using discretization order 6, which already leads to problems with more than 8 billion degrees of freedom.

6.2. Scaling behavior

Since 2014, SeisSol has been regularly used for landmark simulations and extreme scaling experiments on various petascale machines. As an example, Figure 6-1 shows a strong-scaling study for the ChEESE-1P pilot demonstrator PD4 on up to 6336 nodes (full machine) of the SuperMUC-NG supercomputer. The up-to-now largest production run with SeisSol was executed in October 2023 on the Frontera supercomputer at the Texas Advanced Computing Centre. LMU simulated the 2023 Kahramanmaraş earthquake (with viscoelastic attenuation and dynamic

¹¹ www.seissol.org



rupture on rough faults) on meshes with up to 691 million elements (approx. 10^{12} degrees of freedom) achieving 9.6 PFlop/s on 8192 compute nodes (full-machine run over 9 hours; 2x28 cores per node, Intel 8280 “Cascade Lake” CPUs).

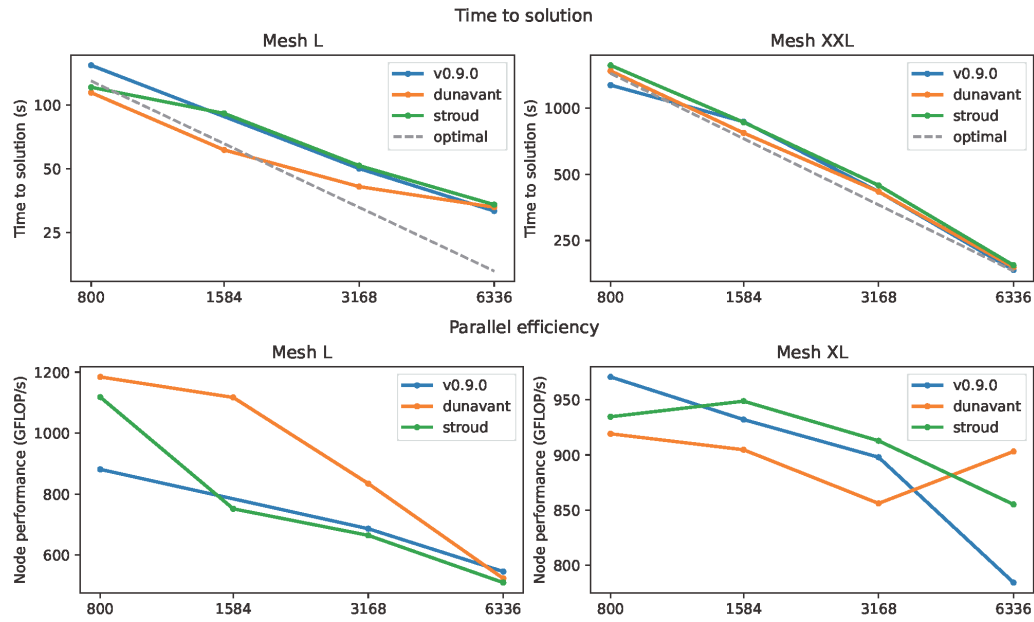


Figure 6-1 Strong scaling of SeisSol on SuperMUC-NG (fully-coupled elastic-acoustic earthquake-tsunami simulation of the 2018 Palu event) for meshes with 89 million (L) to 518 million (XXL) elements, comparing SeisSol’s existing (0.9.0) dynamic rupture implementation with a new implementation that offers two different integration rules (dunavant vs. stroud). The runs on 6,336 nodes were slightly impeded by performance fluctuation of SuperMUC-NG.

SeisSol uses an MPI+X parallelization approach (‘X’ = OpenMP on CPU architectures). In ChESEE-1P SeisSol has been ported and optimized for GPU architectures (using primarily SYCL and CUDA as GPU programming model). SeisSol relies on CUDA graphs to mitigate the overhead for launching many kernels to GPUs. It uses a separate communication thread to ensure MPI progress for overlapping computation and communication.

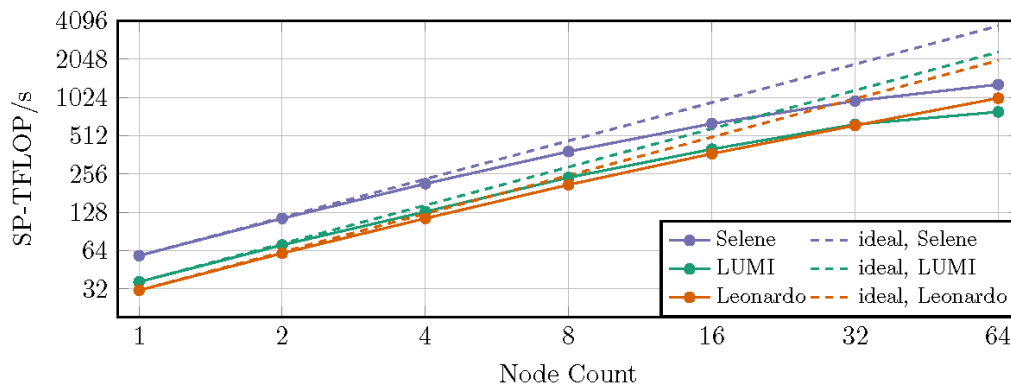


Figure 6-2 Strong scaling of SeisSol on three GPU supercomputers (Selene, Leonardo and LUMI) for the TPV5 benchmark with 17 million elements (see Fig. 3 for distribution of

LTS clusters). Selene and Leonardo use NVIDIA A100 GPUs, LUMI is based on AMD MI250x.

As part of ChEESE-2P, we have evaluated the performance and verified the usability of the GPU port for production runs on LUMI and Leonardo as part of respective EuroHPC Development Access project) – see Figure 6-2 for respective scaling tests and the overview in Table 6-1 for production scenarios.

Table 6-1 Production runs for several scenarios on the LUMI and Leonardo supercomputer ("Turkey": Kahramanmaraş earthquake 2023, performed on LUMI; "Ridgecrest": Ridgecrest earthquake sequence, 2019; "Palu": 2018 Sulawesi earthquake and tsunami event; Ridgecrest and Palu performed on Leonardo).

Scenario	#elements	#nodes	#ranks/GPUs	Precision	PFlop/s
Turkey	175 mio	128	1024	single	1.52
Ridgecrest	27 mio	32	128	double	0.22
Palu (M)	89 mio	64	256	double	0.42
Palu (L)	518 mio	256	1024	double	1.78

Recent work has identified SeisSol's **cluster-oriented local time stepping** algorithm as a particular challenge for GPU performance. SeisSol groups elements with similar time step restrictions into local time stepping (LTS) clusters which are then propagated with a uniform time step. Time step sizes of separate clusters are multiples of each other ("multi-rate" scheme, typically with factor 2 or 3). Figure 6-3 shows a histogram of the number of elements in the LTS clusters for our TPV5 benchmark scenario.

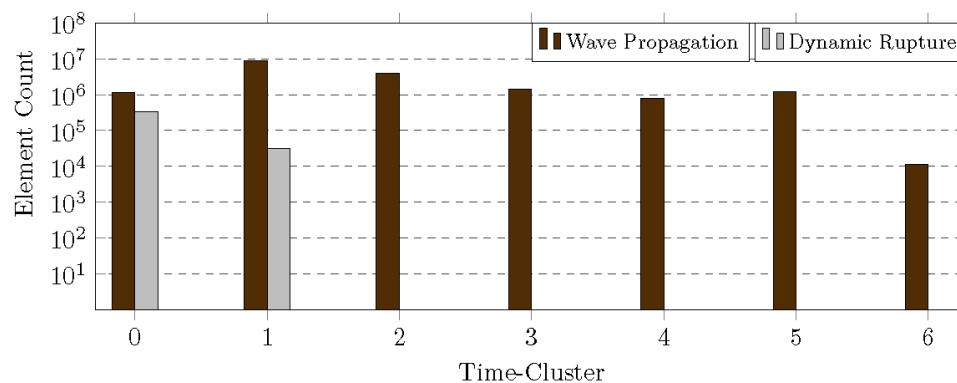


Figure 6-3 Number of elements in each LTS cluster for the TPV5 benchmark with 17 million elements (grey bars indicate the distribution of elements with dynamic rupture faces). Note the logarithmic scale: the largest LTS cluster 1 has 10x more elements than cluster 0, which has the smallest time step and is thus updated most frequently.

Small LTS clusters cannot exploit the massive parallelism provide by GPUs; respective GPU kernels may suffer from substantial performance loss. In strong-scaling setups, this also leads to speedup limitations, as the overall number of elements per node (and thus also per LTS cluster) decreases.

6.3. Performance data acquisition

Initially, we planned to use Extrae to record traces from SeisSol. However, up to now we did not manage to obtain traces. Code developers and performance analysts have spent weeks debugging and solving a long series of issues across the whole development and performance analysis tool chain, but have not yet found a satisfactory solution. These problems included:

- tracing support for CUDA+MPI+Pthread
- lack of support for CUDA Graphs (creating and instantiating of graphs)
- problem with large number of CUDA streams: streams allocate a large (10^6) number of CUDA events on stream creation; SeisSol seems to use much more streams than typical codes
- segfault errors in the CUDA stream search (could be a problem in Extrae's stream registration)

In the end, we decided to resort to manually including time measurements in the source code, rather than using Extrae or Score-P. Building on a previously existing code timing method, the code owners have surrounded each phase of MPI and compute kernel, respectively, with time measurement operations. At the end this information is aggregated and saved. The data is sufficient to calculate the most basic set of POP performance efficiency metrics.

6.4. POP performance efficiency metrics

From the internal code timings, we have calculated POP metrics using the simple MPI-only model. With this data, we could not decompose communication efficiency further into serialization and transfer efficiency. Table 6-2 shows the performance metrics for a strong scaling use case with 16, 32, and 64 nodes, respectively, and 4 MPI processes/GPUs per node.

Table 6-2 POP performance efficiency metrics

	Nodes		
	16	32	64
Global efficiency	96%	82%	62%
Parallel efficiency	96%	96%	93%
Load balance efficiency	97%	97%	93%
Communication efficiency	99%	100%	100%
Computational scalability	100%	85%	67%

From the MPI perspective, i.e., communication efficiency, the code is near perfect. MPI communication overlaps very well with computation and causes virtually no delays. Also, load balance across MPI processes is very high, though it drops slightly at 64 nodes.

The only area for concern is computational scalability, which measures how time spent in actual computation (as opposed to time spent in MPI) changes as compute resources are added with respect to the reference at 16 nodes. There is a clear loss of efficiency reaching 67% at 64 nodes. This is consistent with observations by the developers that GPU performance depends on the size of LTS clusters: for small LTS clusters launch overhead and lack of parallelism limit the GPU performance. Full tracing, once available, could provide valuable further insight on this issue.

7. Code Tsunami-HySEA

7.1. Brief description of code and use-case

Tsunami-HySEA is a numerical model to simulate tsunamis generated by earthquakes. It is implemented in MPI and CUDA. The computations are performed in a single GPU or a multi-GPU cluster. In Tsunami-HySEA the three phases of an earthquake generated tsunami (deformation of the seafloor, propagation of the tsunami and inundation of the coastal areas) are performed in a single code. Many features are supported, such as nested grids, multiple seafloor deformations at any time instant, time series, compressed NetCDF output files, asynchronous file writing, multiple metrics, dynamic deformation and variable friction.

Tsunami-HySEA is one of the codes feeding “WP5 - Farming Pilot Demonstrators” in ChEESE-2P (pilot demonstrators PD3 and PD4). Besides the standard one-layer hydrostatic version of Tsunami-HySEA code, two new operational versions are being developed for ChEESE-2P. The first one, a one-layer non-hydrostatic/dispersive version for earthquake generated tsunamis and, the second one, a version to be coupled with atmospheric forcing for meteotsunamis, Meteo-HySEA. All these three codes share common features at the formulation and algorithmic level. Therefore, improvements in efficiency in Tsunami-HySEA code are immediately inherited by these two new versions. Besides this, a HySEA mini-app will also be released. In addition to the tasks associated with “Code Audit” and the present deliverable, Tsunami-HySEA code is involved in the tasks dealing with “Optimization for Exascale” and “Continuous Performance Efficiency Monitoring”, associated to WP2; HySEA mini-app, OpenSequana and Risk-V prototypes in WP3; the deployment in EuroHPC system of Tsunami-HySEA in a context of CI/CD in WP4; and the already described implementation of novel features in HySEA in WP5. Finally, Tsunami-HySEA plays an important role in the “Set-up of Simulation Cases”, in two “Exascale Simulation Cases” (SC3.1 and SC4.1), and in a “Scientific Grand Challenge” (SC4.1).

The selected use case for the audit has been the simulation of the tsunami caused by the earthquake that occurred on March 11, 2011, in Tohoku (Japan), using a mesh of 336.84 millions of volumes covering all the Pacific Ocean with a spatial resolution of 30 arc-sec. A very big mesh

has been used in order to obtain good performance using a high number of GPUs.

7.2. Scaling behavior

The audit has been carried out on the Leonardo supercomputer (CINECA) using up to 64 NVIDIA A100. The following Table 7-1 and Figure 7-1 show the strong scaling obtained, based on the runtime measured inside the application, which considers only the simulation time.

Table 7-1 Strong scaling of main simulation loop on Leonardo.

Number of GPUs	Speedup over 1 GPU
1	1
4	3.98
8	7.91
16	15.69
24	23.23
32	30.41
40	37.33
48	44.25
56	50.30
64	57.40

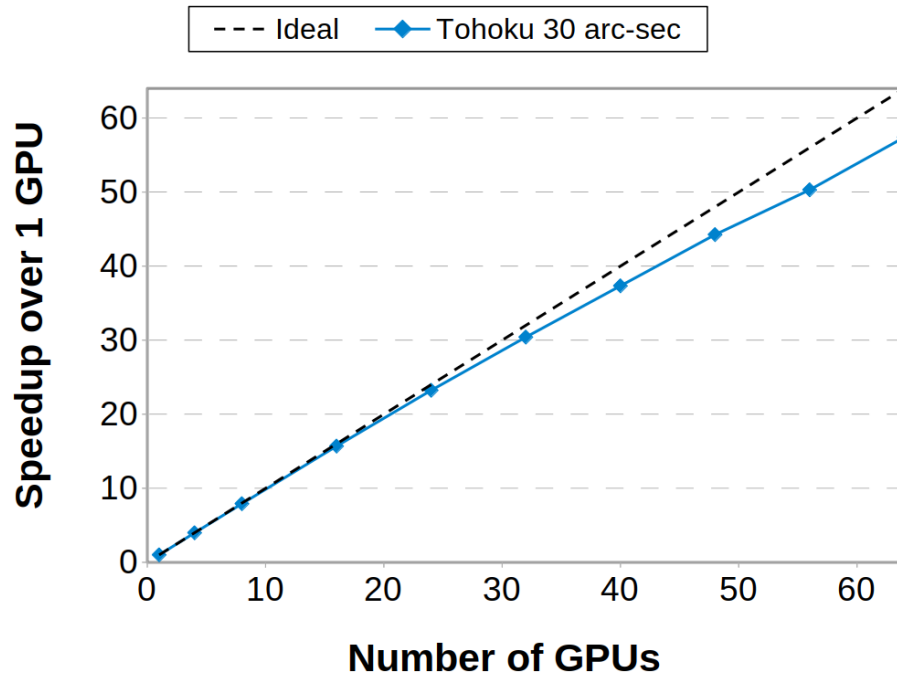


Figure 7-1 Strong scaling of main simulation loop on Leonardo.

7.3. Performance data acquisition

Extrae has been the tool used to perform the full audit. The traces have been obtained by executing Tsunami-HySEA on Leonardo with the libcudaampitracecf.so library preloaded in the mpirun command. This generates intermediate Extrae traces. Then, mpi2prv has been executed to obtain the final traces to be used in Paraver (.prv, .pcf and .row files) from the intermediate ones. Traces have been obtained using 1, 4, 8, 16, 24, 32, 40, 48, 56 and 64 GPUs.

It has been necessary to compile Tsunami-HySEA with special flags ("-gdwarf4") to avoid an error regarding the dwarf version at the end of the mpi2prv execution.

By observing the traces, there exists more variability between processes (GPUs) than over time. This means that little data is needed to perform the analysis. Specifically, 30 iterations have been selected to carry out the study.

7.4. POP performance efficiency metrics

In the following Table 7-2 we summarize the results obtained for the efficiency metrics using 1, 16, 32, 48 and 64 GPUs

Table 7-2 POP performance efficiency metrics

	1	16	32	48	64
Global PE	47.43	46.27	44.48	43.30	42.04
Global LB	50.00	49.91	49.83	49.64	49.40
GPUs GE	94.85	91.03	85.99	82.77	79.00
GPUs PE	94.85	92.39	88.67	86.18	83.52
GPUs LB	100.00	99.65	99.33	98.80	98.15
GPUs CE	94.85	92.71	89.27	87.23	85.09
GPUs CS	100.00	98.53	96.98	96.05	94.59

PE: Parallel Efficiency

LB: Load Balance

GE: Global Efficiency

CE: Communication Efficiency

CS: Computation Scalability

The numbers obtained for Global PE and Global LB (around 40-50%), metrics that consider all the resources, are explained by the low contribution of the CPUs. Most of the work is performed on the GPUs, which have a very high usage. The GPUs are executing kernels the 94% of the time. The weight of MPI is small (less than 1% of the time).

There is a small degradation of the parallel efficiency in CPU and GPU as the number of GPUs increases.

The GPUs GE reports a degradation with the scale that is related mainly with the communication efficiency but also with the computation scalability. The load balance between GPUs is fine for all the scales.

No counters are available at GPUs to compute IPC and instructions scaling factors.

7.5. Communication

The communication efficiency corresponds to waiting time, either idle or in the CUDA API that is not due to global unbalance. It corresponds to

temporal unbalances compensated over time, or an overhead required to implement the runtime.

After one CUDA kernel ends, the device calls *cudaMemcpy*. There is a gap of around 230us where the CPU is still on the CUDA call. This gap has the same duration using 4 and 64 GPUs, and corresponds to runtime overhead. When scaling up, this gap becomes proportionally more relevant, reducing the communication efficiency. After this phase, the *MPI_Allreduce* absorbs the unbalance.

With 1 and 4 GPUs we identify a gap between the kernel launch and its execution, not related with MPI. There is almost no gap with 64 GPUs. We can observe that this gap reduces proportionally with the number of GPUs. It seems related to the volume of data. Since the duration of the gap reduces with more GPUs, the scaling time efficiency is improved.

The *MPI_Allreduce* increases its percentage time with the number of GPUs.

7.6. Computational scalability

Most of the kernels have a very good scaling behavior. 10 kernels have a scaling efficiency between 0.94 and 0.98. The reduction kernel has a bigger degradation with a scaling efficiency of 0.87 using 64 GPUs. There are 4 kernels with very bad scaling (which correspond to the processing of communication edges), with a relative weight smaller than 0.5%.

In the chart that depicts the relative weights of the kernels, the 4 inefficiently parallelized kernels start to become visible with 64 GPUs, but they are still far from being relevant.

A load balance bigger than 0.8 can be considered good. There are 3 kernels with a load balance lower than 0.8, being 0.7 the lowest load balance for one kernel, representing 9% of the runtime. The other 2 kernels with low value represent less than 5% each.

7.7. Observations and recommendations

The key points that have been observed regarding HySEA can be summarized in the following items:

- The scaling of HySEA is good, and it has improved since the previous study.
- The efficiencies considering all the resources are low because the CPUs are basically dedicated to feed the GPUs. Considering only the GPUs, the efficiencies, as well as their scaling, are good.
- At hybrid level, the interaction between CUDA and MPI is optimized and the only delay generated by MPI is in the allreduce.
- The analysis identified some overheads for the CUDA runtime with a different impact in the scaling.
- Most of the CUDA kernels have good scaling and balance. The study detected 2 kernels that may be considered for improvement.
- Using strong scaling limits the range of scales to analyze. It may be interesting to analyze a weak scaling study.

8. Code LaMEM

8.1. Brief description of code and use-case

LaMEM¹² is a parallel 3D numerical code that can be used to model various thermo-mechanical geodynamical processes such as mantle-lithosphere interaction, or the dynamics of magmatic systems, for rocks that have visco-elasto-plastic or (compressible) poroelastic rheologies. The code uses a marker-in-cell approach with a staggered finite difference discretization and is built on top of PETSc. A range of (Galerkin) multigrid and iterative solvers are available, for both linear and non-linear rheologies, using Picard and quasi-Newton solvers. LaMEM has been tested on massively parallel MPI-clusters and has checkpointing/restarting capabilities.

The use-case in ChEESE-2P¹³ is a 3-D model of the Etna volcano, which is a part of simulation case SC8.1. The model is built from a high-resolution topographical survey around the Etna volcano and literature-based lithosphere structure. We include the formation of faults in the simulation and have the aim to understand the conditions that may lead to an unstable eastern flank of the volcano (as currently observed). The goal in ChEESE-2P is to increase the performance portability of the code for different accelerators to guarantee an efficient use of available EuroHPC systems (such as LUMI). It is meant to accomplish this by porting and tuning efforts devoted to the most effective use of hardware with emphasis on performance portability. Currently, LaMEM does not utilize GPUs, but PETSc has support for it in recent versions. The goal of the ChEESE-2P project is to start porting LaMEM to make use of accelerators, and to improve scalability of the code on CPU machines.

This will be achieved by meeting the objectives of pilot demonstrator PD8: improving the workflow of geodynamic codes to the extent that they can be rapidly used to simulate any part of the Earth; preparing codes for next-generation GPU accelerators by improving multigrid strategies, implementing matrix-free solvers and novel w-BFBt preconditioners.

¹² <https://github.com/UniMainzGeo/LaMEM>

¹³ https://github.com/UniMainzGeo/LaMEM//input_models/ScalingTests/Volcano.dat

8.2. Scaling behavior

LaMEM can employ PETSc built-in logging instruments. It allows monitoring performance of every PETSc function that was called. We used information provided by PETSc log view options such as time for each multigrid level, time to solve coarse grids, average flops and amount of MPI messages performed during simulation.

LaMEM simulations run in multiple stages. The most important and computationally-heavy stage is solving the nonlinear Stokes problem. To solve it we use a multigrid approach with a parallel direct solver employed on the coarse grid. This approach allows for substantial speedup by reducing the amount of iterations. This method scales provided that the time for solving the coarse multigrid level is relatively small compared to the smoothing steps on the finer levels.

8.2.1. Model setup

In the prepared setup, the LaMEM simulation runs through stages of temperature initialization and diffusion, an initial stage of stress distribution under lithostatic pressure, and a time step that solves the Stokes problem using the multigrid method. The part of the code that solves the non-linear Stokes problem is the most computationally and memory heavy part of the run (Figure 8-1).

With this in mind, we provide scaling behavior that shows time spent on running 1 timestep of the Stokes non-linear solver.

We performed tests on different machines. We used LUMI-C because the goal of ChEESE-2P is to provide codes that can run efficiently on pre-exascale EuroHPC systems. We also used the HPC cluster MOGON-2 in Mainz as comparison. On both systems we used PETSc version 3.18.5 to perform tests.

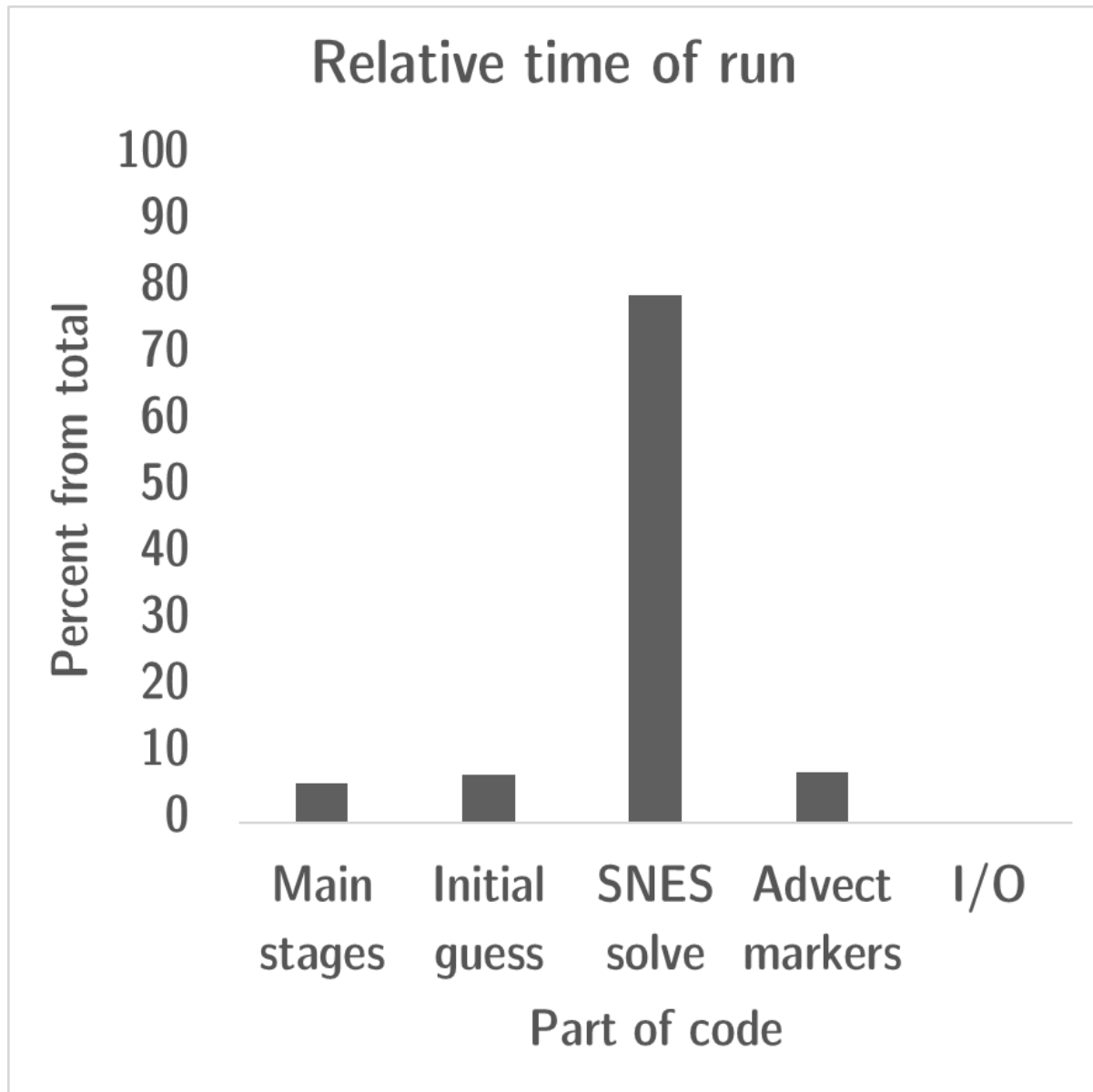


Figure 8-1 Time spend in different stages of a typical LaMEM simulation.

8.2.2. Weak scalability

For weak scalability we chose an Etna volcano 3-D setup where we gradually increased resolution and thread count while keeping the same problem size per thread with $32^3=32768$ dof/thread. As we use the multigrid method, we ran the simulation with different multigrid levels (3,4,5,6) and selected the best results for each resolution for the report. For a resolution less than 128^3 and less than 64 threads there were used multigrid levels 3 and 4. For a bigger model, multigrid levels 5 and 6 were chosen. Accordingly, our global coarse grids and local coarse grids also varied, changing from configuration [8 8 8] to [16 16 32] and [1 1 1] to [8 8 8] respectively.

Considering weak scaling from 1 thread we can observe a substantial drop in efficiency. However, with thread counts higher than 64 threads we see a stabilization in efficiency which show an almost flat trend. Note that a LUMI-node has 128 cores, and intra-node communication therefore only starts with >128 cores (see Figure 8-2).

Figure 8-3 demonstrates in more detail that weak scaling trend with thread counts higher than 64 threads is close to 1. Note that on LUMI-C we encountered an issue with running on >1024 threads, and are working with the team to resolve that.

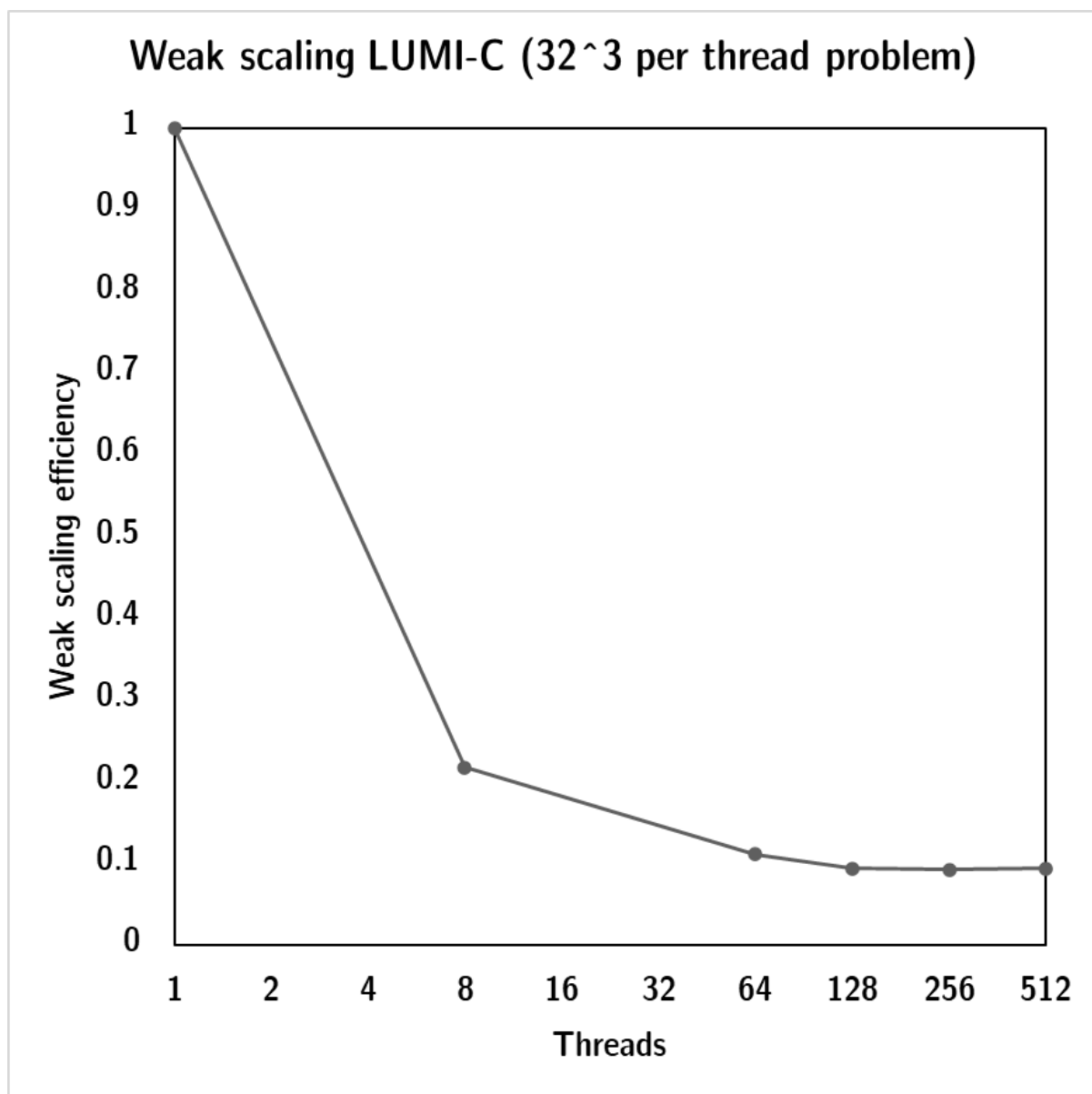


Figure 8-2 Weak scalability of a single timestep simulation on LUMI with 32^3 resolution per thread. Note that intra-node communication only starts at >128 threads.

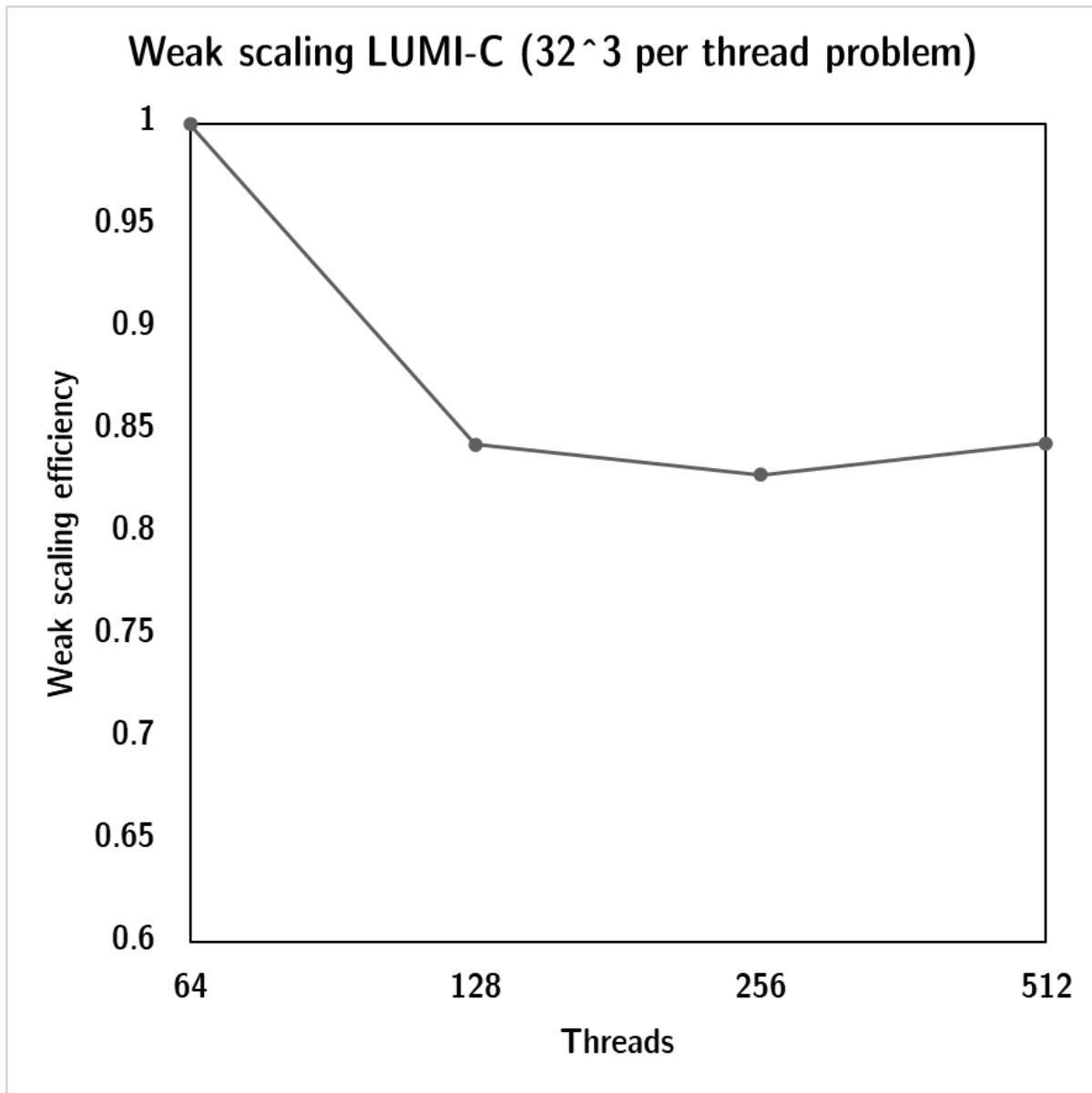


Figure 8-3 Weak scalability for simulations that saturate the nodes. Intra-node communication starts from >128 threads.

For comparison we performed similar tests on MOGON II, where we observed the same behavior. The efficiency with less than 512 threads is close to 1. However, when we ran weak scalability beyond 512 threads there is a significant declining efficiency. One of the possible causes for such a decline will be given in the discussion section.

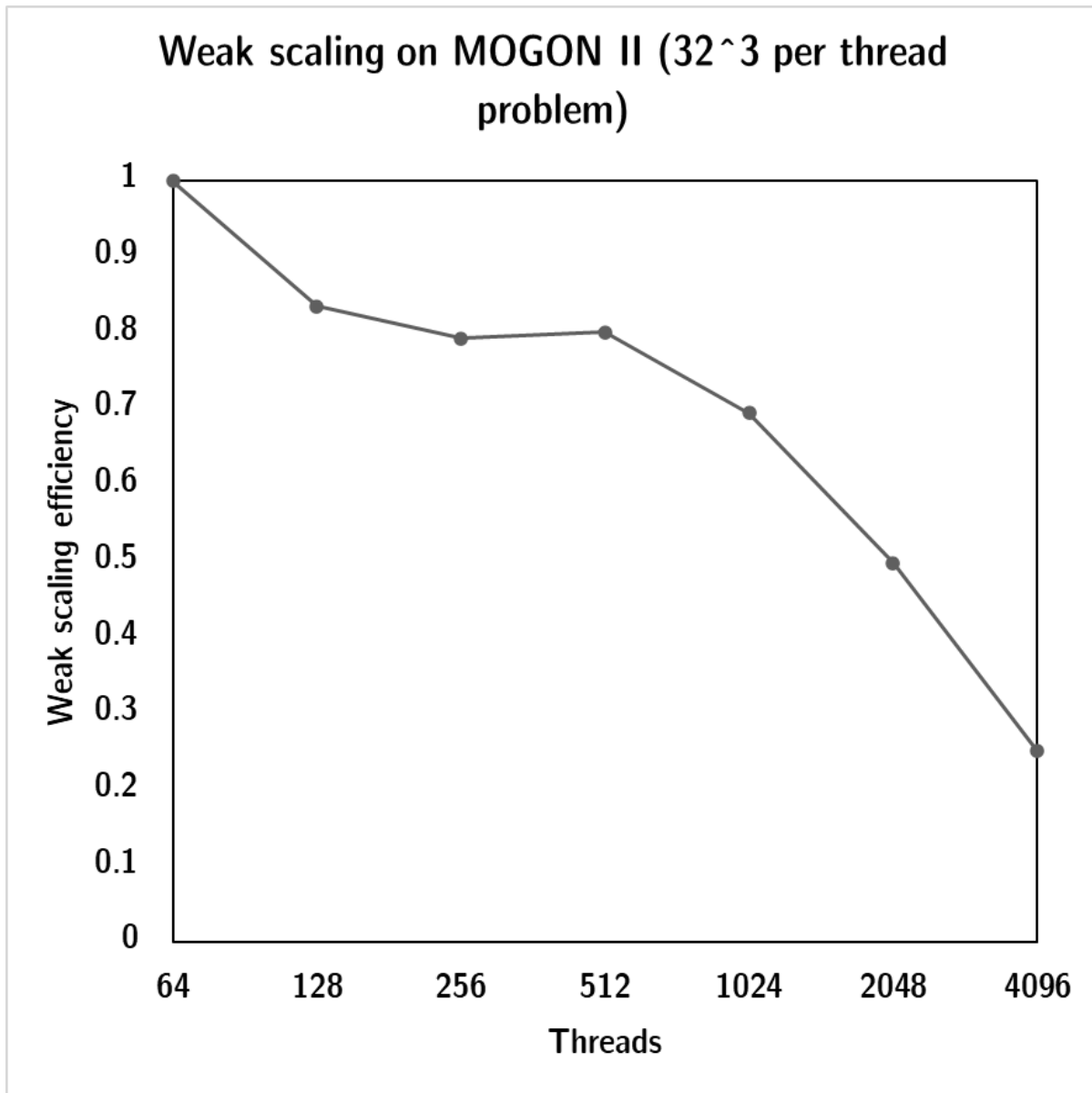


Figure 8-4 Weak scalability tests on MOGON2 using the same setup as in Figure 8-3.

8.2.3. Strong scaling

We performed strong scaling tests only on MOGON II, where each node has a maximum of 64 threads and 96 Gb/memory. For strong scaling we choose a model with resolution of 128^3 since it allows us to run a model from 1 thread without having “out of memory” on the available computing nodes but also such a resolution is enough for 1024 threads.

Figure 8-5 demonstrates ideal speedup up to 16 threads which then shows a decline for up to 64 threads. This corresponds to runs on 1 node. Runs from 128 threads and beyond use multiple nodes and show linear speedup increase up to 512 threads. Run with 512 threads demonstrates

speedup of 112.7 times in comparison to 1 thread run and speedup of 7.3 times in comparison to 64 threads run, which has 8 times less computing power. Runs with 1024 threads and more indicate low efficiency for a chosen problem.

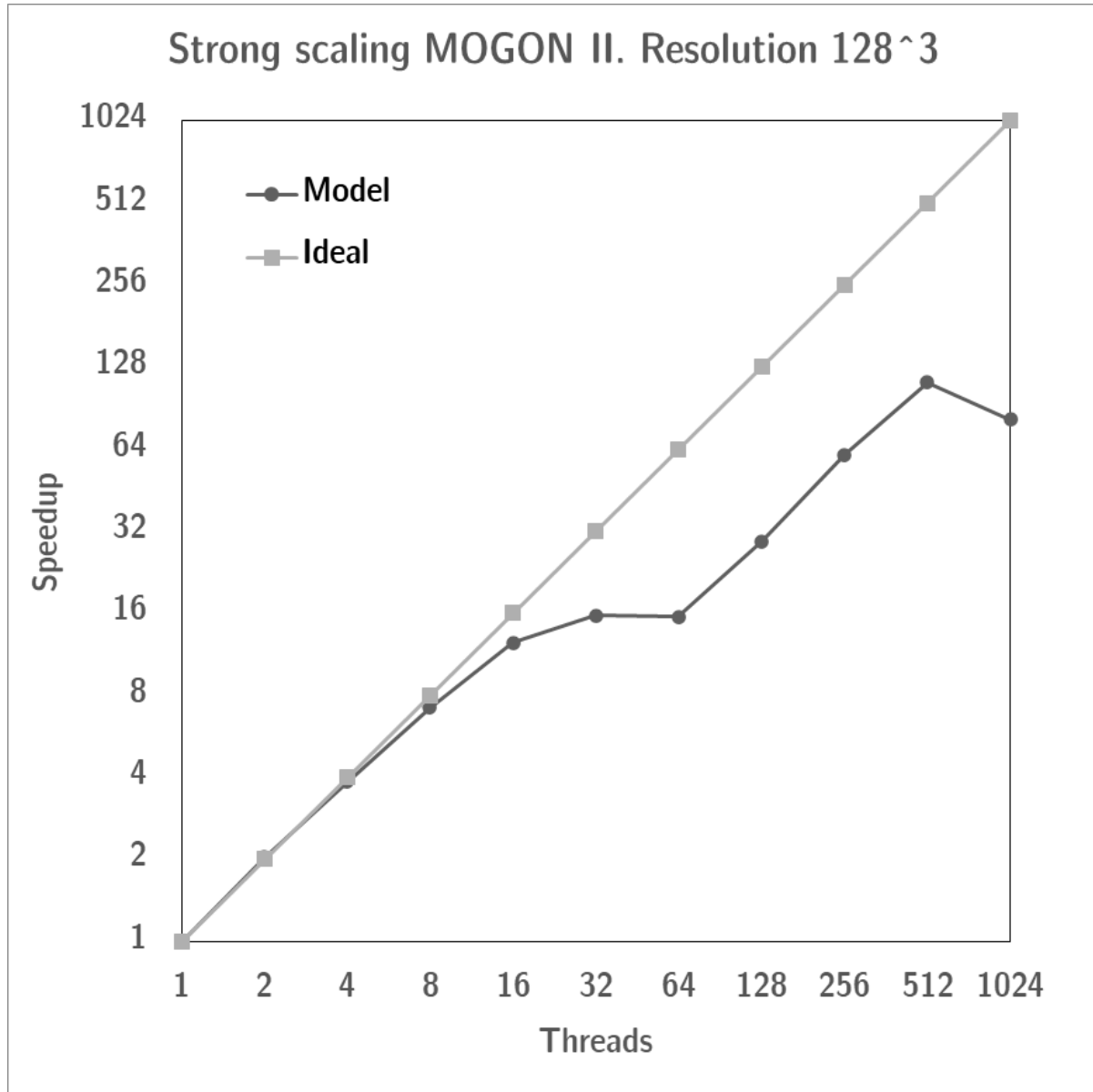


Figure 8-5 Strong scalability experiment for LaMEM using a resolution of 128^3 .

Strong scaling analysis for a problem of size 256^3 can only be done with a thread count of more than 128 threads, due to memory constraints. Moreover, calculation of such a big problem consumes large amounts of resources. We needed to use only part of the available threads per node to redistribute memory consumption to each node. It means that to run a problem with 128 threads we needed to allocate 4 nodes with 128 threads

each and use only half of the available threads in every node to fit in memory budget for a simulation.

In Figure 8-6 we can observe that we have linear scaling close to ideal for a run with up to 1024 threads. Speedup from running only at 128 threads is 7.6 times. Runs with more than 1024 threads show low efficiency.

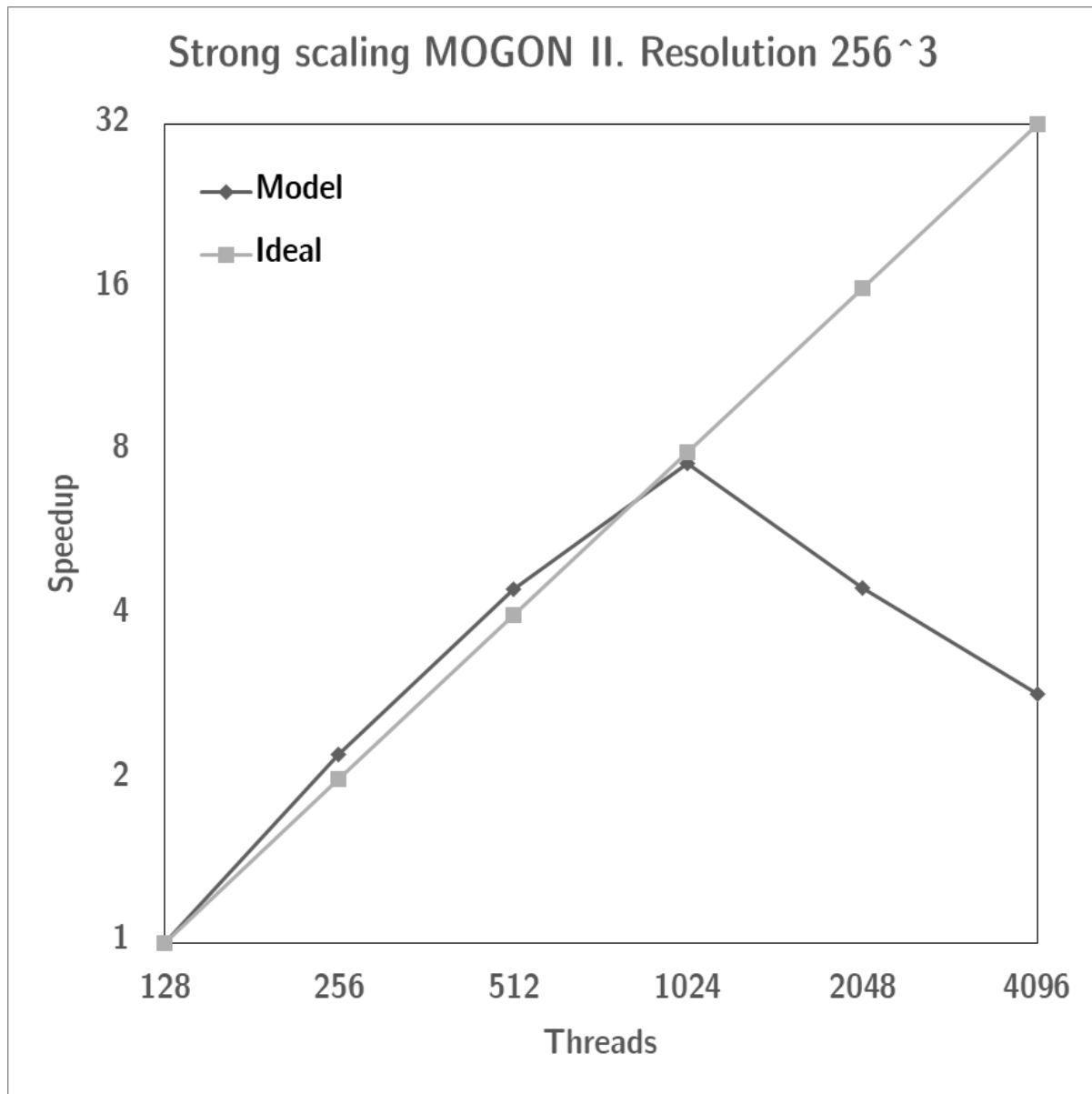


Figure 8-6 Strong scalability test of LaMEM using a resolution of 256^3 .

8.2.4. Discussion

3-D thermo-mechanical models of lithosphere with a resolution of 128^3 and 256^3 are very common since it requires less resources. However, it is relatively low resolution, even though the 3-D nature of the problem makes increasing the resolution a very difficult process. For example, a

3-D model with a resolution of 256x256x256 for a non-linear Stokes problem that involves calculating temperature, pressure and velocity has 67 million degrees of freedom. Doubled the resolution in all directions to 512x512x512 cells has 537 million degrees of freedom.

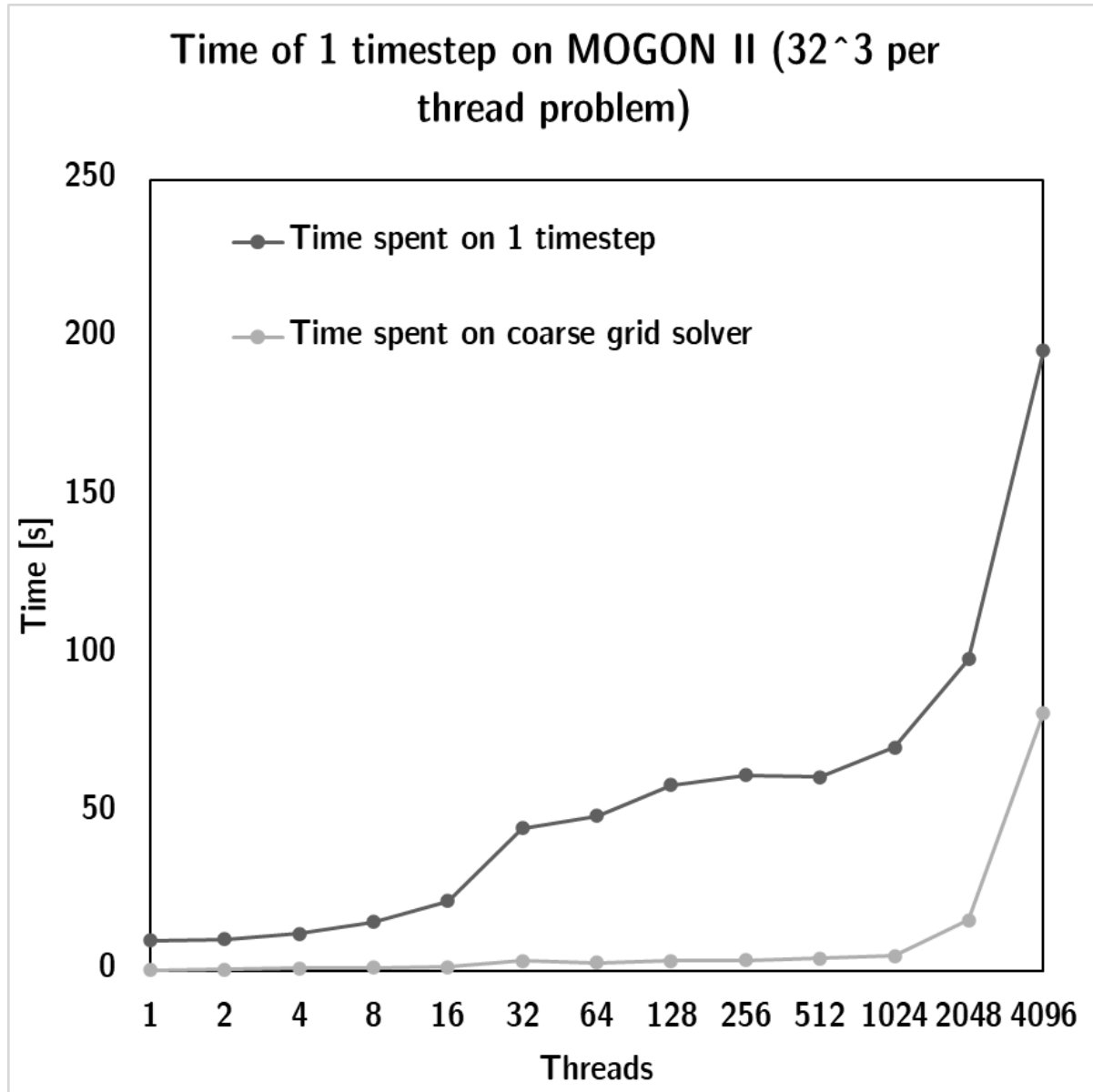


Figure 8-7 Time spend for the coarse grid parallel direct solver of LaMEM (grey line) versus the total time (black) during a weak scalability test, demonstrating that the coarse grid solve becomes a computational bottleneck at high resolutions.

The black line in Figure 8-7 shows results of weak scaling runs but now we demonstrate time for each iteration. The gray line shows the amount of time that was spent by solving the coarse grid using a parallel direct solver (superlu_dist) on the full amount of threads. As can be seen, the computational time for the coarse grid solve step is negligible for small amount of cores, but dominates the computational time when running

with >1024 threads, which explains the poor scalability beyond this point. Also we do not observe a serious impact of intra-node communication timings on run times during scaling. The average time for MPI barrier PETSc log view reported less than 10^{-5} sec for every simulation.

This suggests that moving forward, the biggest impact will be made by making the coarse grid solver more scalable. There are a number of possibilities for this in PETSc such as using PCTELESPOPE, which allows solving the coarse grid on only a subset of the processors. An alternative way is to use an algebraic multigrid solver for the coarse grid rather than a direct one which can be done with the GAMG preconditioner in PETSc. We will explore these options in the future.

8.3. Performance data acquisition

The code owners have provided Extrae traces for runs with 64, 128, 256, and 512 MPI ranks respectively. Initially, the size of traces was impractically large. The trace volume has been addressed in several ways. Firstly, the code was instrumented to mark regions of interest and to shutdown/restart trace recording. Secondly, traces do not contain hardware counters. And finally, the code was reconfigured to do only a fixed, small number of iterations rather than iterate to convergence. This renders the results of the benchmark runs physically inaccurate, but preserves the characteristics of the code with regard to parallel performance.

Even with these mitigation actions, the size of the trace data is very large and a full analysis of all traces could not be finished in time for this deliverable. In particular, we have analyzed only one out of two regions of interest which uses more compute time.

The focus of analysis (FoA) of this report is the user region *SNESsolve* as illustrated in Figure 8-8 and Figure 8-9. The FoA consists of 8-9 internal iterations, which are very similar to each other.

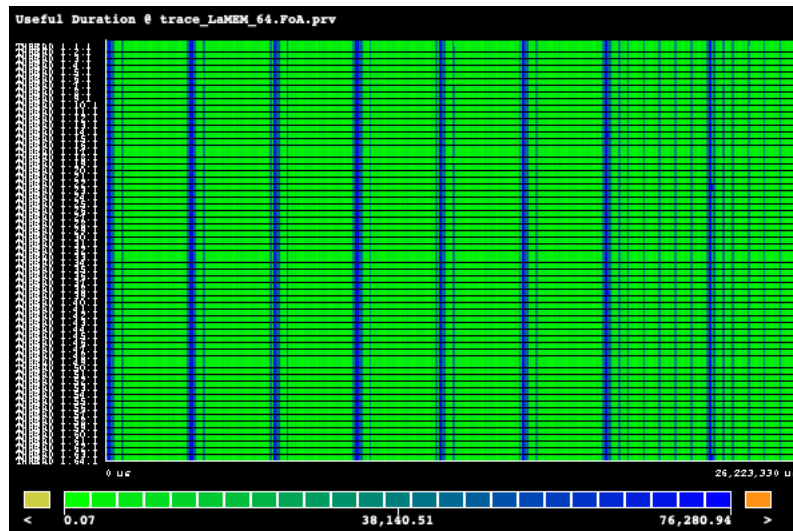


Figure 8-8 FoA of a run with 64 ranks. The colors indicate the duration of compute bursts between two MPI operations. The repetitive behavior is clearly visible.

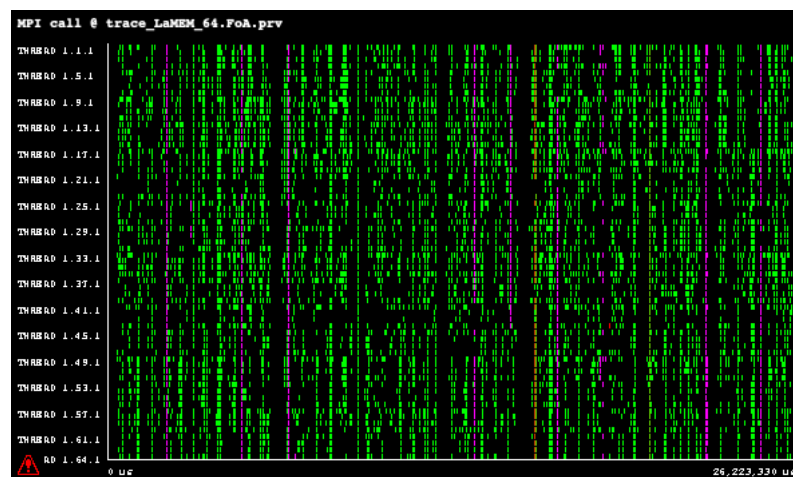


Figure 8-9 FoA of a run with 64 ranks. The colors indicate dominant MPI operations: MPI_Waitall (green), MPI_Allreduce (purple).

8.4. POP performance efficiency metrics

We have calculated POP performance efficiency metrics using the simple (MPI-only) model (see Table 8-1). Note, that due to the preliminary nature of the analysis, we have not further decomposed communication efficiency into transfer and serialization efficiency for some traces.

The parallel efficiency drops from a marginally acceptable value of 73% at 64 MPI ranks to a low value of 60% at 512 ranks. This decline is due to a declining communication efficiency (drops from 80% to 68%). Load balance efficiency is and remains good at values around 90%. Communication efficiency seems to be limited by serialization efficiency (quality of the communication patterns) rather than transfer efficiency, but the difference is only marginal for the available traces.

Table 8-1 POP performance efficiency metrics

	MPI Processes			
	64	128	256	512
Global efficiency	73%	59%	57%	36%
Parallel efficiency	73%	69%	67%	60%
Load balance	91%	90%	92%	88%
Communication efficiency	80%	77%	73%	68%
Serialization efficiency	91%	84%		
Transfer efficiency	88%	90%		
Computational Scalability	100%	86%	85%	60%

Also, computational scalability decreases with the number of ranks, i.e. actual computations (as opposed to MPI communication) takes more and more time. The effect is so large, that almost double the time is spent in computations at 512 MPI ranks with respect to the reference at 64 ranks.

In combination, the global efficiency is never above 73% and drops to a very low value of 36% at 512 MPI ranks.

Preliminary analysis of MPI communications show that a larger fraction of the time in MPI is due to *MPI_Waitall* operations, which could indicate an inefficient communication pattern or lacking MPI progress. Also, collective operations such as *MPI_Allreduce* are becoming more important at higher scales.

No further analysis can be presented at this time.

9. Code OpenPDAC

9.1. Brief description of code and use-case

OpenPDAC is an open source multiphase code for simulating volcanic gas-particle flows, is available via a public github repository¹⁴. OpenPDAC is developed as an OpenFOAM¹⁵ module and is based on multiphaseEuler, distributed with OpenFOAM. Compared to the original module, in OpenPDAC the kinetic theory equations for granular flows are modified to model more than one dispersed solid phase, and by introducing models for particle-particle drag. Furthermore, a Lagrangian library (one-way coupling with the gas-solid mixture) is used to model large solid particles. OpenPDAC also implements the initialization of the hydrostatic pressure profile, necessary for simulations on large domains.

Being based on OpenFOAM, OpenPDAC inherits its CPU¹⁶ parallelization implementation, which is based on a MPI-based domain decomposition, in which the geometry and associated fields are broken into pieces and allocated to separate processors for solution. The parallel running uses the public domain implementation of the standard message passing interface (MPI) by default, although other libraries can be used.

OpenFOAM-based CFD solvers for volcanology were already tested (and some of the bottlenecks to exascale already addressed) in ChEESE-1p (in particular, the ASHEE solver for Plinian volcanic eruption plumes). However, the new multiphase flow solver has new features, capability and a different numerical structure. With OpenPDAC, we aim at providing an effective and easily deployed numerical tool to be used by a broad volcanological community for addressing various volcanic hazards, in particular phreatic eruptions.

The case selected for the code audit is the initial phase of a simulation of a phreatic explosion at Vulcano island, Italy. This test case will be used as a test-bed for urgent computing simulations that might be used in case of reactivation of the volcano. To this aim, the mesh resolution and the time-to-solution are constrained by the needs of emergency response. These constraints will be defined in the course of the project.

¹⁴ <https://github.com/demichie/OpenPDAC-11>

¹⁵ OpenPDAC is based on the OpenFOAM distribution from the OpenFOAM foundation (<https://openfoam.org>).

¹⁶ At present no GPU implementations are available in the OpenFOAM distribution from the OpenFOAM foundation.

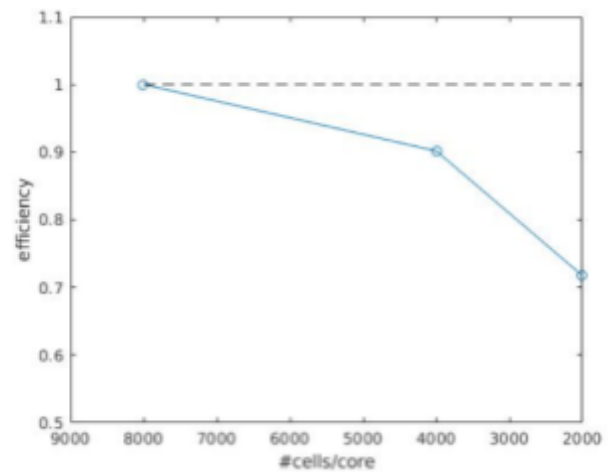
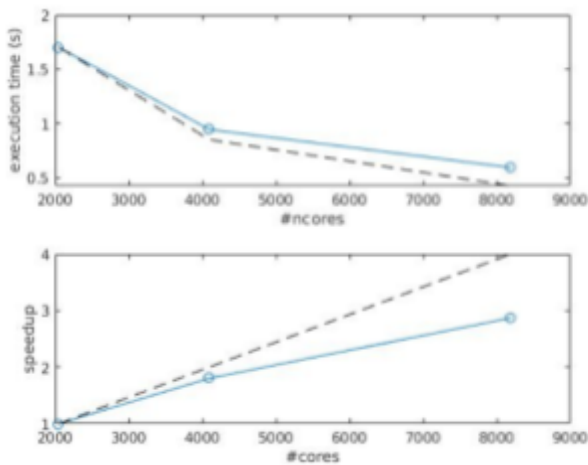
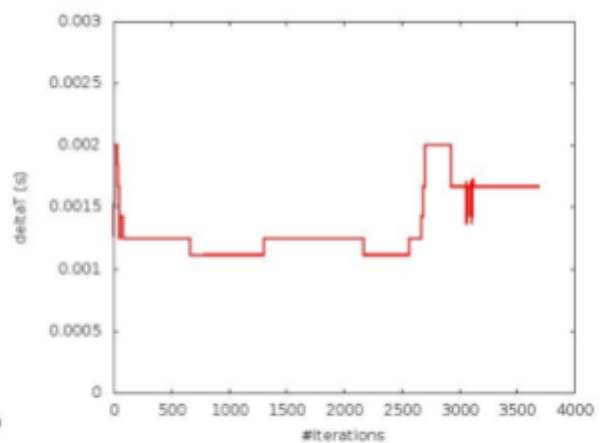
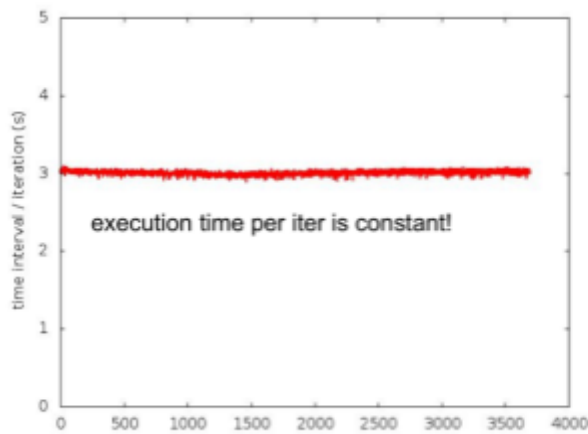
This test case uses a real, digitized topography and thus requires a complex unstructured grid for which proper load balancing is mandatory. This case is also important because high-resolution simulation of phreatic explosions is the focus of Pilot Demonstrator 6 of ChEESE-2P.

9.2. Scaling behavior

The computational speed of OpenPDAC is tested using as a metric the mean execution time per single iteration/time step. In preliminary tests, we observed that this is a rather general and robust metric since it is independent of details of the simulation setup used to test the scalability (such as the time step and the size or topology of the mesh). This metric is also proven to remain constant throughout the entire duration of a full simulation, even if the time step is not kept fixed but varies according to the CFL condition (Figure 9-1). Therefore we measure the computational speed of the application by computing the mean value of the execution time per time step on just a hundred of iterations, instead of the thousands typically required to reach the end time of a full run.

The strong scaling test (execution time, speedup and efficiency) for a mesh size of $\sim 16\text{M}$ cells has been performed on the CPU partition of the machine LUMI using up to 64 nodes, that is 8192 cores (Figure 9-2). MPI communications start to affect the parallel scalability when reaching 4000 cells/core. However, for 2000 cells/core (8192 cores) the parallel efficiency (0.7) is still acceptable and the application achieves the top computational speed of $\sim 430\text{k cells}/(\text{s node})$.

Although not reported here, a super-linear scaling is observed with fewer cores (< 2048), and in general, in any strong scaling test when considering runs with more than ~ 9000 cells/core. This behavior has already been reported in the literature for OpenFOAM applications [Brogi et al., 2024]. It is most likely due to the presence of levels of cache that, as the number of nodes grows, increasingly reduces the memory bandwidth bottleneck providing an additional speedup related to an increased computational intensity per core.



9.3. Performance data acquisition

After inspection of the scaling curves, the code owners intended to provide Extrae traces for 512 - 4096 MPI ranks. However, the size of the trace was very big. Even reducing the number of iterations and discarding all but the minimum necessary trace information, the trace for 512 MPI ranks is still larger than 30GB. Determining all performance metrics for such large traces is prohibitively costly and would require compute resources much larger than the original code execution.

For MPI-only codes, such as this version of OpenPDAC, a subset of POP metrics can be calculated from MPI profiles provided by the tools DLB/TALP and mpiP. However, both tools produced errors when applied to OpenPDAC. These issues could not be solved as of today.

The remainder of this chapter will present preliminary POP metrics for the only available trace of 512 MPI ranks.

The focus of analysis (FoA) excludes initialization and finalization phases and consists of four timesteps as illustrated in Figure 9-3 below. The first time step seems to be slightly different than the other four. Each of the time steps consist of five internal iterations. Further repetitive substructure at a smaller scale is visible.

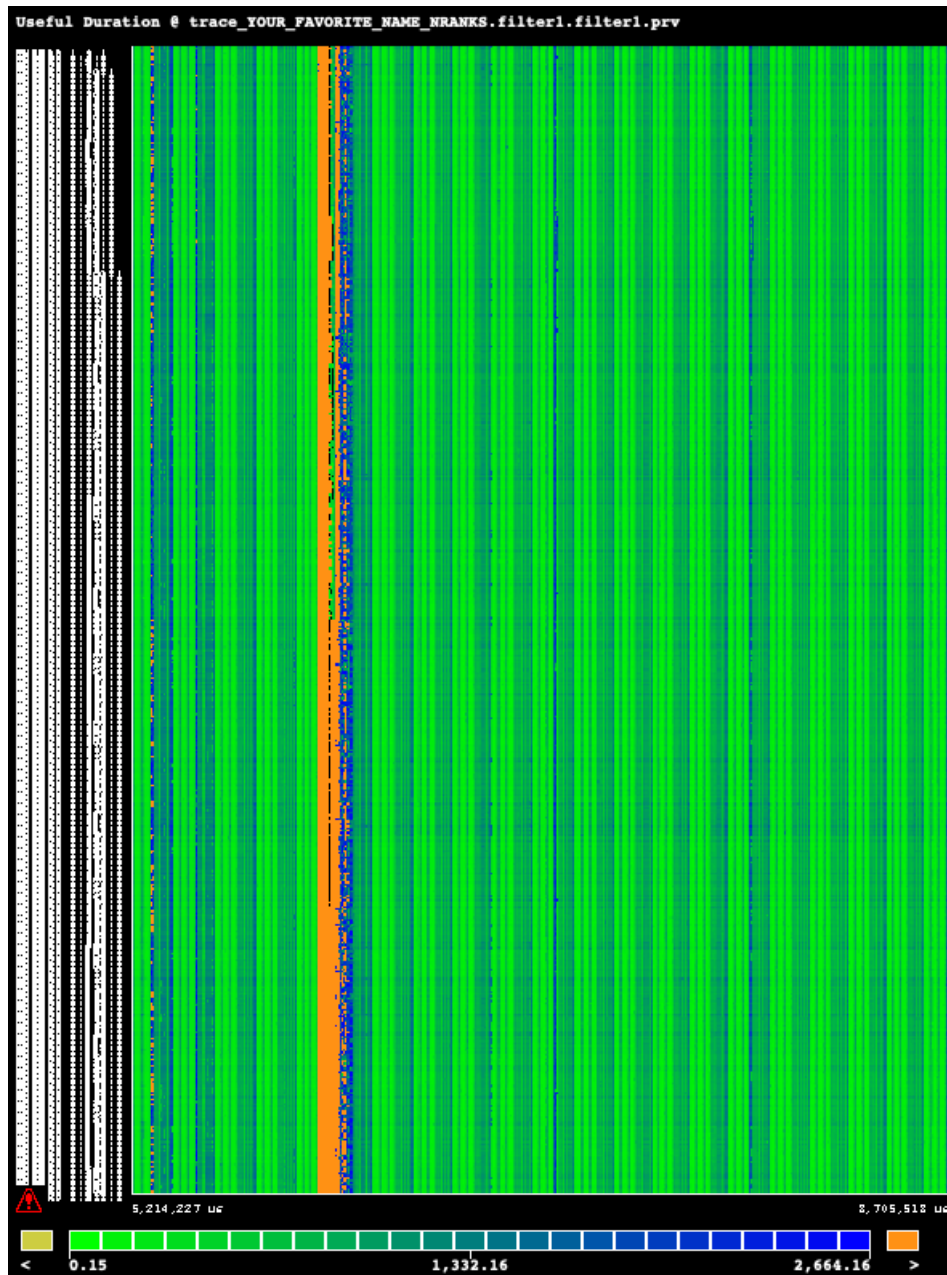


Figure 9-3 FoA of a run with 512 MPI ranks . The colors indicate the duration of compute bursts between two MPI operations and black indicates MPI operations.

9.4. POP performance efficiency metrics

We have calculated POP performance efficiency metrics using the simple (MPI-only) model (see Table 9-1below). Note, that we could not apply the

standard rigorous methodology for calculating POP metrics, but had to make some simplifying assumptions. This will have some impact on the exact numerical values, but not on the general picture as such. The impact is largest for serialisation and transfer efficiency.

Table 9-1 POP performance efficiency metrics

	MPI Processes			
	512			
Global efficiency	36%			
Parallel efficiency	36%			
Load balance	74%			
Communication efficiency	49%			
Serialization efficiency	69%			
Transfer efficiency	71%			
Computational Scalability	100%			

Parallel efficiency is only 36% which means that the code spends only roughly a third of its time doing actual computations. The main reason for low parallel efficiency is the low communication efficiency of around 50%. Load balance efficiency is better at 74%. Communication efficiency can be broken down further into serialisation (which is a measure of the quality of the communication pattern) and transfer efficiency (actual transfers of data). In this case, however, both contributions seem to have similar magnitude at around 70%.

The numbers above show that for this particular use case and number of MPI ranks, OpenPDAC already wastes a significant amount of time due to all three of load imbalance, poor communication patterns, and large volume of data transfers, all of which contribute roughly similar shares.

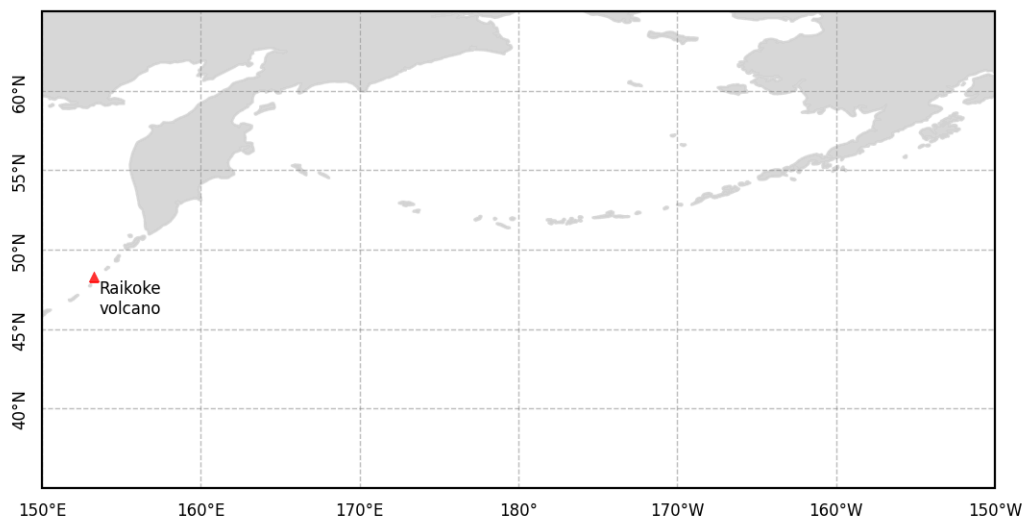
No further analysis can be presented at this time.

10. Code FALL3D

10.1 Brief description of code and use-case

FALL3D is an Eulerian model for the atmospheric transport and ground deposition of particles, radionuclides, and aerosols, including volcanic

tephra, ash and gasses (Folch et al., 2020; Prata et al., 2021). The model solves a set of advection-diffusion-sedimentation (ADS) equations on a structured grid using a high-resolution central-upwind scheme derived from finite volumes, which minimizes numerical diffusion even in presence of sharp concentration gradients and discontinuities. The following Figure 10-1 shows the computational domain for the test audit case, which simulates atmospheric tephra dispersion from the 2019 eruption of the Raikoke volcano (Kuril archipelago). The use cases presented below (for audit and strong scaling) are based on this case of the FALL3D test-suite



All tests presented in this document were conducted on the CINECA Leonardo cluster, equipped with 3,456 computing nodes. Each node has 32 Ice Lake cores running at 2.60 GHz, 512GB of RAM, and is equipped with 4 NVIDIA Ampere A100 GPUs, each one with 64GB.

10.1.1. Use case scaling

This use case has been designed to push a single node of the Leonardo cluster with its 4 GPUs to the limit, enabling strong scalability analyses under a high computational load.

The grid resolution is 1000x1000x100 cells in the X, Y and Z axes respectively, considering 5 different types of particle classes (bins). In terms of memory, this resolution is close to the GPU memory limit (64GB for Leonardo machine) as the baseline for the strong scalability analysis is 1 single GPU. With the aforementioned dimensions, the Raikoke scenario has been simulated over a period of 1 hour. To prevent excessive initial

zero values, the simulation is restarted from a precomputed checkpoint established in a prior execution that spans from 0 hours to 18 hours. The simulation period then focuses on the timeframe between 18 and 19 hours. For this reason, the data loading strategy is being modified for the next code release (as outlined in Section 10.6) to address this limitation.

10.1.2. Use case audit

This use case has been specifically crafted for trace analysis using the POP tools and metrics. Its design is focused on a faster execution, facilitating a comprehensive examination of the entire code stages for potential bottlenecks or optimization opportunities. The mesh used is the same as described in use case 1 (scaling) but considering one particle type (bin). The simulation time is the same as in use case scaling, lasting for one hour also.

A private Git group has been created within the FALL3D main repository to store all the tests,¹⁷ with a subgroup for each target system (currently Leonardo and LUMI) as shown in Figure 10-2 below.

At the root of each use case, a folder named "shared" contains the scripts that prepare the subfolders for code runs at various levels and the files that are shared. When an execution is being prepared using the specific script, a folder (binary folder) is created with an identifier. Into this folder, a JSON file is generated by the script, detailing the necessary metadata values to easily replicate that run in the future (see Figure 10-3).

Additionally, a script retrieves data from the JSON file in the binary folder and various details from the runs in the different runs folders, generating a final JSON that can be processed locally to simplify the task and automatically generate a report. This report contains all the binary options and the details for each code run, and uses the data contained in the report to create plots with the computing time, the speedup and the parallel efficiency.

¹⁷ <https://gitlab.com/fall3d-suite/tests>

```

1  {
2    "test": {
3      "testType": "test_scalability",
4      "testOwner": "Horibert Pascual",
5      "note": "Copy and modify this template in each new test_scalability_ID folder",
6      "comment": "Test to write in the log the initialisation time"
7    },
8    "code": {
9      "origin": "decube83/fall-3-d-gpu-cicd.git",
10     "version": "8.2.1-gpu",
11     "branch": "/cicd/hpasual",
12     "commitID": "30349dce61c8ba3bd3f20ea79041c543a1471f76",
13     "commitAuthor": "decube83 <hpascual@geo3bcn.csic.es>",
14     "commitDate": "Mon Nov 06 09:15:29 2023 +0100"
15   },
16   "system": {
17     "host": "Leonardo@CINECA",
18     "partition": "booster",
19     "nodeCPU": "32 cores Intel Ice Lake ",
20     "nodeGPU": "4 NVIDIA Ampere 64GB",
21     "nodeRAM": "512 GB"
22   },
23   "modules": {
24     "fortran": "nvhpc/23.1",
25     "cuda": "cuda/11.8",
26     "mpi": "openmpi/4.1.4--nvhpc--23.1-cuda-11.8",
27     "hdf5": "hdf5/1.12.2--openmpi--4.1.4--nvhpc--23.1",
28     "netcdf-c": "netcdf-c/4.9.0--openmpi--4.1.4--nvhpc--23.1",
29     "netcdf-fortran": "netcdf-fortran/4.6.0--openmpi--4.1.4--nvhpc--23.1",
30     "netcdf-parallel": "parallel-netcdf/1.12.3--openmpi--4.1.4--nvhpc--23.1",
31     "zlib": "zlib/1.2.13--gcc--11.3.0"
32   },
33   "compilation": {
34     "FCFLAGS": "FCFLAGS=-g -gpu=pinned -cuda -fast -Minline -acc",
35     "FFLAGS": "FFLAGS=-g",
36     "LDLAGS": "LDLAGS=-gpu=pinned -cuda -acc",
37     "enableGPU": "--with-gpu=pinned",
38     "enableACC": "--with-acc=yes",
39     "precision": "--with-r4=no",
40     "binary": "/Leonardo_work/EUHPDC_002_008/TESTS/test_scalability/shared/binaries/Fall3d.gpu.231106.r8.init_time.x"
41   }
42 }
43

```

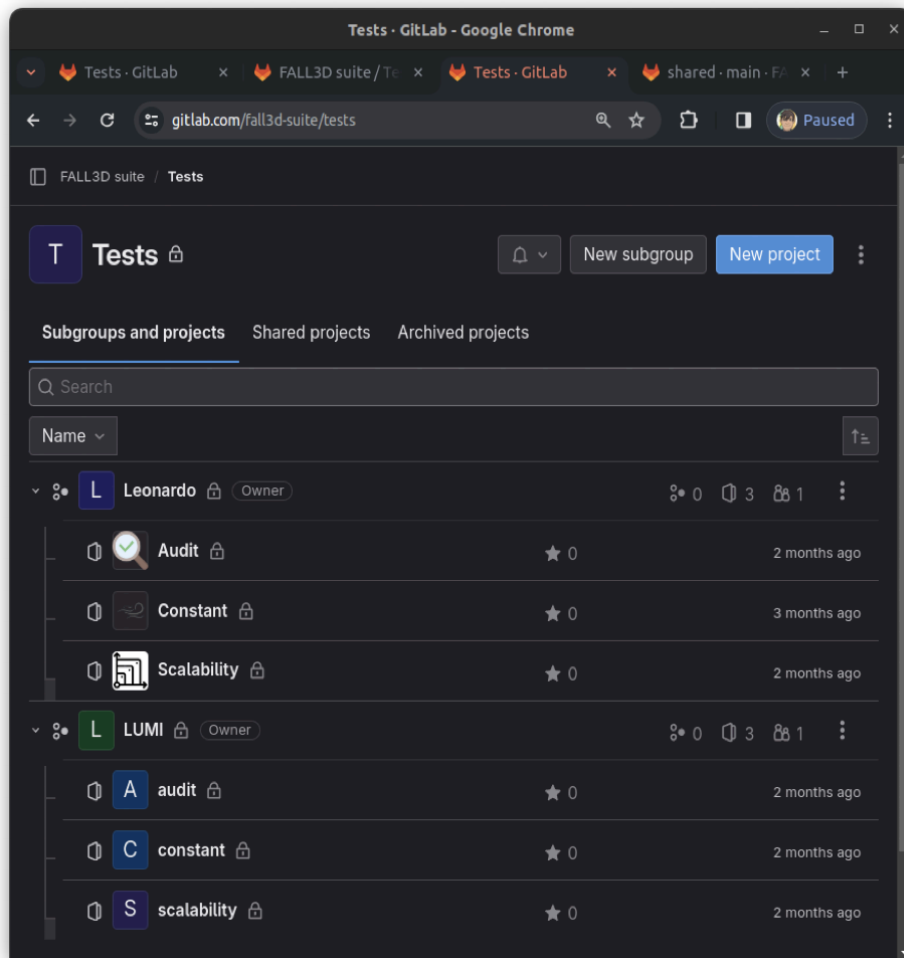


Figure 102 Group containing different system's tests

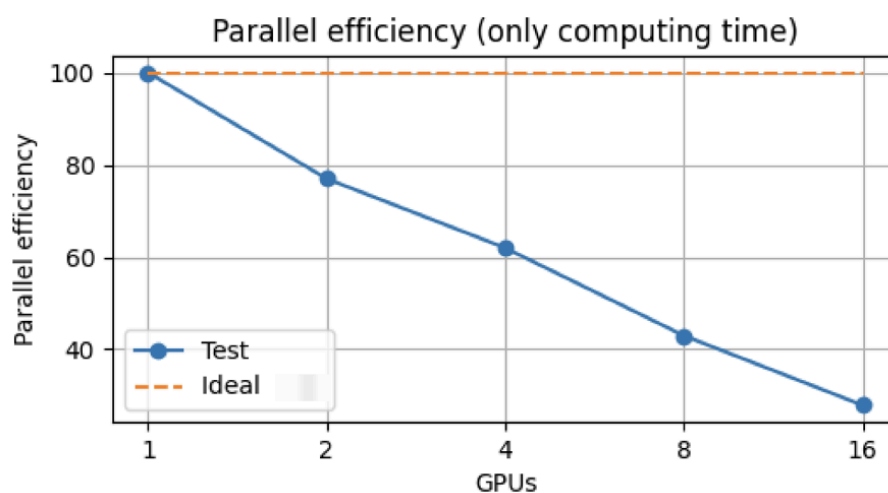
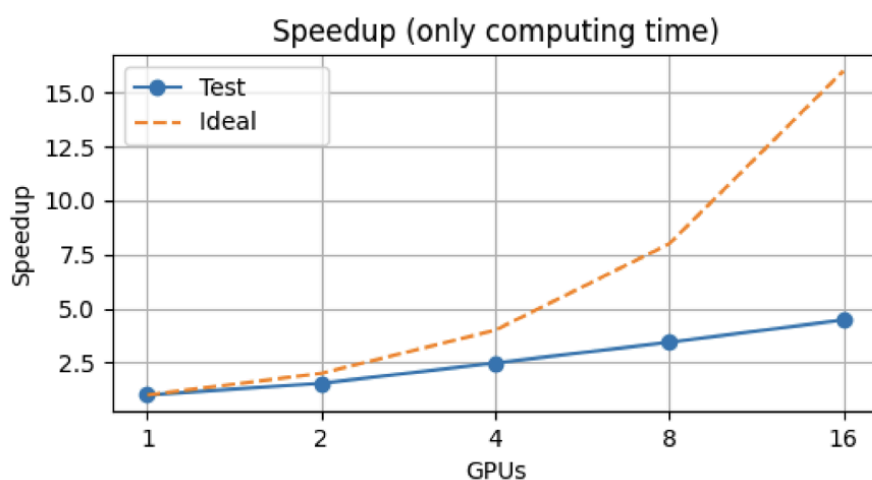
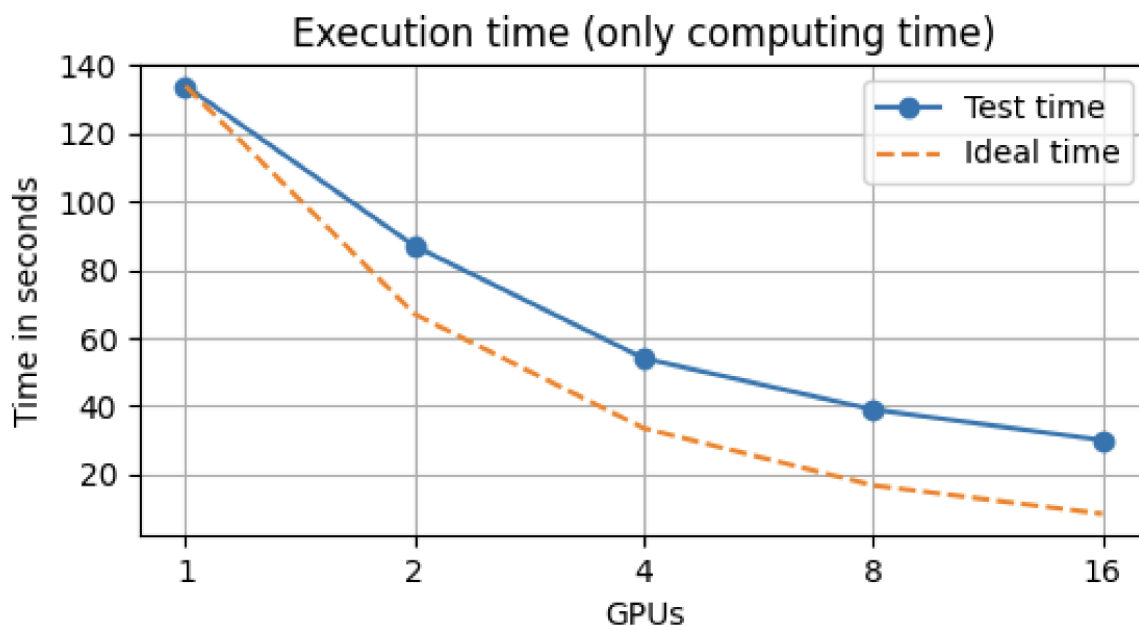
10.2. Scaling behavior

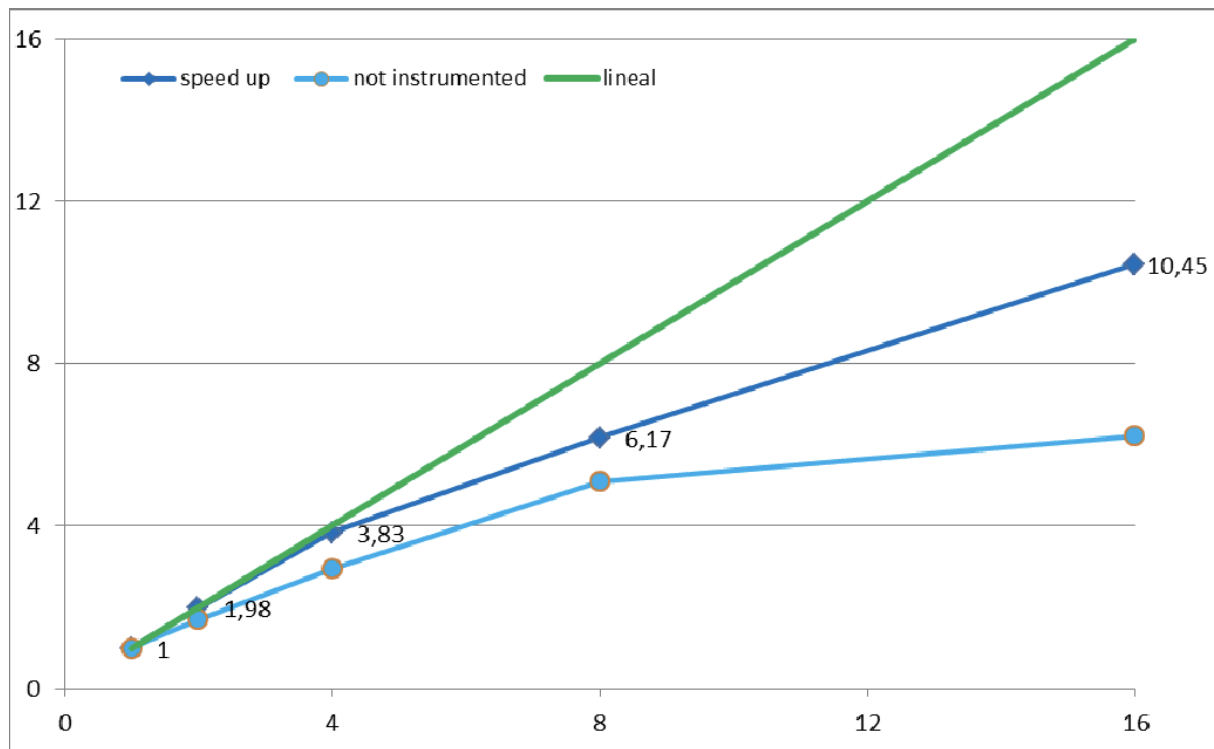
During ChEESE-1P, the CPU version of FALL3D (version 8.2) was optimized up to level 2 (i.e. code able to perform full capability runs or capacity workloads on petascale machines with >90% parallel efficiency proven in more than 100 nodes), and the code was accelerated using OpenACC, improving drastically the time to solution. However, no optimisation effort was done on the GPU version. This is a major objective in ChEESE-2P and this first audit round aims at setting the current baseline of the GPU version 8.2.

The scalability tests were conducted using the scalability use case 1 as follows:

- 1 GPU: no domain decomposition is performed
- 2 GPUs: domain divided in half along the x-axis.
- 4 GPUs: domain partitioned into two sections along both the x and y axes.
- 8 GPUs: domain divided into halves along all three axes: x, y, and z.
- 16 GPUs: domain split into four segments along the x-axis, two along the y-axis, and two along the z-axis.

Figure 10-4 illustrates the computation time and the ideal performance for execution on 1, 2, 4, 8, and 16 GPUs (excluding initial I/O overhead). The speedup and parallel efficiency are depicted in Figure 10-5. The POP team observed that during the preprocessing stage, a significant amount of time was spent by the code launching small kernels to GPUs in sets larger than one GPU. These kernels were surrounded by MPI communications, resulting in prolonged waiting times between GPU pairs. Consequently, the code was instrumented to gather metrics specifically for the FOA (solver cycles, see Section 10.3). In Figure 10-6, the ideal speedup is depicted in green, the speedup of the FoA code region in dark blue, and the previously presented speedup (all code regions) is shown in light blue.





10.3. Performance data acquisition

There was no need to discard any specific region for the performance analysis, as the execution with a single GPU only took approximately one minute and generated manageable size traces (300MB). Additionally, mpiP was utilized to generate a report employing strong scaling (Section 10.4). Traces for deep analysis were obtained using the Extrae tool from BSC, with some modifications and some handcrafted work from the POP team side to capture OpenACC kernels launched in the GPUs under MPI code regions.

Following a preliminary performance optimization analysis conducted by the POP team, a joint meeting was held. The POP team highlighted the issue affecting our code commented in Section 10.2. This problem was easily identifiable, and we agreed to concentrate our analysis on the main iteration zone of code solver (focus-of-analysis, FoA). This approach aims to establish a clear baseline and focus on enhancing the code within that specific area, with the primary goal of maximizing portability across GPUs from various vendors, as the code is now oriented to NVIDIA devices.

10.4. POP performance efficiency metrics

The results obtained using the mpiP tool in a complete execution of the audit case are presented in Table 10-1. The complete execution includes I/O time, initialization, preprocessing and computation.

Table 10-1 POP performance efficiency metrics for the full audit case (computational scalability computed with strong scaling)

	MPI Processes and GPUs				
	1	2	4	8	16
Global efficiency	100%	57%	36%	21%	12%
Parallel efficiency	100%	64%	42%	26%	17%
Load balance	100%	73%	48%	29%	19%
Communication efficiency	100%	88%	87%	90%	90%
Computational Scalability	100%	89%	87%	80%	70%
Elapsed time	80 s	70 s	55 s	47 s	43 s

Table 10-2 shows the execution times for the same use case without any profiling tool overhead. From the log file, it is determined that 97 iterations of the main loop are performed. This information allows for a reasonably accurate calculation of iterations per second for each distinct set of GPUs.

Table 10-2 Iterations by second running the code without any analysis tool

GPUs	Time	Total iterations	Iterations/second
1	80	97	1.21
2	70	97	1.39
4	55	97	1.76
8	47	97	2.06
16	42	97	2.31

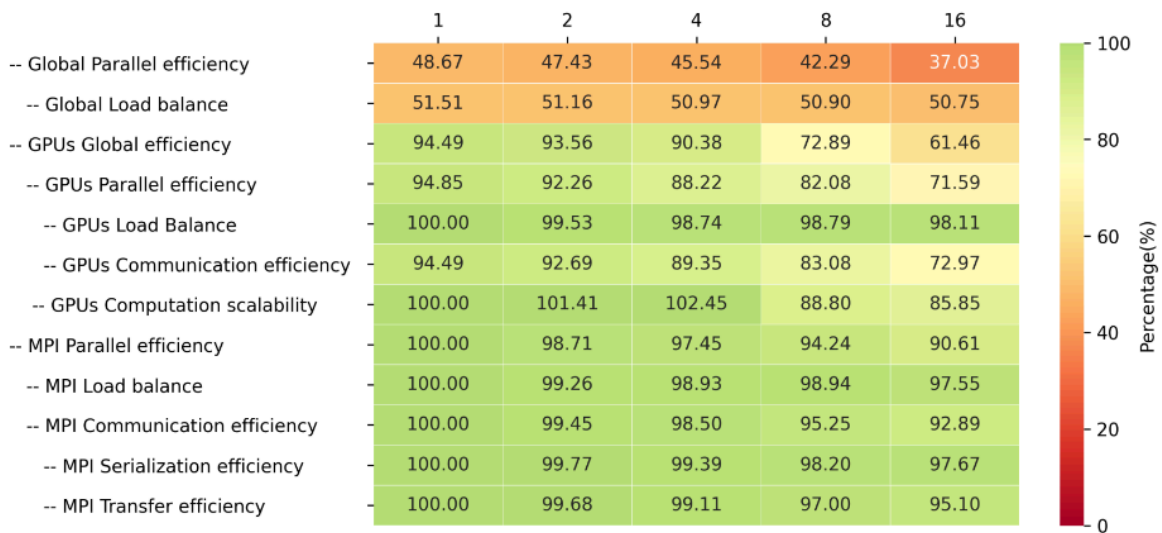
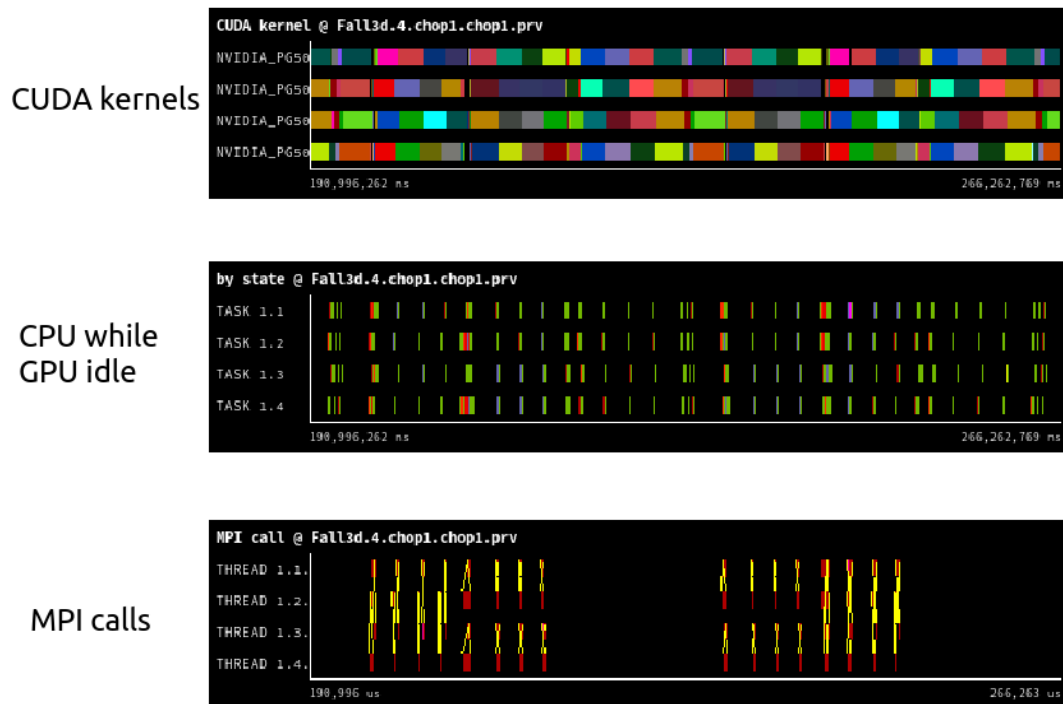


Figure 10-7 POP metrics FOA region

Finally, the metrics provided by the POP team focusing only on the FoA region are presented in Figure 10-7 and reveal 2 bottlenecks of different origin:

1. The FoA region reveals a good GPU global efficiency (90.38%) up to 4 GPUs. Issues arise when the computational domain is divided along the Z-axis, as this leads to a substantial increase in communication overhead (72.89%).
2. On the other hand, the low values of the global load balance (about 50%) are attributed to FALL3D employing synchronous OpenACC operations, causing CPUs to wait for GPU kernel completion, resulting in CPUs having low utilization when the GPUs are working. This is further illustrated in Figure 8 below.



10.5. Communication

In FALL3D, the user specifies the computational domain division when running the code, which may not always be the optimal one. For the scaling and audit analyses the divisions employed are outlined in Table 10-3.

Table 10-3 Domain decompositions used in performance analysis

GPUs	X	Y	Z
1	1	1	1
2	2	1	1
4	2	2	1
8	2	2	2
16	4	2	2

Working with a grid of dimensions 1000x1000x100 and dividing the domain along the Z-axis significantly increases domain communications, given that the division in the Z-axis represents the largest area in this case. Figure 10-9 illustrates the communications of the code with 4 GPUs on the left side, demonstrating an important near duplication of communications compared to the execution with 16 GPUs depicted on the right side (the XY plane is much larger than the XZ and YZ planes).

	MPI_Wait	MPI_Irecv	MPI_Isend	MPI_Allreduce
Total	1,024	512	512	16
Average	256	128	128	4
Maximum	256	128	128	4
Minimum	256	128	128	4
StDev	0	0	0	0
Avg/Max	1	1	1	1

4 GPUs

	MPI_Wait	MPI_Irecv	MPI_Isend	MPI_Allreduce
Total	7,168	3,584	3,584	64
Average	448	224	224	4
Maximum	512	256	256	4
Minimum	384	192	192	4
StDev	64	32	32	0
Avg/Max	0.8750	0.8750	0.8750	1

16 GPUs

Figure 10-9 MPI calls for 4 GPU and 16 GPU runs

An illustration of the domain decomposition problem is depicted in Figure 10-10, wherein it is evident that with each division along the X-axis depicted in red, assuming the transmission of only one element, one hundred thousand data points should be sent. The same happens for each division along the Y-axis represented in purple. However, when performing the division along the Z-axis in yellow, communication would involve one million elements. It is important to note that this scenario may not always hold, as the shape of the domain is subject to change depending on the user needs.

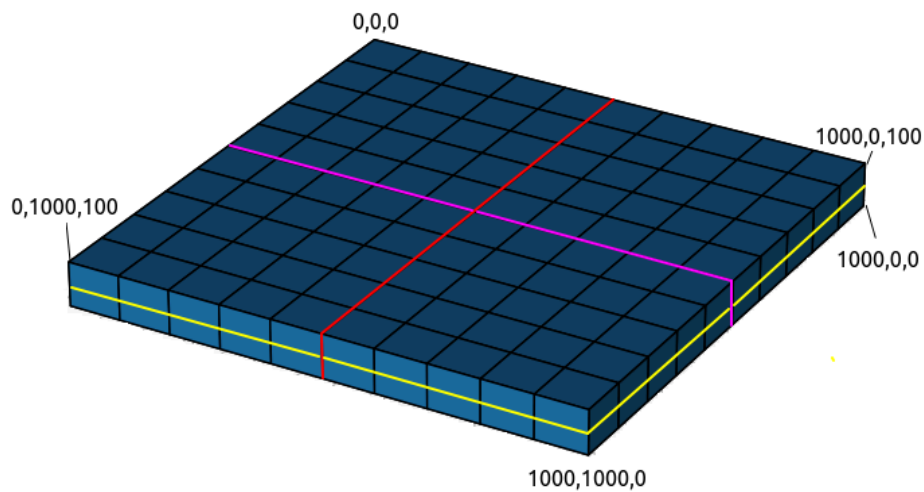
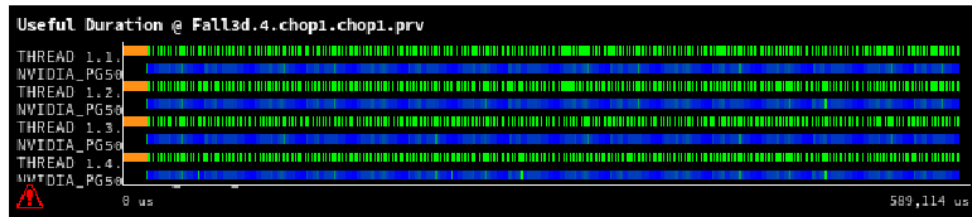


Figure 10-10 Computational domain decomposition example for $x=2$ $y=2$ and $z=2$

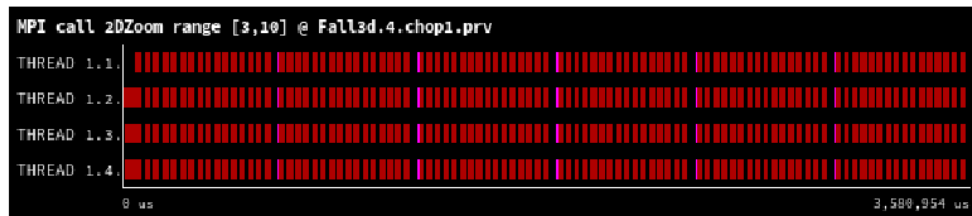
10.6. Computational scalability

The scalability analysis shows an almost perfect trend up to 4 GPUs (95.7% parallel efficiency). Figure 10-11 presents a segment of the execution with 4 GPUs (thus avoiding decomposition along the Z-axis).

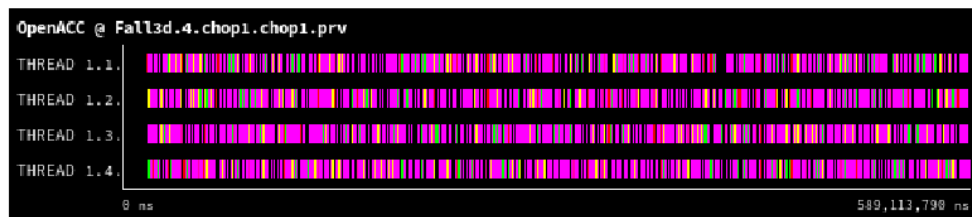
Computations



MPI calls



OpenACC



CUDA

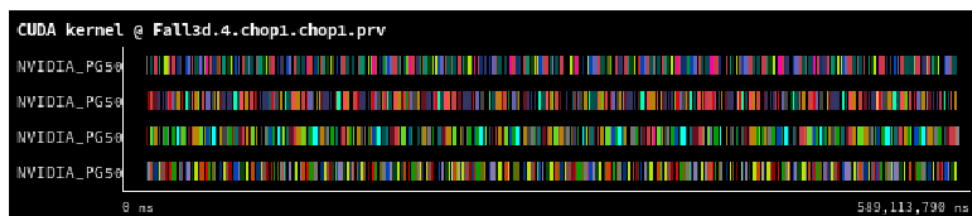


Figure 10-11 4-GPU run FOA region

The total time spent on MPI is only 2.5%. The primary issue, as previously discussed, lies in CPUs remaining idle while awaiting the completion of kernels dispatched to GPUs. The time spent for CPUs in the ACC runtime is 94%, with 93% of that time dedicated to waiting for kernels. Kernels executed on GPUs account for 88% of the overall run time.

10.7. Observations and recommendations

In the current baseline code version (8.2), all the meteorological data needed for a simulation is loaded into memory before the simulation starts. When dealing with GPUs, which typically have more constrained memory when compared to CPUs, this limits the size of each subdomain. To address this limitation, a strategy is being developed to load chunks of meteo data when needed. Users will have the flexibility to define the size of these chunks, aligning them with the hardware specifications where the code is executed from the execution configuration file. Note that smaller data chunks will slightly increase execution time due to the recurrent file access operations. However, this approach will facilitate the execution of

the code across a wider spectrum of computing platforms, ranging from personal computers to HPC systems.

After the audit, 3 deficiencies have emerged:

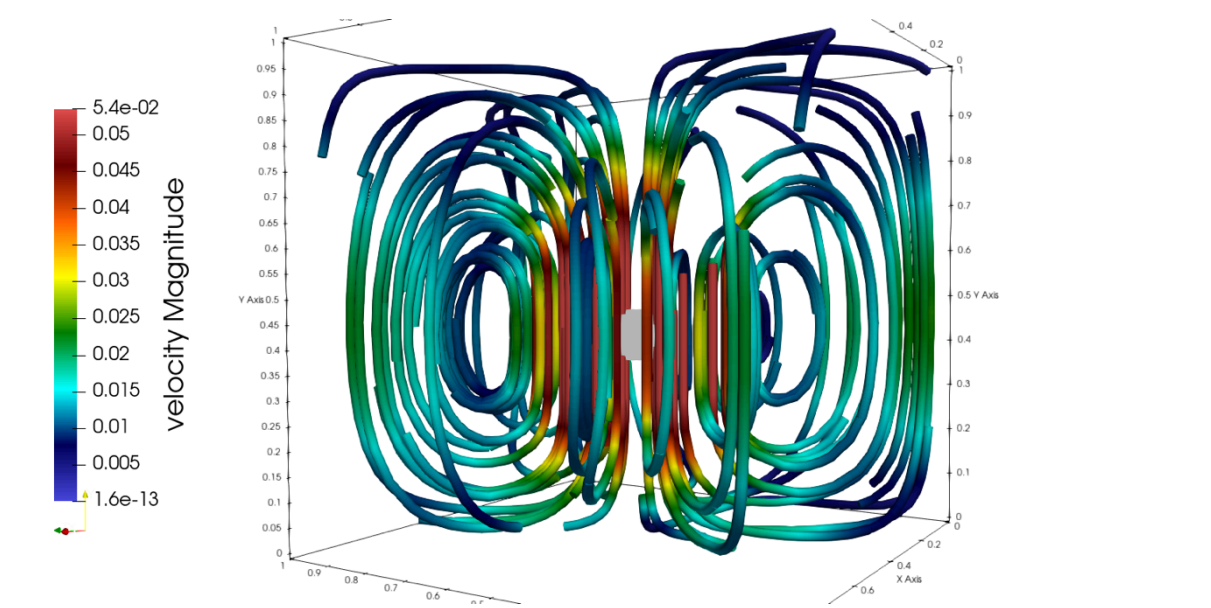
1. Preprocessing phase (Figure 10-6). Addressing this matter has the potential to enhance runtime efficiency with relatively minor modifications.
2. Communications. Regarding the communication problem, they advise adopting an alternative approach for future analyses. Nonetheless, the Fall3D developers have planned in work to identify a solution that optimizes this aspect, as it may give a significant improvement to code.
3. Load balance. The POP team has also noted that the CPUs are idle while awaiting the completion of kernels launched on GPUs, this fact significantly impacts the parallel efficiency. The optimal solution to this issue would involve modifying the code to leverage asynchronous OpenACC processing.

11. Code pTatin3D

11.1. Brief description of code and use-case

pTatin3D is a software package designed for studying long time-scale processes relevant to geodynamics. The original motivation for this development was to provide the community with an open-source toolkit capable of studying high-resolution, three-dimensional models of lithospheric deformation. It provides fast, parallel scalable matrix-free definitions for the Stokes operators which are utilized by a hybrid geometric-algebraic multi-grid preconditioner.

The matrix-free operators are designed to accelerate the matrix-vector product on CPU using AVX accelerator and on GPU with CUDA and HIP accelerators. The goal in WP2 is to enhance the HIP kernels optimization and scalability. We also plan to support HIP for assembled matrix operators to increase the part of the work performed on GPU instead of CPU.

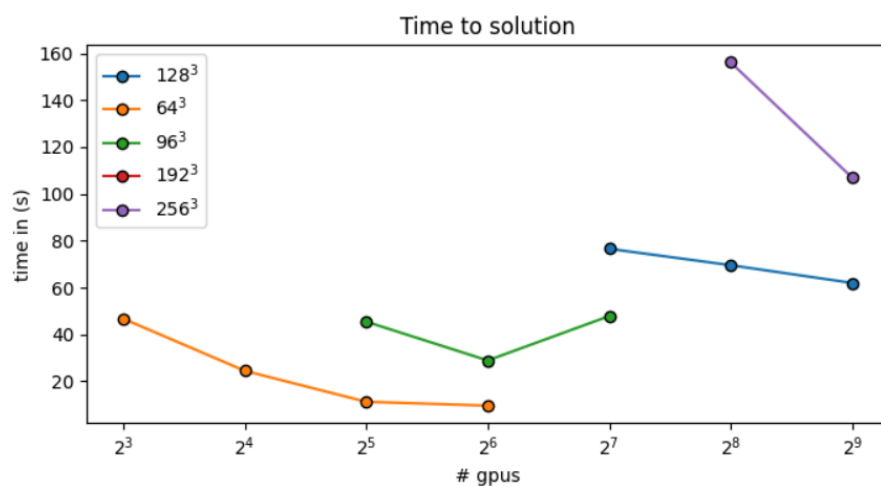


To analyze code performance and scaling behavior we selected the use case of the Stokes' law (Figure 11-1). This test can be easily adapted to stress the solver by increasing the viscosity and/or density contrasts between the sinking ball and the surrounding fluids. In addition, this use case has an analytical solution that we can compare to within a single time step.

11.2. Scaling behavior

Scaling curves show that for a given mesh size, the time to solution decreases with an increasing number of GPUs. However, the scaling is not linear and the performance improvement reduces when the number of gpus becomes too large for a given mesh size.

This behavior may be due to an increasing time of communication that starts to become too large with respect to the actual work performed by each process.



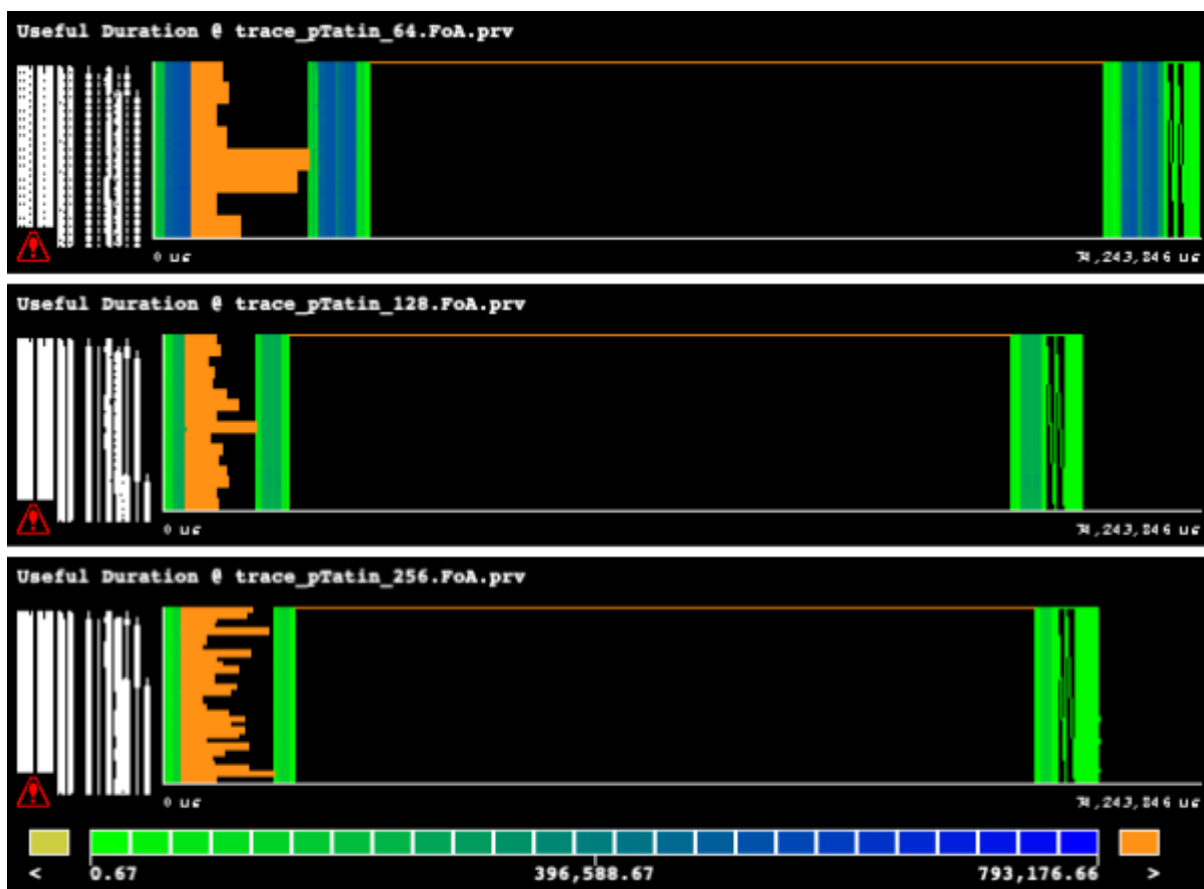
11.3. Performance data acquisition

The code owners have provided Extrae traces for a strong scaling experiment with 64, 128, and 256 MPI ranks, respectively. The code uses MPI and HIP to use GPUs. However, as of today, Extrae does not support tracing HIP events. The analysis therefore will be limited to performance issues related to the parallelization with MPI.

Initially, the size of traces was impractically large. The code was reconfigured to do only a fixed, small number of iterations rather than iterate to convergence. This renders the results of the benchmark runs physically inaccurate, but preserves the characteristics of the code with regard to parallel performance. Even with this mitigation actions, the size of the trace data is very large and a full analysis of all traces could not be finished in time for this deliverable.

The initialization and finalization phases of the code are very short. Therefore the whole application is taken to be the focus of analysis (FoA).

The structure of the application is illustrated in Figure 11-3 below. For most of the runtime, only the first MPI rank does computations (long orange line at top of panels), while all other ranks wait in MPI for its completion (large central black area). The duration of this section of the code remains constant with increasing number of ranks. The areas to the left and to the right thereof, however, reduce in duration with increasing scale.



11.4. POP performance efficiency metrics

We have calculated POP performance efficiency metrics using the simple (MPI-only) model (see Table 11-1 below). Note, that due to the preliminary nature of the analysis, we have not further decomposed communication efficiency into transfer and serialisation efficiency.

Table 11-1 POP performance efficiency metrics

	MPI Processes/GPUs		
	64	128	256
Global efficiency	20%	11%	6%
Parallel efficiency	20%	12%	10%
Load balance	22%	13%	10%
Communication efficiency	90%	93%	94%
Computational Scalability	100%	94%	55%

As expected from the structure of the FoA, load balance is very low at 22% at 64 MPI ranks and drops to 10% at 256 MPI ranks. The reason for this large imbalance is the central region of the code, where only the first MPI rank does any calculations.

Communication efficiency is above 90% for all ranks and thus very good.

The computational scalability decreases with number of ranks, i.e. actual computations (as opposed to MPI communication) take more and more time. The effect is so large, that almost double the time is spend in computations at 256 MPI ranks with respect to the reference at 64 ranks.

No further analysis can be presented at this time.

12. Code ExaHyPE 2

12.1. Brief description of code and use-case

ExaHyPE is an engine for solving hyperbolic PDE systems using high-order discontinuous Galerkin discretization on tree-structured adaptive Cartesian grids. Of particular relevance to ChEESE-2P is ExaSeis, a package to solve seismic wave propagation with kinematic and dynamic rupture earthquake sources on curvilinear meshes. ExaSeis is used in Pilot Demonstrators PD1 (Extreme-scale modeling of seismic hazards) and PD4 (Complex multi-source tsunami modeling). A special focus will be on realizing uncertainty quantification and hazard assessment workflows with ExaSeis.

The ExaHyPE core is developed together by the groups of Tobias Weinzierl (at Durham University, UK) and Michael Bader (TUM). The Durham group leads the work on the underlying Peano framework (for parallel adaptive mesh refinement and overall parallelization approach). TUM is primarily concerned with the development of the ExaSeis solver and with optimization of kernels; in ChEESE-2P a strong focus is put on porting and optimizing ExaHyPE (and ExaSeis) for GPU architectures.

For the performance audit, we consider the LOH1 benchmark, a common benchmark to verify correctness of seismic wave propagation codes. LOH1 models a sediment layer over a base material. The uniform Cartesian grid chosen for this benchmark contains 531,441 cells with a discretization order of 7. Each cell holds 9 unknowns and 3 auxiliary variables. The auxiliary variables remain constant over time. With discretization order 7, this leads to a problem with 3.2 billion degrees of freedom.

12.2. Scaling behavior

The performance optimization and evaluation performed in ChEESE-1P have revealed a bottleneck in the tree-oriented parallelization approach in the underlying parallel AMR framework Peano. This has led to a major rewrite of Peano and ExaHyPE. The resulting version ExaHyPE 2 is now also more closely integrated into the Peano framework. The ExaSeis package has been ported to the ExaHyPE 2 engine. Its ADER-DG kernels are already optimized for CPUs. For ExaHyPE 2, the focus of ChEESE-2P is now on GPU parallelism, following an OpenMP target offload approach.

The performance assessment in this Deliverable is thus the first evaluation of the ADER-DG seismic wave propagation solver ExaSeis on GPU architectures.

Parallelization approach: Peano/ExaHyPE follow a tree-oriented parallelization approach. MPI partitions are created via space-filling orders on the tree-structured Cartesian grid. Within each MPI rank, the adaptive spacetree can be traversed by multiple threads. A so-called *enclave solver* identifies cells in the adaptive mesh which are of “lower priority” (e.g., cells which are not about to be refined or coarsened or (MPI) boundary cells within the timestep). These *enclave cells* can be offloaded as enclave tasks. To avoid too fine grain tasks, the enclave solver collects these enclave cells until a user-defined threshold is reached and fuses them to a so called “meta cell” which is then offloaded to the GPU. If the mesh is large enough this yields batches of meta cells which are dynamically scheduled and offloaded to all available GPUs.

Known limitations: (1) The threshold for fusing enclave cells is currently user-defined and remains constant over the entire simulation. Finding suitable thresholds requires testing for the specific number of GPUs and also problem size. In a strong scaling scenario, when adding more GPUs, some partitions might not be able to reach the threshold for fusing enclave tasks – GPUs remain underutilized then, which results in a significant drop-off in the strong scaling behavior.

(2) During grid creation, ExaHyPE 2 creates a cascade of spacetrees. It starts with a single space tree assigned to one spacetree-traversal task worked on by one thread. A load balancer takes responsibility to split the spacetree into smaller spacetree during grid creation. Then, multiple spacetree-traversal tasks can start to further create the grid (if applicable, using mesh refinement) in parallel. After grid creation, but before initialization of the problem, the load balancer makes some final adjustment to get the best-balanced load possible. For a static (possibly) adaptive mesh, the load balancer then remains idle during the simulation. Currently, this load-balancing strategy is not yet parallelized, and the grid is completely created in one spacetree. The load balancer only kicks in after the grid is created, which limits the size of the grid that can be created (due to compute time and memory). We excluded this sequential part from the performance analysis.

As an application specific performance metric, the number of element updates per seconds is used. Each rank uses 16 CPU threads per rank

available on one node to speed up the fusion of the enclave cells. This number has been fine-tuned and proved to yield the best performance. To setup a strong scaling test a big enough problem size is to be found which can fill at least one GPU's memory. Since 16 CPU threads are necessary per rank/GPU, only one GPU per node is used. Hence, scaling the number of GPUs is achieved by increasing the number of nodes.

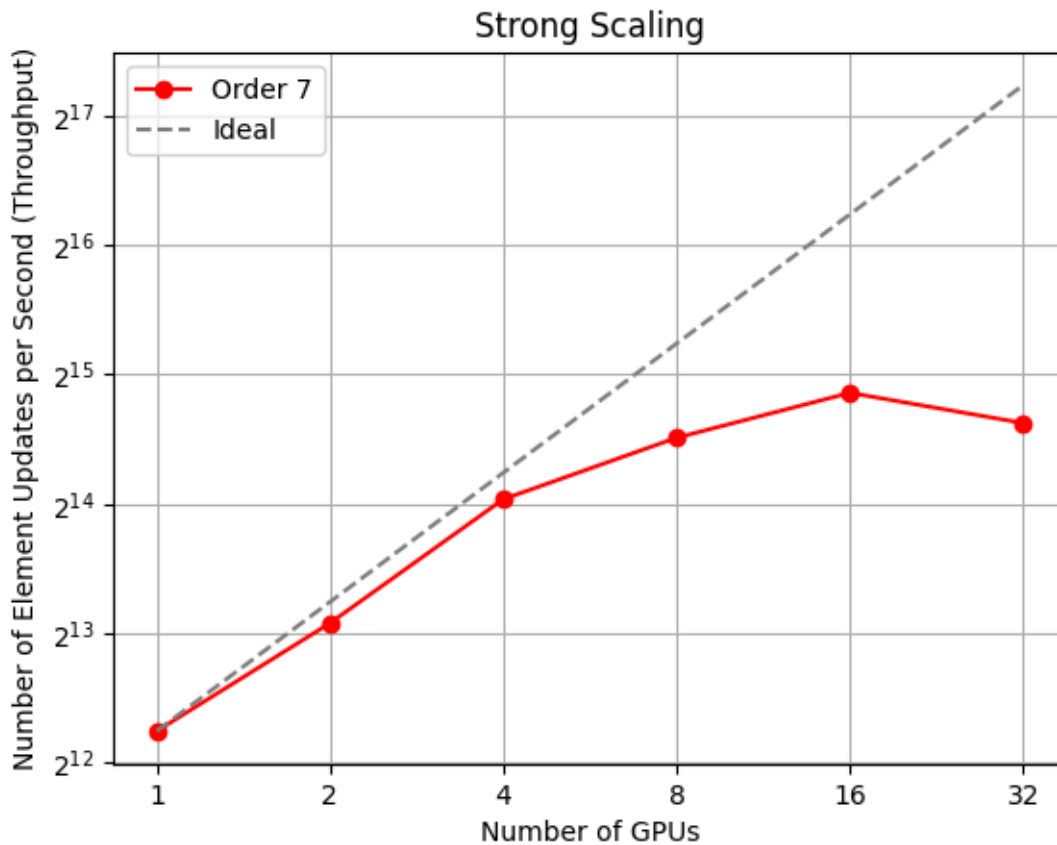


Figure 12-1 Strong scaling curve for main computational loop (1 GPU used per node).

12.3. Performance data acquisition

Score-P has been used to obtain traces and performance metrics. The maximal overhead Score-P introduced in the simulation has been calculated to be 2.90%. The used compiler is NVHPC in version 23.5. To also collect OpenMP events with Score-P a patched Score-P installation was needed to support the OMPT interface with NVHPC 23.5.

The focus of analysis (FoA) is the computation of the timestep in the ExaSeis application, i.e., the main loop of the wave propagation kernel. All acquired data can be reproduced using the code tag 2024ChEESECodeAudit.

12.4. POP performance efficiency metrics

To assess the parallel efficiency of the application, the *multiplicative hybrid model* was chosen. The efficiency metrics were calculated using the advisor plugin of CubeGUI 4.8.1. Hardware counters were measured using Linux Perf. The following Table 12-1 shows the resulting efficiency metrics for the focus of analysis.

Table 12-1 POP performance efficiency metrics

POP hybrid multiplicative metrics for ExaSeis-FoA-(timestep)						
	1	2	4	8	16	32
Hybrid Parallel Efficiency	0.88	0.79	0.79	0.59	0.4	0.21
Hybrid Load Balance Efficiency	0.91	0.86	0.83	0.71	0.49	0.25
Hybrid Communication Efficiency	0.97	0.92	0.95	0.83	0.8	0.81
MPI Parallel Efficiency	0.94	0.89	0.89	0.89	0.88	0.88
MPI Load Balance	0.94	0.89	0.89	0.89	0.88	0.88
MPI Communication Efficiency	1.0	1.0	1.0	1.0	1.0	1.0
OpenMP Parallel Efficiency	0.93	0.89	0.89	0.66	0.45	0.23
OpenMP Load Balance Efficiency	0.96	0.96	0.93	0.8	0.56	0.29
OpenMP Communication Efficiency	0.97	0.92	0.95	0.83	0.8	0.81
Computational Scaling	1.0	0.95	0.92	0.87	0.82	0.67

Overall, the *Hybrid Parallel Efficiency* significantly declines with scale. The trend highly correlates with the strong scaling plot from section 1.2. From 1 to 2 GPUs there is a 9% loss of parallel efficiency. From 2 to 4 the efficiency remains the same. Beyond 4 GPUs, there is a loss of efficiency of approximately 20% each time the number of GPUs is doubled.

Only a small part of the inefficiency stems from the MPI parallelization as indicated by the almost constant *MPI Parallel Efficiency* around 88%. Most inefficiencies stem from the OpenMP parallelization as *OpenMP Parallel Efficiency* declines down to 23%. Most of the OpenMP inefficiency is caused by a significant load imbalance as indicated by the *OpenMP Load Balance Efficiency* declining down to 29%.

It is important to note, that both the *Hybrid Parallel Efficiency* and the *OpenMP Parallel Efficiency* include all organizational tasks, such as the creation and grouping of enclave tasks, which is executed on CPUs. Many of these organizational tasks are still unoptimized, and, in particular, not

parallelized. The respective values are thus not representative of the task distribution to GPUs – the latter cannot be analyzed with Score-P.

12.5. Communication

The metrics report no issue regarding MPI communication as the *MPI Communication Efficiency* is constantly at 100%. All loss of efficiency by communication is attributed to OpenMP as indicated by the *OpenMP Communication Efficiency*. This efficiency captures waiting time caused by dependencies between threads. In total the different threads perform billions of MPI_Iprobe and MPI_Test calls, which on average are shorter than 1 microsecond. This results in huge overhead when tracing the application and hindered us to obtain traces for runs with more than 1 node.

12.6. Computational scalability

The computational scalability of the application is reasonable. At the scale of 32 nodes the *computational scaling* metric reports a value of 67%, which means that the useful computation time is almost 1.5 times longer compared to the reference run on a single node. The longer useful computation time mostly correlates with a larger number of instructions as the *instruction scaling* metric shows. So there seem to be some redundant computations that increase with scale.

12.7. Observations and recommendations

The most salient observation is the very poor *OpenMP Parallel Efficiency* of 23% at the scale of 32 GPUs, which reports a substantial load imbalance between different OpenMP threads. We suspect that this imbalance is already caused by the administrative tasks to orchestrate the OpenMP offloading to GPUs – further analysis and optimization of these orchestration tasks will be necessary.

Moreover, the *instruction scaling* indicates that redundant operations are performed on the CPU that are not necessary in the reference run with a single GPU on a single node. Here we again suspect influence of non-optimized orchestration tasks and suggest investigating the cause of this instruction overhead.

Appendix: POP performance efficiency metrics

This chapter briefly outlines the POP metrics. See also the training material¹⁸ on the website of the POP project.

A.1. Pure-MPI metrics

This section refers to non-hybrid parallelization, i.e., typically pure MPI (pure OpenMP would also fit in this category but does not exist for any ChEESE code). For single environments, all POP auditors furnish the same set of metrics that quantify inefficiencies at process/thread level. The top-level Global Efficiency (GE) metric, which measures the overall quality of parallelization. It is composed of two sub-components: Parallel Efficiency (PE; efficiency of parallelising over processes) and the Computational Scalability (CS; scaling of an application with the number of processors/threads from a reference case and considering only the scaling of the computing phases). These two sub-metrics result, in turn, from contributions of several lower-level child metrics. Table 13-1 and Figure 13-1 show the different levels of metrics involved in single environment evaluations. Note that metrics related to Parallel Efficiency are in the range 0 to 1 (or from 0 to 100%), whereas scalability metrics can be larger than 1 (than 100%) because they are normalized with the reference case.

Table 13-1 Definition of the multiplicative POP metrics. Labels refer to the different childs in Figure 13-1.

Level	Metric	Acronym	Definition
1	Global Efficiency	GE	Measures the overall quality of parallelisation. Computed as: $GE = PE \times CS$
1.1	Parallel Efficiency	PE	Measures the fraction of time spent in computation (as opposed to time spent in MPI). Calculated as: $PE = LB \times CE$

18

<https://pop-coe.eu/further-information/learning-material/pop-standard-hybrid-metrics-for-parallel-performance-analysis>

1.1.1	Load Balance Efficiency	LB	Measures the distribution of work across processors. Computed as the ratio between average computation time and maximum computation time (across all processes)
1.1.2	Communication Efficiency (sometimes also Orchestration Efficiency)	CE	Measures the loss of efficiency by communication (processes waiting at communication points or transferring data) or orchestration of IO and GPU task. Computed as $CE = SE \times TE$
1.1.2.1	Serialization Efficiency	SE	Measures loss of efficiency due to waiting time in MPI calls where no data is transferred
1.1.2.2	Transfer Efficiency	TE	Measures inefficiencies due to time in data transfer. Computed as the ratio between total runtime on ideal network and total runtime on real network
1.2	Computational Scalability (sometimes also Computation Efficiency)	CS	Measures the scaling actual computation of an application for a given number of processors/threads relative to a reference case. Loss of CS can come either from contention for shared resources or because dividing work over additional processes/threads increases the total computation.

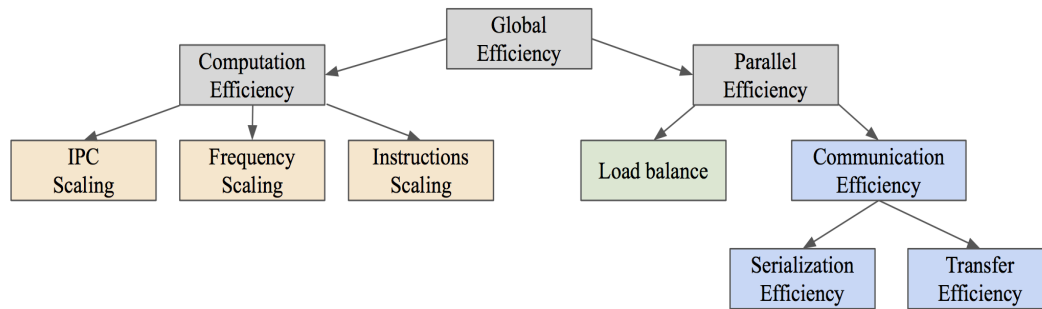


Figure 13-1 POP Standard Metrics for Parallel Performance Analysis.

A.2. Hybrid environment metrics¹⁹

In the case of hybrid parallel programs, i.e. combining multiple parallel programming paradigms in a single program to harness the full power of modern HPC systems as for instance MPI+OpenMP, or MPI+CUDA, POP uses two complementary approaches of parallel performance metrics: multiplicative (Figure 13-2) and additive (Figure 13-3).

A.2.1. Multiplicative approach

In this option, Parallel Efficiency is also split as load balance and communications, but further child metrics are defined as an additional sub-level. For example, Hybrid Load Balance Efficiency is computed as the product of the MPI Load Balance Efficiency times the OpenMP (CUDA) Load Balance Efficiency, the later intended as the time cost of imbalance of useful computation inside OpenMP (CUDA) regions.

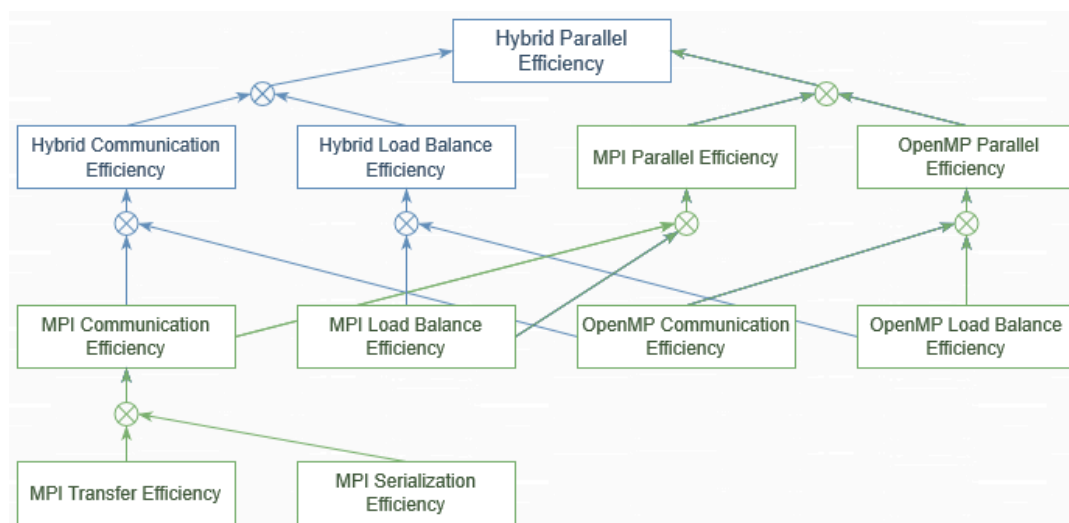


Figure 13-2 POP Multiplicative Hybrid Model.

¹⁹

<https://pop-coe.eu/further-information/learning-material/pop-standard-hybrid-metrics-for-parallel-performance-analysis>

A.2.2. Additive approach

The additive approach considers processes (MPI) and threads (OpenMP) separately and splits the Parallel Efficiency into Process Efficiency and Thread Efficiency: $\text{Parallel Efficiency} = \text{Process Efficiency} + \text{Thread Efficiency} - 1$.

Moreover, sub-metrics in the tree are now additive in the sense that the sum of the child inefficiencies is equal to the parent inefficiency. Process efficiency metrics analyse inefficiencies at a process level whereas thread efficiency metrics measure inefficiencies occurring at a thread level (including any inefficiency from MPI inside OpenMP regions). The additive model can be also applied to pure MPI codes, and the difference is on the weight assigned to load balance and serialization.

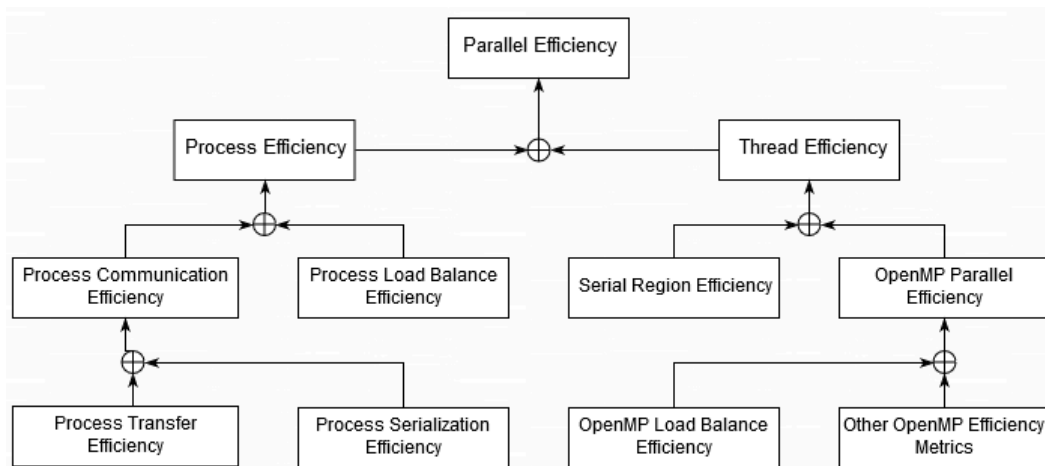


Figure 13-3 POP Additive Hybrid Model.